LIVE DISTRIBUTED OBJECTS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Krzysztof Jan Ostrowski

August 2008

© 2008 Krzysztof Jan Ostrowski

ALL RIGHTS RESERVED

LIVE DISTRIBUTED OBJECTS

Krzysztof Jan Ostrowski, Ph.D.

Cornell University 2008

Distributed multiparty protocols such as multicast, atomic commit, or gossip are currently underutilized, but we envision that they could be used pervasively, and that developers could work with such protocols similarly to how they work with CORBA/COM/.NET/Java objects. We have created a new programming model and a platform in which protocol instances are represented as objects of a new type called *live distributed objects*: strongly-typed building blocks that can be composed in a type-safe manner through a drag and drop interface. Unlike most prior object-oriented distributed protocol embeddings, our model appears to be flexible enough to accommodate most popular protocols, and to be applied uniformly to any part of a distributed system, to build not only front-end, but also back-end components, such as multicast channels, naming, or membership services.

While the platform is not limited to applications based on multicast, it is replication-centric, and reliable multicast protocols are important building blocks that can be used to create a variety of scalable components, from shared documents to fault-tolerant storage or scalable role delegation. We propose a new multicast architecture compatible with the model and designed in accordance with object-oriented principles such as modularity and encapsulation. The architecture represents multicast objects as compositions of smaller objects, glued using a decentralized membership mechanism. The benefits include support for heterogeneous networks by allowing a single multicast protocol instance to simultaneously leverage different mechanisms in different parts of the Internet.

Finally, we describe and evaluate a scalable reliable multicast prototype that incor-

porates a novel approach to scaling with the number of protocol instances by leveraging regular patterns of overlap. The system achieves exceptional performance, scalability, and stability in the presence of perturbations, in part thanks to a novel application of techniques such as priority scheduling or pull-based processing. We describe a previously unnoticed relationship between memory overheads and scheduling and the performance and scalability of a reliable multicast protocol. Our results demonstrate that to build a new global Web operating system that the live objects vision leads to, the distributed protocol and the local system architecture cannot be treated in isolation.

BIOGRAPHICAL SKETCH

Krzysztof Ostrowski is originally from Poland. He received his Bachelor of Science in Mathematics and Computer Science from the University of Warsaw in Poland in 1998, and his Master of Science in Computer Science from the same university in 2001. In August, 2008 he received his Ph.D. from Cornell University.

To my beloved Yejin.

ACKNOWLEDGEMENTS

First and foremost, I am deeply grateful to Ken Birman for being an extremely supportive advisor, open to my ideas and encouraging, and for enabling me to develop and to pursue my vision. Ken is a great teacher of assertiveness, optimism, and confidence in one's work, and has motivated me to stay firmly on the course that I have taken, and to be determined in pursuing my goals despite the difficulties and obstacles I met along the way. Ken has created countless opportunities for my work to be noticed and recognized by people in the research community and in the industry. His critical feedback and guidance were invaluable in helping me learn how to efficiently communicate my ideas. Ken has also very much inspired me with his passion for building and measuring real systems and with his commitment to tackling concrete real-world problems, even if hard and intimidating. I especially appreciate having been able to benefit from Ken's tremendous practical knowledge and experience while working together on understanding the performance of QSM and refining the design of the system.

I am also profoundly grateful to Danny Dolev, whose deep insight and wisdom at both academic and personal level not only enriched my research, but also energized me with high morale. Danny has been a fantastic collaborator with an extraordinary ability to grasp the essence of one's ideas and to quickly point out the subtleties involved in realizing them. While working together on the foundations of Properties Framework, he challenged and guided my thinking to help me steer around pitfalls; the architecture would not have been possible without him. I am tremendously lucky to have been able to work with Danny.

I am also much indebted to Jong-Hoon Ahnn for all the great work he has done for the live objects platform. John has provided a great amount of feedback that has been extremely helpful in improving the stability and usability of the platform, and his help was essential to the success of our demos. He has also developed many components,

without which the platform would be incomplete.

I am sincerely grateful to Paul Francis, Andrew Myers, Anil Nerode, and Robbert van Renesse, who have been members of my special committee, for their many insightful comments and suggestions regarding my dissertation, research vision, style of presentation, and career in general. I would like to especially thank Andrew and Robbert for reading my dissertation so thoroughly; their detailed feedback has been very helpful in improving the structure and style of my writing.

I would like to also thank Hussam Abu-Libdeh, Mahesh Balakrishnan, Lars Brenna, Daniel Freedman, Lakshmi Ganesh, Maya Haridasan, Chi Ho, Ingrid Jansch-Porto, Tudor Marian, Amar Phanishayee, Stefan Pleisch, Robbert van Renesse, Michael Siegenthaler, Yee Jiun Song, Ymir Vifgusson, Werner Vogels, Einar Vollset, and Hakim Weatherspoon, who currently are, or who have once been members of our group, collaborators, and visitors, for listening to my practice talks, reading drafts of my papers, and for their many helpful comments and suggestions. They have been the greatest colleagues I ever worked with, and I am proud to have been a part of the team.

I would like to thank Lars Brenna, Dmitri Chmelev, Adam Davis, Richard Grandy, Ankur Khetrapal, Daniel Margo, Chuck Sakoda, Weijia Song, Zhen Xiao, as well as all the students in CS514 in Spring 2007 and 2008, for being the users of QSM and live objects. I am especially indebted to Lars for pointing out the bugs in QSM's support for unmanaged applications.

I am grateful to Ernie Davis for believing in our technology and convincing Cornell to file patent applications for QSM and live objects, and to Kathleen Chapman for the hard work we went through together preparing these patent applications.

I would like to also thank Brian Bershad, Marin Bertier, Jason Bryantt, Antonio Carzaniga, Michael Caspersen, Ranveer Chandra, Wei Chen, Gregory Chockler, Jon Currey, Alan Demers, Kathleen Fisher, Davide Frey, Johannes Gehrke, Rachid Guer-

raoui, Chang Heng, Robert Hillman, Mingsheng Hong, Ray O. Johnson, Idith Keidar, Anne-Marie Kermarrec, Annie Liu, John Manferdelli, Milan Milenkovic, Benjamin Pierce, Josh Rattner, Fred Schneider, Gennady Staskevich, Daniel Sturman, Eric Suss, Marvin Theimer, Toste Wallmark, Calvin Wong, Zheng Zhang, and all the anonymous COMSWARE, DEPSA, ECOOP, HotOs, ICDCS, ICWS, Internet Computing, IPTPS, JWSR, NCA, NSDI, OSDI, SOSP, and STC reviewers for comments and suggestions. I would like to thank Ranveer Chandra for the opportunity to give a talk and to meet his colleagues researchers in Microsoft Research labs at Redmond.

I would like to present my special gratitude to Bill Hogan for all his help and advice in administrative matters. I would like to also very much thank Ben Atkin for being my mentor at Cornell, and a great friend. Ben has been like an older brother, and made me feel at home in Ithaca.

I am grateful to my parents Tadeusz and Maria for their love and sacrifice, and to my brother Piotr for being a wise, supportive and loving brother, and for being there for me whenever I needed him. Last, but not least, I would like to thank my wife Yejin for her support, for the time she spent reading early drafts of my papers and listening to my practice talks to help me improve my presentation and writing, for the many valuable technical comments she made, and most importantly, for being so wise and charming at the same time.

This work was supported by DARPA/IPTO under the SRS program, by the Rome Air Force Research Laboratory, AFRL/IF, and by AFOSR (under AF TRUST). Additional support was provided by Intel and NSF. Any opinions, findings, or recommendations presented in the following pages, however, are my own, and do not necessarily reflect the views of DARPA/IPTO, AFRL, AFOSR, Intel, or NSF.

TABLE OF CONTENTS

Biographical Sketch iii
Dedication iv
Acknowledgements v
Table of Contents viii
List of Figures xi
List of Code xiv
List of Symbols xv

1 Introduction **1**
1.1 The Limited Adoption of Distributed Computing 7
1.2 The Active Web 9
1.3 The Emerging Paradigm 15
1.4 Prior Work 25
1.4.1 Embedding 25
1.4.2 Composition 32
1.4.3 Replication 38
1.5 Contributions 47

2 Programming **51**
2.1 Programming Model 51
2.1.1 Objects and their Interactions 51
2.1.2 Defining Distributed Types 61
2.1.3 Constraint Formalisms 66
2.1.4 Language and Type System Embeddings 72
2.1.5 Construction and Composition 79
2.1.6 Deployment Considerations 86
2.2 Prototype Implementation 88
2.2.1 OS Embedding via Drag and Drop 88
2.2.2 Language Embedding via Reflection 92

3 Extensibility **102**
3.1 Introduction 103
3.1.1 Terminology 103
3.1.2 Technical Challenges 104
3.1.3 Design Principles 106
3.1.4 Our Approach 108
3.2 Architecture 111
3.2.1 The Hierarchy of Scopes 111
3.2.2 The Anatomy of a Scope 114
3.2.3 Hierarchical Composition of Policies 117
3.2.4 Communication Channels 121
3.2.5 Constructing the Dissemination Structure 124

3.2.6 Local Architecture of a Dissemination Scope 129
3.2.7 Sessions 132
3.2.8 Incorporating Reliability and Other Strong Properties 135
3.2.9 Hierarchical Approach to Reliability 139
3.2.10 Building the Hierarchy of Recovery Domains 143
3.2.11 Recovery Agents 151
3.2.12 Modeling Recovery Protocols 151
3.2.13 Implementing Recovery Domains with Agents 162
3.2.14 Reconfiguration 164
3.2.15 Reliability and Consistency 166

4 Performance **171**
4.1 Introduction 171
4.1.1 Application Scenario 171
4.1.2 Exploiting Overlap 174
4.2 Implementation 182
4.2.1 Protocol 182
4.2.2 Protocol Stack 192
4.3 Evaluation 196
4.3.1 Memory Overheads on the Sender 199
4.3.2 Memory Overheads on the Receiver 205
4.3.3 Overheads in a Perturbed System 206
4.3.4 Overheads in a Lightly-Loaded System 211
4.3.5 Per-Group Memory Consumption 213
4.3.6 Extrapolating the Results 217
4.3.7 Discussion 228
4.3.8 Conclusions 233

5 Future Work: Properties Framework **237**
5.1 Protocols 238
5.2 Properties 239
5.3 Rules 240
5.4 Performance 243

6 Conclusions **245**
6.1 Component Integration 246
6.1.1 Types 251
Formalisms to Express Reliability and Security Properties . . . 252
Ontology and Inheritance for Constraint Formalisms 253
Proof-Carrying Annotations for Provable Composition 254
Incorporating Theorem Provers in a Scalable Manner 255
6.1.2 Components 256
Automatically Finding and Downloading Libraries 256
Managing Multiple Versions of Objects and Types 257

Conversions Between Binary-Incompatible Values 257
Synchronous vs. Asynchronous Object Interactions 258
6.1.3 Security 260
Assigning Proxies to Isolated Application Domains 260
Scalable Components for Implementing Security 261
Integration with Existing Security Infrastructure 262
Leveraging Dynamic Composition and Reflection 262
Managing Distributed Object-Object Connections 263
Just-in-Time Typing Support for Library Objects 264
6.1.4 Resources 265
Intelligent Activation and Deactivation of Proxies 265
Proxies with Different "Degrees" of Connectivity 267
Managing Resources Required for Correct Operation 268
6.1.5 Language 269
Object Specifications Embedded in Object References 269
High-Level Constructs in our Composition Language 270
Controlling Inflation of the Size of Object References 271
6.2 Multicast Infrastructure 272
6.2.1 State 281
Limiting Dependence on Costly Flavors of Multicast 281
Backup and State Persistence for Replicated Objects 283
Using Locally Cached Configuration State in Proxies 284
Synchronizing Replicated State Stored in Documents 285
Modeling Nested Transactions as Object Composition 286
6.2.2 Scalability 286
Making Systems Regular through Clustering of Objects 287
Indexing Schemes for Amortizing Work Across Objects 287
6.2.3 Communication 288
Connecting Clients Separated with NATs and Firewalls 289
Incorporating Mechanisms for Privacy and Anonymity 289
Embedding Content-Based Publish-Subscribe Objects 290
Infrastructure Objects in Support of Ad-Hoc Networks 291
6.2.4 Configuration 291
Application Model Based On Distributed Membership 291
General-Purpose Scalable Role Delegation Framework 293
Self-Discovery and Bootstrapping with Gossip Objects 294
6.3 Summary 294

Bibliography **296**

Glossary **324**

LIST OF FIGURES

1.1 A live distributed object accessed by multiple users. 16
1.2 Live objects can represent low-level or high-level components. 20
1.3 Multiple local "sub-protocols" combined into an Internet-wide structure. 22
1.4 With overlapping ACK trees, nodes can have unbounded indegrees. . . 41

2.1 Live objects are executed by proxies that interact via endpoints. 53
2.2 Applications are composed of interconnected live objects. 59
2.3 A class diagram and a proxy structure of a composite object. 81
2.4 An example of a hybrid live object. 85
2.5 A live object dynamically controlling its own deployment. 87
2.6 Screenshots of the live objects platform in action. 90

3.1 Nodes register for topics with a subscription manager. 103
3.2 Nodes can be scattered across several administrative domains. 104
3.3 A hierarchical decomposition of the set of publishers amd subscribers. 109
3.4 An example hierarchy of management scopes in a game. 113
3.5 A scope is controlled by a separate entity called scope manager. 114
3.6 Standard roles and patterns of interaction between scopes. 116
3.7 Channels created in support of policies defined at different levels. . . . 119
3.8 A forwarding policy as a code snippet. 119
3.9 Forwarding graphs for different topics are superimposed. 120
3.10 A channel split into sub-channels and a corresponding filter tree. . . . 121
3.11 A channel may specify a protocol or decompose into sub-channels. . . 122
3.12 Channel policies are realized by per-subchannel filters. 122
3.13 A distributed scope delegates or replicates channels to its members. . . 123
3.14 An example hierarchy of scopes with cascading subscriptions. 125
3.15 Channels created by the policies based on subscriptions. 126
3.16 Channels are delegated. 126
3.17 *B* and *D* contact *Q* and *R* to create channels. 126
3.18 The flow of messages and the channels in the scenario of Figure 3.17. . 128
3.19 The architecture of a leaf scope in the most general scenario. 129
3.20 Example architectures with the SM and the local controller. 130
3.21 Internal architecture of the application process 136
3.22 Processing messages on the send and receive paths. 137
3.23 The similarities between a hierarchical dissemination and recovery. . . 139
3.24 Membership information is passed one level down the hierarchy. 147
3.25 A hierarchy of recovery domains that clusters nodes based on interest. . 149
3.26 A group of peers in a reliable protocol. 152
3.27 A peer modeled as a component living in abstract environment. 153
3.28 RMTP expressed in our model. 154
3.29 Another way to express RMTP. 156
3.30 A hierarchy of recovery domains and agents. 158

3.31 Agents stacks that are not vertical. 162
3.32 A node as a “container” for agents. 163

4.1 A fragment of the network partitioned into three regions. 175
4.2 Groups overlap to form regions. 176
4.3 Inclusion relation between multicast groups in a mashup. 178
4.4 Multicast groups in data centers forming regular hierarchies. 179
4.5 A decomposition of an irregular system into perfect hierarchies. 181
4.6 Most traffic a node sees is concentrated in 2 regions. 182
4.7 A two-level membership service used in QSM. 184
4.8 One level of indirection for multicast in QSM. 185
4.9 An example hierarchy of recovery domains for a group in QSM. 187
4.10 Recovery domains are implemented by agents hosted at “leader” nodes. 188
4.11 Token rings in partitions enable recovery from nearest neighbors. . . . 189
4.12 Responsibility for caching is subdivided among partitions in the region. 189
4.13 QSM core thread controls three event queues. 193
4.14 QSM uses time-sharing with a fixed quanta per event type. 194
4.15 QSM assigns priorities to types of I/O events. 194
4.16 In a pull architecture, data to send is produced in a lazy fashion. 195
4.17 Elements of the protocol stack in QSM form trees rooted at sockets. . . 196
4.18 Max. sustainable throughput. 199
4.19 CPU utilization as a function of multicast rate. 200
4.20 The percentages of the profiler samples taken from QSM and CLR DLLs. 200
4.21 Memory overheads on the sender. 200
4.22 Time spent allocating byte arrays in the application, and copying. . . . 201
4.23 Memory used on sender and the # of multicast requests in progress. . . 201
4.24 Token roundtrip time and an average time to acknowledge a message. . 202
4.25 Varying token circulation rate. 203
4.26 More aggressive cleanup with O(n) feedback in the token and in ACKs. 204
4.27 More work with O(n) feedback and lower rate despite saving on memory. 204
4.28 Instability with O(n) feedback. 204
4.29 Varying the number of caching replicas per message in a 192-node region. 205
4.30 As the # of caching replicas increases, the throughput decreases. 206
4.31 Throughput in the experiments with a perturbed node. 207
4.32 Average packet loss observed at the perturbed node. 207
4.33 Memory usage at a perturbed node. 208
4.34 Number of messages awaiting ACK with perturbations. 208
4.35 Token roundtrip time and the time to recover in the ”sleep” scenario. . 209
4.36 Token roundtrip time and the time to recover in the ”loss” scenario. . . 209
4.37 Histogram of maximum alarm delays on the receivers. 210
4.38 Histogram of maximum alarm delays in 1s intervals, on the sender. . . 210
4.39 Growing memory usage on sender and flat usage on the receivers. . . . 211
4.40 Number of unacknowledged messages and average token roundtrip time. 212
4.41 The send-to-receive latency for varying rate. 213

4.42 Alarm firing delays on sender and receiver. 213
4.43 Number of messages pending ACK and token roundtrip time. 214
4.44 Memory usage grows with the # of groups. 214
4.45 Time spent in the CLR code. 215
4.46 Throughput decreases with the number of groups. 215
4.47 Cumulative distribution of the multicast rates. 216
4.48 Token roundtrip times for 4K and 7K groups (cumulative distribution). 216
4.49 Intervals between subsequent tokens (cumulative distribution). 218
4.50 Verifying the model: predicted vs. actual memory usage. 221
4.51 Verifying the model: predicted vs. actual CPU utilization. 223
4.52 Dependency of the token roundtrip time on CPU usage. 224
4.53 Verifying the model: improved predicted vs. actual CPU utilization. . . 224
4.54 A failed throughput prediction. 225
4.55 A correct throughput prediction. 226
4.56 Throughput predictions based on Equations 4.10 and 4.11. 227
4.57 Throughput predictions modified to account for memory limit. 228
4.58 A variety of forces controlling system behavior form a vicious cycle. . 229
4.59 Breaking the vicious cycle by controlling memory use. 229
4.60 Breaking the vicious cycle by controlling scheduling. 230
4.61 Combined send rate oscillates. 232
4.62 An experiment with a "freezing" receiver. 232
4.63 On reconfiguration following a crash, changes take time to propagate. . 233

5.1 Using a protocol generated by PF within the live objects platform. . . . 238
5.2 On the end hosts, properties are linked to application state and actions. 239
5.3 Protocols are modeled as distributed transformations on sets of properties. 240
5.4 An instance of a *property* is associated with each domain. 240
5.5 Collecting and aggregating information in the cleanup example. 241
5.6 Making and disseminating decisions in the cleanup example. 241
5.7 The effect of churn on the performance of the protocol from Code 5.2. . 244

LIST OF CODE

2.1 A live object event handler code in a C#-like language. 73
2.2 An object reference using a shared document template. 80
2.3 An example live object reference for a custom protocol. 84
2.4 A portable reference to the hybrid object of Figure 2.4. 85
2.5 An example of a correct use of the "registry" pattern. 86
2.6 Declaring a live object type from an annotated .NET interface. 93
2.7 Declaring a live object template by annotating a .NET class. 99
5.1 An example program in our Properties Language: "Cleanup". 242
5.2 An example program in our Properties Language: "CoordinatedPhases". 243

LIST OF SYMBOLS

$\propto$	the relation of "matching" for pairs of endpoint types	63
$\rightsquigarrow$	events from the former endpoint are valid events of the latter	63
$\lhd$	the relation of subtyping	63
$\overset{*}{\Rightarrow}$	"weaker than" relation on sets of behavioral constraints	63
$\overset{(a)}{\Rightarrow}$	logical consequence in formalism a	63
$\overset{*}{\preceq}$	"can substitute" relation for sets of endpoints	65
$\mathcal{P}(X)$	the power set of X	177

Chapter 1
Introduction

The work reported in this dissertation, and the long-term research agenda of the author, are motivated by the desire to make distributed computing accessible to the end user, and in particular by the vision of a world, in which distributed multiparty protocols, such as leader election, reliable multicast, atomic transactions, gossip, and consensus, are used pervasively. We envision that creating new instances of such protocols would be as easy and natural as defining new variables, and that developers could treat distributed protocols simply as reusable building blocks that can be added to their applications as needed, anywhere they seem to naturally fit as a functional part of a larger system, much in the same way one uses queues, lists, dictionaries, files, or sockets whenever necessary.

We'd like to enable developers to work with distributed multiparty protocols as one of the common programming abstractions, cleanly embedded into their development environment, and supported by the operating system, language, and programming tools. In particular, we'd like to enable building applications that use distributed protocols by leveraging the same kinds of visual rapid application development (RAD) tools based on drag and drop technology as those used today for desktop and client-server applications. We envision that instances of distributed multiparty protocols would be represented as components that can be dragged and dropped from toolboxes and easily connected to one another, and that even users with little programming experience could compose complex distributed systems with ease simply by "mashing up" various building blocks.

For example, we'd like the user to be able to build a shared document by dragging a replicated data structure template from a toolbox onto a design sheet and connecting it to a graphical user interface (GUI) on one end, and a reliable multicast protocol component on the other. The reliable multicast protocol, which would also be dragged from a toolbox, might then need to be connected to an unreliable multicast component and

a membership component, again both dragged from a toolbox or a shared folder with components created by other users, and so on. When an application composed this way is instantiated on multiple machines, a complex distributed system would arise that involves instances of reliable and unreliable multicast, membership, and data structure replication protocols running behind the GUI. Because the distributed protocols used to implement such application would be represented as reusable building blocks, users could build such systems without having to understand the intricacies of these protocols, much in the same way one can currently use sorting functions or priority queues from a standard template library without having to understanding the sorting algorithms or balanced tree data structures used to implement them.

In this dissertation, we describe a programming model and the platform designed to support this vision. The approach is based on the idea of treating instances of distributed protocols much in the same way we treat "objects" in an object-oriented programming language, or components in CORBA [198, 144, 234], .NET [98] or COM [53]. Thus, for example, we would like to be able to talk about "distributed types" of protocols, and consider a distributed transaction and leader election as protocols of a different distributed type. We would like to be able to compose distributed protocols into more complex distributed entities, and do so in a type-safe manner, with type-checking support from the platform. These distributed entities would also have distributed types, and could be recursively composed to form complex strongly-typed distributed applications. We envision that the abstraction could be used pervasively, and that the entire future Web could be built from strongly-typed distributed entities, down to the hardware level.

We do believe this is possible, and, in fact, inevitable, because it reflects the way modern software is built today, and the way developers like to think about their systems. Developers have gotten used to building systems from reusable building blocks that may be developed independently, but can be composed in a type-safe manner thanks to their

strongly-typed interfaces and the underlying "managed" runtime environment, such as Java/J2EE or .NET. Complex functionality can be designed visually, often without writing any code. The paradigm has many benefits: modularity, extensibility, collaborative development, code reuse. Making contracts between components explicit and their code more isolated reduces the risk of bugs resulting from undocumented or implicit assumptions, cross-component behaviors and side effects. So far, distributed systems developers have been denied many of these benefits.

To emphasize this object-oriented perspective on distributed protocols, we'll refer to an instance of any distributed multiparty protocol as a *live distributed object*, or simply a *live object*. The term *live object* thus represents a certain distributed activity: a programming logic executed by a set of software components that might reside on different machines and communicate by sending messages over the network. The term *live object* will refer to all aspects of implementation and execution of a protocol instance, including the data maintained locally on each node by the software components involved in running the protocol instance, as well as any processing performed internally by those components and any messages they might exchange with one another over the network.

At first, the notion of an instance of a distributed protocol and that of an object might seem fundamentally incompatible. One might want to say that protocol instance *implements* an object, or that it *provides access* to an object, but not that a protocol instance *is* an object, the reason being that we typically think of "objects" as encapsulating data, or providing a service, and of protocols as means of accessing a service. However, if we look closely, the differences turn out to be insignificant. A protocol instance, much like an ordinary object, can also encapsulate data: the state maintained by the software components running it. In certain protocols, such as replicated state machines, there is really a single logical unit of data replicated across all nodes and consistently updated, whereas in some others, components running the protocol on different nodes might be in different

states, and consistency might be weaker. At the same time, an ordinary object, like many protocols, might not store any data at all: the object might be a stateless service that performs computation on demand or orchestrates the execution of other services. Similarly, both objects and protocol instances may be either entirely passive, and only respond to external events, or active, and encapsulate an internal thread of execution. If we abandon the data-centric perspective and think of "objects" in an object-oriented language simply as black boxes that might encapsulate certain state and programming logic, and that communicate with their environment by messages, the abstraction matches our live object definition naturally: a live object (a running protocol instance) also encapsulates certain state and logic, and it communicates with its environment via events: those are the events exchanged by the software components running the live object (the protocol instance) with the operating system and application components they are connected to.

The main difference between what we shall call *regular objects* (objects in the same sense as in CORBA, .NET or COM) and our live objects stems from the fact that while a regular object normally exists in a single location, a live object exists in multiple locations at once: at any given time, a live object (a protocol instance) in some sense exists simultaneously on each of the nodes running it. As a result of this distributed existence, the state of a live object is also distributed (parts, replicas, or versions of this state are stored simultaneously in multiple locations), and so are its interactions with the environment. One might exchange messages with a live object by interacting with any of the software components running it, on multiple nodes simultaneously.

Because of this difference, it is more difficult to classify a live object's behavior and assign it a type. While for regular objects, one can reasonably well define their semantics in terms of request-response patterns, in a live object the patterns of interaction might be more complex because they involve exchanges of events at different locations (nodes running the protocol instance), and the set of such locations might vary (nodes might

join and leave the protocol instance). For example, an event sent to a live object on a single node might result in events delivered by the live object to the application on multiple other nodes. Live objects thus do differ from regular objects, and as we'll argue later, trying to completely mask these differences and hide the distributed nature of protocols may have been one of the main reasons for the limited adoption of the past approaches to embedding distributed computing within object-oriented programming.

Despite these differences, one can embed distributed multiparty protocols into the object-oriented programming paradigm in a way that retains what we believe are the key benefits of object-orientation – encapsulation and support for type-safe composition – while respecting the distributed nature of protocols. In this dissertation, we propose a new object-oriented programming model for distributed protocols designed to address this limitation and a prototype development platform supporting it. In contrast to most of the prior approaches to embedding distributed protocols within an object-oriented environment, the model we propose can be applied uniformly from the front-end down to the hardware level, and it is flexible enough to accommodate a wide range of modern protocols, and yet it is easy to use.

In the remainder of this chapter, we discuss step by step the motivation, approach, prior work, and the specific contributions of this dissertation. We begin by arguing that distributed protocols are currently underutilized, and that an emerging class of interactive decentralized applications carries the potential of a pervasive use of this paradigm. In the following chapters, we focus on what we believe are key problem areas that need to be addressed for the paradigm to be adopted on a large scale.

The first of these has just been introduced: prior distributed programming platforms did not treat protocols as components in the same way as components in CORBA, .NET or COM. Our new distributed component integration model and platform designed to enable this are discussed in Chapter 2.

The remaining problem areas are focused on just one type of live objects: reliable multicast channels. As explained later in this chapter, reliable multicast protocols play a special role in our vision, in that they are extremely versatile as building blocks that can underlay implementations of many kinds of higher-level distributed components. Thus, while the overall vision is not limited to reliable multicast or to applications using it, a discussion of reliable multicast objects is highly relevant.

Prior multicast systems typically focused on a single specific protocol instance at a time. One consequence was to impose a single mechanism across the Internet, thereby ignoring necessary forms of heterogeneity. In practice, most systems are comprised of many independent, and often very differently configured, administrative domains. In Chapter 3, we propose a new architecture for building large-scale reliable systems that addresses these limitations. The architecture is inspired by object-oriented design principles, especially encapsulation, isolation, and separation of concerns, and it is compatible with the model discussed in Chapter 2. Another consequence of the single-protocol focus was to largely ignore a crucial performance metric: scalability with the number of protocols running simultaneously. In Chapter 4, we outline a new approach to achieving scalability in this dimension, by leveraging regular patterns of overlap between sets of nodes running the different protocol instances.

Also, prior work on multicast protocols has focused too much on the network. Most designs are evaluated in emulators such as NS [143], via simulations, or on systems like Emulab [309] that emphasis network loss and latency. In Chapter 4, we demonstrate that effects such as memory overheads or scheduling and the interplay between these and the distributed protocol can have serious performance implications. Our results show that in order to build a new global Web operating system that may stand as a logical outcome of pursuing our vision, distributed protocols and local system architecture cannot be treated in isolation.

We believe that the techniques discussed in this dissertation in the context of reliable multicast objects are useful for implementing other types of live objects. Our vision for achieving this is outlined in Chapter 5 and Chapter 6.

1.1 The Limited Adoption of Distributed Computing

At a first glance, the claim that distributed protocols are underutilized may seem preposterous. Many distributed protocols have already been adopted on a massive scale. In particular, technologies such as web services, service-oriented architecture (SOA) [158], or RSS [33] are commonplace, and have been cleanly integrated into programming languages and operating systems. Peer to peer applications are also increasingly popular, and protocols such as BitTorrent [84] and other forms of file sharing have grown to become a major source of traffic on the Internet, with the estimates ranging from 35% and 90% [247, 100] of overall bandwidth, to the point that they spurred a heated debate over *network neutrality* [316].

However, few protocols have been this successful. The web community has so far strictly adhered to the client-server paradigm, and even in the most recently proposed standards, such as BPEL [161] or SSDL [245], the notion of a multiparty computation is limited to a rather shallow concept of a business process among a fixed set of named participants. Higher-level constructs such as groups and subtle synchronization options related to rigorous notions of reliability are absent. The use of distributed protocols by this community is mostly limited to unreliable content dissemination or atomic transactions through a version of the *two-phase commit* protocol [270]. Essentially the same is true of the grid computing community and standards such as Jini [303], and even of the most recent initiatives, such as Croquet [272, 85], which introduce the concept of a replicated object and share some of the goals and techniques with the work reported here. Likewise, although the dynamically evolving field of peer-to-peer computing has

produced many sophisticated protocols, the most widely used of these systems are limited to a rather narrow category of tasks, such as file sharing or content distribution.

Meanwhile, the research community developed a tremendous variety of protocols. There has been a long line of work on replication [124, 123, 35, 44, 1, 267, 136], and many group communication toolkits [125, 42, 222, 95, 295, 145, 20, 13] were designed over the years to support building fault-tolerant systems. The *distributed consensus* [183] problem never ceased to be popular: new protocols are invented even today [178].

Although these kinds of academically interesting protocols may offer provable benefits relative to alternatives, it is rare to see even best of breed solutions in use, to say nothing of widespread use. Instead, sophisticated protocols are used almost exclusively in clusters within large data centers, or to implement transactions in financial institutions. They are buried in layers of heavyweight, proprietary middleware. Systems that use these protocols are typically designed from scratch by small groups of engineers, and reuse is uncommon. Moreover, since their creation involves a degree of "black wizardry", they are expensive to maintain or adapt as a system evolves. When occasionally made available to developers as general-purpose components, they tend to be employed for a great variety of different tasks, often including improper uses for which they were not designed for, so that access to these tools eventually has to be limited [58].

One might argue that the reason why replication, group communication, and other types of complex multiparty distributed protocols with strong semantics have not made their way into the mainstream use is because their sophisticated properties are just not a good match for the Web. This may have been true in the past, but the Web has evolved, and as we shall argue in the following section, the next generation of Internet browsers and operating systems will likely need to use distributed protocols rather heavily to keep up with the demand for interactive "live" experience across large numbers of users.

1.2 The Active Web

A brief look at the evolution of the Web helps us identify a number of apparent trends. A premise of the work reported here is that the demand for shared, interactive, real-time experience across large groups of users is pushing data and computation away from the costly servers in data centers, and towards clients, to eliminate the central point of bottleneck, and to reduce the perceived lag. Extrapolating this trend points to an emerging category of applications that would shift data and computation directly onto the clients, and thus require distributed protocols such as replication in order to be implemented efficiently. This perspective flies in the face of contemporary industry mantras such as Google's much-touted "cloud computing" [201], but we believe it is better supported by the facts than other visions for the future.

First, web content is increasingly dynamic, and bundled with executable code. In some sense, this has been true since the invention of the Web, since unlike the pre-assembled FTP [252] content, HTML [34] web page content was assembled on the user's machine from parts that could reside in different locations. At the time when the initial drafts of the HTTP [110] protocol started circulating in 1995, the Netscape browser already supported JavaScript [112], an interpreted language based on Java [133], which was essentially invented for the Web. A year later, JavaScript morphed into ECMAScript [97] and became a Web standard, and a few years later gave rise to ActionScript [88], a technology used extensively in Flash [314] animations that are present on nearly every web page today. Flash itself was invented back in 1996, simultaneously with the ActiveX [101] standard for embedding executable objects, and it is still widely used for video playback in sites such as YouTube [322], for interactive games and animated commercials. Executable code became so pervasive that ActiveX and other forms scripting have become one of the main security threats (most recently, Google's "caja" [130] attempts to address some of such concerns), and Flash advertisements on some of

the popular websites consume enough CPU cycles to seriously slow down even modern PCs. By now, the tendency to bundle content with executable code has crossed even to what traditionally has been the domain of passive and pre-assembled content: selling copies of movies, recorded on media such as optical discs and magnetic tapes, by recording companies such as Sony or Universal and their distributors, to home users. In the past, such content used to be non-interactive; once mastered and replicated in millions of copies, it only allowed passive playback of the pre-recorded content. The Blue-Ray [325] specifications completed in 2004, however, now include a language BD-J based on JavaScript, and since 2007 all new Blue-Ray players must be able to run it.

Second, web content is increasingly personalized, interactive, and involves frequent communication between the client and the server at which the content originated. The origins of such interactions, frequently referred to as *remote scripting* [17], date back to the introduction of the IFRAME [86] element in 1996. In the following years, Microsoft and Sun introduced Microsoft's Remote Scripting (MSRS) [83] and JavaScript Remote Scripting (JSRS) [16]. In 2002, the term *rich Internet applications* [8, 195] was coined to refer to web sites that mimic the features and functionality of traditional desktop applications. Since then, several technologies such as Adobe Flex [6], Adobe AIR [5], Ajax [17], Google Web Toolkit [131], Microsoft Silverlight [209], and Java FX [211] were created, specifically to enable this sort of functionality. In these, the traditional request-response pattern of communication is replaced by series of asynchronous upcalls and downcalls between the client and the server, and the interactions have become more fine-grained and limited to a small portion of the content displayed by the client.

Third, web content is becoming short-lived, in large part because from the point of view of a typical user, the Internet is becoming less of an information archive, and more of a platform for social interactions and a meeting space for groups of users. This is visible in the growing popularity of portals such as MySpace [239], the purpose of

which appears to be not as much about documenting one's life for archival purposes and broadcasting one's personal information, as it is about meeting others and interacting with them in real time. Indeed, users have repeatedly expressed concern about the prospect of mining their personal records [173]. This trend towards dynamic interaction is even more visible in *massively multiplayer online role-playing games* (MMORPG), such as World of Warcraft (WoW) [49], and the closely-related concept of *virtual realities*, such as Second Life [189].

At this point, we should note that technically speaking, MMORPGs and virtual worlds are not Web applications in the strict sense. However, they do attract millions of active players, and an even larger number of casual ones, and in principle, the general model for interacting with the users is similar as in the case of a regular browser: the user navigates around a large virtual space composed of a vast web of interconnected locations (rooms, islands), watches content stored in those locations (other users, virtual objects, billboards, or screens), and sometimes can also change the world by posting new content (leaving objects behind). We believe that even though currently, one cannot navigate into MMORPGs and virtual realities using a regular web browser, this is bound to change. Users have traditionally liked "media center" applications and all-in-one devices that allow them to access different types of content through a single interface, and there have been many examples of technology convergence in the past; one recent example is video streaming, which is now embedded directly into web sites and does not require the use of an external player any more. Likewise, we expect that MMORGPs, social networks, and the Web will eventually converge into a single virtual space of dynamic, interactive content, and Web browsers of the future will serve as portals into this space (indeed, experiments with this idea have already begun [85]). We expect that the distinction between today's Web browsers and game clients will vanish; the future Web browsers will simultaneously play both roles, serve both types of content combined

into mashups, and allow the user to navigate between different types of content seamlessly, without the need to explicitly switch between a browser, a MMORPG client, and a virtual world client. For the reasons just articulated, in the remainder of this section we'll build our argument based on the assumption that MMORPGs are going to shape the future of the Web, and we'll use the term "browser" in the more general sense of an application that displays the content of the future Web to the user, in a traditional 2-dimensional or 3-dimensional form, and allows the user to navigate around it.

Early systems were crude because of limited technology, but the idea goes back to the *multi-user dungeons* (MUD) in late 1970s, which are essentially text-based precursors of MMORPGs. The first MMORPGs, with a crude graphical interface, started to appear as early as 1991, roughly contemporaneously with the first Internet browsers. Commercial MMORPG platforms, such as Active Worlds [3], introduced the concept of a 3D web page as early as in 1995, soon to be followed by open source initiatives such as WorldForge [315] in 1998, and eventually Blizzard's World of Warcraft in 2001. By the time of writing this thesis, the number of actively playing WoW users has grown to over 7 million. This is about as many as the number of customers of NetFlix [227], the today's largest movie-rental company, and over twice as many as there are user channels on YouTube [322], the most popular video-publishing web site.

Some of the MUDs and MMORPGs allowed users to modify the virtual world they are in. Most recently, SecondLife implemented this capability in a particularly atractive way, and since its creation, it has gained much publicity in the media.

We believe that MMORPGs will continue to gain popularity, and that within a few years, virtual realities and other variants of MMORPG-style entertainment will become a dominant form of web content. This trend has profound implications. Note that unlike a typical, archival content, created offline, published to persist and to be indexed, a virtual reality is dynamic, and most of the traffic does not need to be indexed or archived.

Thus, for example, words spoken by a user in virtual reality, actions the user has taken, music or videos the user is currently streaming to other users, need to be efficiently distributed, but not necessarily stored, and as the user engages in new activities, the history of the user's interactions ceases to be important. While for some users, the concept of recording their interactive experience might be attractive, doing so for all users would represent unnecessary overhead, and would require tremendous resources.

From this perspective, considering that certain types of data do not need to be stored centrally for archival purposes, one might question the need for relaying all data through the server. Indeed, in terms of content dissemination, one might compare the central server in a data center to a proxy that relays traffic, and represents a bottleneck. For the type of dynamic content that requires fine-grained, two-way asynchronous interactions with the place where it is stored, and often does not require indexing or archiving, one should consider at least partially delegating responsibility for certain tasks, in particular maintaining state and relaying updates, to the clients. This is not to say that servers should be eliminated: they play several other important roles besides just relaying updates and storing state, e.g., they provide security, and serve a number of other administrative tasks, such as organizing clients into groups, etc. Also, for certain types of content, such as the contents of a virtual room, one would need a level of persistence that may be hard to implement in a purely peer to peer fashion. Despite this, there is a merit in getting clients involved; the potential payoff is a greatly improved scalability, achieved by means of harnessing the computation power of the users' machines.

Traditional push-based approaches to scalability such as web caching [126] are not of much help in the scenarios we outlined because the content is so dynamic. In effect, the load on the servers rises in proportion to the number of clients. To deal with this load, most MMORPGs partition the world across servers, and limit the number of users that can access a particular world to a few thousand. SecondLife takes a different approach,

and divides the virtual land into 256m x 256m square pieces, each of which is controlled by a separate server. Maintaining such infrastructure requires substantial resources: in 2006, SecondLife was reportedly [289] using over 2579 dual-core Opteron servers, and at peak usage each server was handling 3 users at a time, out of a population of about 230,000 active players. Although according to the same report, other virtual world implementations hosted up to 100 users per server, the numbers cited are still orders of magnitude below what one would need if this style of interaction were to become as massively popular as more "traditional" content providers, such as MySpace with its 110 million active users, and images and posts counted in billions [239]. Moreover, Second Life currently still offers a rather primitive experience.

From the point of view of the end user, maintaining this expensive pool of servers represents inefficiency. This inefficiency is expensive: most MMORPGs, such as WoW, require monthly subscriptions at a price comparable to the cost of high-bandwidth Internet connection. SecondLife is reportedly selling virtual land, and thus essentially renting server space, for prices comparable to top-end web hosting offerings. Google is believed to have acquired YouTube with the intention to extend its advertising to commercials in video clips, and has recently submitted a patent for this purpose [221]. Yet it is not at all clear that users will tolerate pervasive advertising in contexts such as games, social networks and private experiences. One may question both the scalability and the economic rationale behind the use of cloud computing as the model for the future Web.

We believe that in the longer term, decentralization of virtual worlds is inevitable because most content will never be sufficiently popular to justify maintaining dedicated infrastructure, and could be handled in a peer-to-peer fashion between smaller groups of users. Most studies on web traffic point to a heavy-tailed Zipf-like distribution [54, 120, 241], and studies of peer-to-peer sharing applications such as Kazaa [265] suggest that in case of the file sharing content, the distribution is even more flat and heavy-tailed. This

is consistent with the fact that the Web is becoming, in some sense, more "democratic", and content created by users dominates. At the time of writing, the number of homebrew video clips on YouTube has grown to nearly 80 million videos, about 1000 times more than the total number of movies available at NetFlix, of which only about 5% are available for viewing online. We believe that when future virtual realities take off on a massive scale, the heavy-tailed trend will continue, and any given piece of virtual content would be accessed by perhaps 10–100 users at a time, on average, and at a rate that would make it relatively easy to support it using a peer to peer approach outlined in the following section, but that multiplied across millions of objects and users, the aggregate load would be impossible to host in a centralized fashion in a data center.

The above analysis suggests that the Internet browsers of the future will render content that is hosted in a distributed fashion, and since as we have observed, today's content is bundled with code, creating such content will involve creating instances of distributed protocols. Our ultimate goal is to make this as painless as designing today's web pages. Moreover, as we will see, this step creates new opportunities to automate the introduction of reliability, security, and privacy-preserving technologies.

1.3 The Emerging Paradigm

Our vision of the future Web suggests that today's server-based web content will be replaced by content that resides on client machines, and that is maintained by the group of its clients in a collaborative, peer-to-peer fashion (Figure 1.1). We will henceforth refer to such content as a *live distributed object*, or simply a *live object*. Here, *live* reflects the fact that each user owns a dynamic "view" of the object that may be synchronized in real time with the view presented to other users, and may be interactive, permitting any user to modify it, whereas *distributed* reflects the fact that this is achieved in a decentralized manner, by running a distributed multiparty protocol. Note how this naming convention

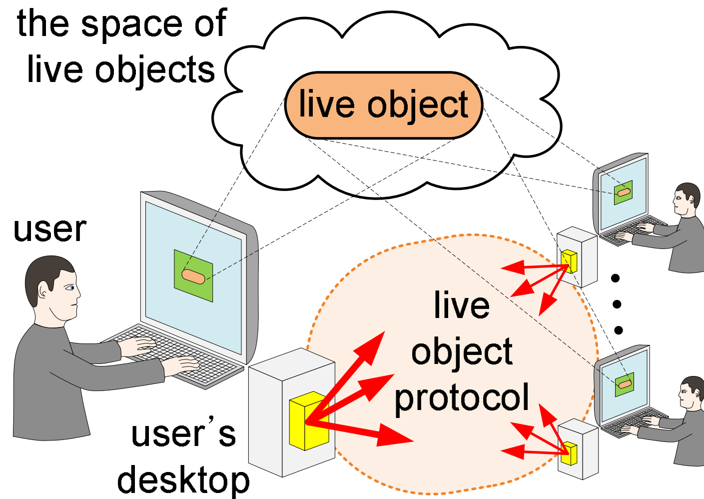

Figure 1.1: A live distributed object accessed by multiple users. The state of the object, if any, is replicated among the users accessing the object, and is updated in a peer-to-peer fashion through a distributed multiparty protocol, by sending updates and control messages directly between clients, without a central authority acting as an intermediary.

conforms with the earlier definition of a live object as an instance of a distributed protocol. A replicated data structure maintained in a collaborative fashion by the group of its clients is certainly an example of a live object in the sense defined earlier: it exists on multiple nodes, it may have distributed (replicated) state, and the software components implementing it communicate with each other by passing messages over the network.

The reader may have noticed that while the focus of the dissertation is a general programming environment for live objects defined as instances of any type of distributed multiparty protocols, our vision of the future Web at a glance may seem to only motivate one particular type of live objects that have replicated state and graphical interface. It is not immediately clear why one would need support for features such as type-safe composition. In the remainder of this section, we explain the rationale for the more general problem statement and argue that approaches focused exclusively on the presentation aspects of the development, or that force programmers to work within a narrow model and a single class of protocols, can facilitate only a small part of the overall programming effort involved in building the future Web, and thus in order to make a difference,

we need an object-oriented programming environment that facilitates building any type of distributed components, not only the front-end, and that has the ability to leverage any type of distributed protocols as reusable building blocks. This motivates the earlier broader definition of a live object as an instance of any distributed protocol, and the need for a general-purpose development platform that can support this new class of objects.

Indeed, at the first glance, live objects of the sort just described could be realized simply as state machine replication protocols that internally leverage some form of reliable multicast. For example, if an object represents a document, opening it may correspond to joining the underlying multicast protocol and receiving a copy of the replicated state from another user. This state may include the document body, metadata, or locks on sections of it acquired by other users. Individual edits may be announced by reliable multicast to all clients using the document. Similarly, if the object represents a room in a virtual world, walking into or out of the room may correspond to joining or leaving the protocol. The replicated state may include the interior design, a list of users or objects, their positions, etc. Users may announce their actions via multicast when they speak, walk, pick up or drop objects, etc.

Given this, one might optimistically assume that to enable the future Web, it is sufficient to deploy an Internet-wide reliable multicast substrate using some scalable, high-performance protocol, wrap the protocol into some easy to understand language construct, such as a shared variable, that abstracts all of its intricacies away and hides them from the developer, and let the developers use this mechanism pervasively, on a massive scale, simply by adding application-level logic that handles user interface events, etc. Indeed, this is exactly the approach of the Croquet [272, 273] project, which paints a similar vision of the Web composed of objects driven by replication protocols, and even has a metaverse browser Cobalt [85], which looks similar to the prototype of our platform. Croquet uses a fixed communication model, based on the two-phase commit

protocol, and with peculiar real-time synchronization properties. All objects in Croquet are derived from the base class *TObject* and inherit its default replication behavior, augmenting it with application logic. The system is implemented in Smalltalk [127].

One one hand, the work done by the Croquet team is impressive. It offers an intuitive interface, and extremely useful capabilities such as debugging the system from within itself (a mechanism that was first explored in an earlier work on Kansas [275], a runtime environment for the programming language Self [293, 274], also inspired by Smalltalk).

On the other hand, the platform is limited to a set of built-in communication and replication mechanisms, and provides little flexibility in tweaking the underlying infrastructure. Thus, for example, it would be hard to change consistency model of a critical object to Byzantine state machine replication, protect it using a secure key management protocol, or leverage a time-critical multicast protocol for objects that might be a part of a battlefield visualization. Thus, while the environment might be well suited for one class of applications that resembles a simplistic version of Second Life, its mechanisms might be completely inadequate for uses in real systems, where the performance, reliability, security, or fault-tolerance demands might vary, and where an entirely different replication scheme and communication substrate should be used to meet these demands.

The same problem is visible in the closely related work on real-time collaborative editing. Systems of this sort started appearing as early as in 1986. Examples include CES [137], DistEdit [175], MULE [249], GROVE [99], IRIS [50], Duplex [240], Jupiter [228], adOPTed [257], REDUCE [282], and more recently, Microsoft's Groove 2007 collaboration suite [208]. Several approaches to preserving consistency among replicas were surveyed in [281, 280], and [75]. What is most striking after reviewing this work is the diversity of approaches applied to this seemingly trivial task. Some of these approaches use voting, some use transactions or group communication, many restrict synchronization to a small part of the state, use various forms of locking, and tolerate

inconsistent ordering of updates using various application-level techniques. Clearly, no single "one size fits all" solution has a chance of massive, Internet-scale adoption: different users and applications have different needs, and protocols that cater to those needs will contain complex application-specific logic, and application-specific optimizations.

Even though most live objects could be implemented using protocols such as state machine replication [267], consensus [183], virtual synchrony [44], or other techniques based on totally ordered reliable multicast [312], users are unlikely to accept the uniformly high cost imposed by these solutions. While using these techniques would be adequate for objects that are part of a trading system or a banking application, they would be a bad choice of a transport for applications such as a distributed version of WoW or SecondLife. There, simple and efficient FIFO ordering would usually suffice or could be augmented with a simple locking scheme. Many types of updates are idempotent or made obsolete by subsequent ones. The state of the replica owned by a user who crashed or otherwise ceased to use the object is irrelevant and can be ignored, etc.

A single graphical live object could also internally rely on a variety of other protocols (Figure 1.2). For example, replication involves reliable multicasting, which involves unreliable multicast. The latter may be implemented differently depending on where the object runs: it should leverage IP multicast [91] whenever available if used in an enterprise LAN, but switch to a peer-to-peer overlay across a geographically distributed set of nodes. Reliable multicast also often relies on a protocol that provides the group of object replicas with consistent view of membership [79, 41] or failure detection [73]. Often, the protocol relies on discovery and naming services. Membership services may internally use a protocol such as Paxos [183, 71]. Certain stateful objects might need a backup protocol to persist their state when no users are accessing them.

Overall, the infrastructure "behind" a single live object with a graphical user interface running in a browser might involve protocols as diverse as gossip, multicast with

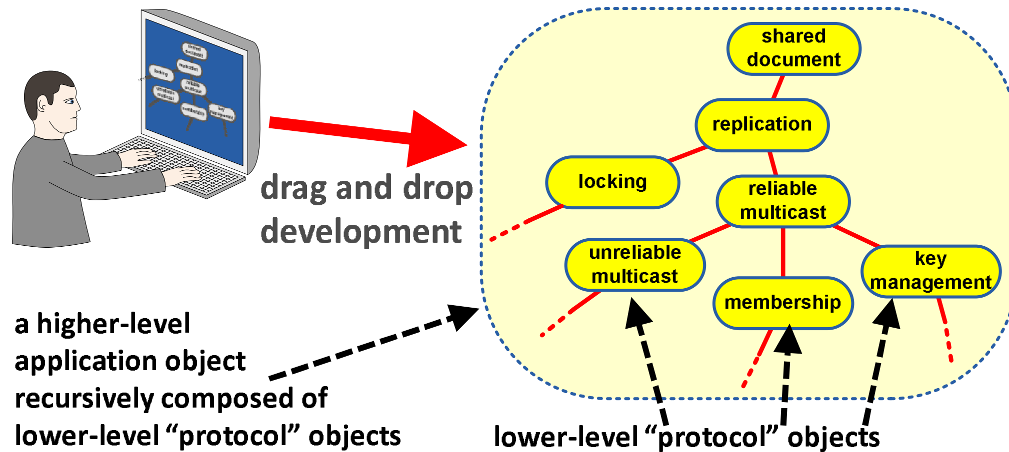

Figure 1.2: Live objects can represent low-level or high-level components. An application object with a user interface and a replicated state can be recursively decomposed into protocols, down to the hardware level, and each distributed protocol instance in such decomposition is itself a live distributed object that "runs" across a set of machines.

different types of consistency, ordering or timing, Byzantine replication, distributed hash tables, credential management, naming, locking, and content distribution protocols, failure detectors, etc., and these might vary depending on the network topology, number of users, the expected pattern or volume of workload, etc.

In the light of this discussion, if we want to enable the future Web, we cannot focus on just one type of protocols, and we cannot focus merely on the front-end; that is why we defined the concept of a *live distributed object* to mean not only a front-end component in a browser that a user may interact with, but more generally, an instance of any type of distributed multiparty protocol that might form a part of the infrastructure supporting such interactive interface. In this unified perspective, the "infrastructure" protocols mentioned earlier, such as multicast, membership, naming, discovery, gossip, etc., are all live distributed objects. Some of these may be running on client machines, some within clusters in data centers, and some might be implemented in the networking hardware. In our new perspective it is "live distributed objects all the way down". Our goal is to make it easier for developers to build such objects and to compose them into complex applications such as SecondLife.

As we shall argue in Section 1.4.1, prior attempts to marry distributed protocols and the component-oriented paradigm have not been entirely successful in the sense that in the attempt to provide an easy to use, uniform, fool-proof technology that fits all uses, they neglected to respect the diversity of the underlying protocols involved in building such systems. In consequence, they tended to impose rigid solutions with properties that for most applications are either too weak, or too strong and unnecessarily costly.

The above is nicely illustrated in the case of fault-tolerant CORBA [198]. The standard provides an object-oriented embedding of a state machine replication protocol that completely hides the distributed nature of the underlying implementation from the programmer by encapsulating groups of services and presenting them as a single, fault-tolerant remote object. The resulting model is extremely transparent, but it is based on a very rigid form of state-machine replication that can be used only with deterministic, event-driven components, and precludes the use of techniques such as multithreading, thus enforcing a style of programming that many developers would find unnatural. Moreover, the embedding is useful only for a handful of protocols with strong semantics, and is a bad fit for most others. For example, the CORBA object abstraction would not be a clean interface to atomic commit, leader election, or gossip protocols.

We believe that while developers may indeed demand uniformity, they also chafe at rigid constraints that may not fit their needs. The conclusion from these past experiences is that in building the language and infrastructure support for a future Web, the key design principle should be a respect for the diversity of distributed protocols and the heterogeneity of the Internet, achieved through an architecture that treats distributed protocols as reusable building blocks.

In this work, we explore two key aspects of this component-oriented perspective: *functional compositionality* and *structural compositionality*.

The former type of compositionality, illustrated on Figure 1.2, allows instances of

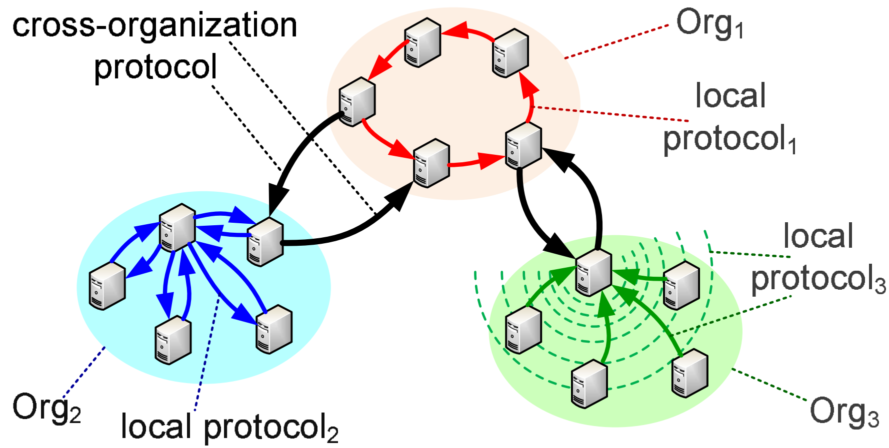

Figure 1.3: Multiple local "sub-protocols" combined into an Internet-wide structure. The local sub-protocols implement the desired functionality Ψ locally within the respective organizations, using different techniques and leveraging different types of hardware support, e.g., by locally organizing nodes into a token ring, a star topology, or by using IP multicast. A cross-organization protocol coordinates the work performed by the local protocols in such a way as to achieve the desired functionality Ψ across the Internet.

different protocols to be stacked together in a type-safe manner to build complex structures, in which the individual protocol instances act as functional components that implement different internal tasks, such as multicast, membership, state transfer, discovery, etc. In Section 2, we propose a new programming model designed for this purpose.

The latter type of compositionality, illustrated on Figure 1.3, allows a *single* protocol instance to be constructed in a modular way: a protocol that runs across the Internet, and spans multiple administrative domains, could be constructed from local "sub-protocols" that run within certain regions of the Internet, combined with a higher-level "glue" protocol that connects the "sub-protocols" into a single structure.

At first, it might not be obvious why this type of compositionality is necessary, and indeed, we are not aware of any prior work that would enable this type of modular protocol construction. For example, as explained in Section 1.4.3, earlier work on multicast protocols has focused almost exclusively on systems that impose uniform mechanisms across the entire network. However, recently several researchers have pointed out that the Internet is heterogeneous, and that techniques that fail to recognize this aspect tend

to perform poorly [36, 48].

Recently, several techniques have been proposed that address certain types of heterogeneity: for example, there has been work on *capacity-aware*, *topology-aware* and *locality-aware* overlays that attempt to factor in aspects such as node capacity or network latency when deciding which nodes should establish peering relationships, or how to route data and control messages. We believe that the work on **-aware* systems is important and useful, but also that there is a limit to how much one can achieve through a single, uniform solution that leverages the same protocol across the entire Internet. In our view, the path to achieving performance and scalability in a heterogeneous system leads not through a single monolithic scheme, no matter how complex and sophisticated, but through a composable, modular architecture that has the ability to leverage different mechanisms in different parts of the network.

For example, while the use of application-level multicast is adequate in WAN settings, where IP multicast is disabled, most enterprise LANs and datacenters do support IP multicast, and a modular system that has the ability to use such hardware mechanisms locally, wherever available, has a much greater potential than techniques strictly limited to application-level multicast. Similarly, while most WAN links have low throughputs, there do exist mechanisms such as National LambdaRail [225] that can move data across large distances at throughputs exceeding 10Gbps, and again, a modular system that can make use of such hardware where available has the potential to outperform techniques optimized for the "average" case. Finally, aside from all the physical aspects of heterogeneity, there is also an important *human* factor: the Internet is collectively owned by a large number of organizations, each of which is governed by a separate authority, with its own local management policies. Thus, while one ISP might disable IP multicast and enable peer-to-peer traffic, another ISP, given the same constraints, may do the opposite.

In Chapter 3, we propose a modular architecture, inspired by object-oriented design

principles, that enables hierarchical composition of protocols. The discussion focuses on reliable multicast, but the approach appears to be quite general, and we are currently working on extending it into a general-purpose platform that could support structural composition of protocols as diverse as multicast, virtual synchrony, or transactions. We briefly discuss elements of this platform in Chapter 5.

By now, the reader might wonder about the precise nature of the relationship between live objects and multicast, and the treatment of multicast in this dissertation. Previously, we have noted that many types of objects, in particular the types of objects with graphical user interface discussed at the very beginning of this section, can be implemented using reliable multicast; yet the concept of a live object is broader and includes many other types of protocols, as well as larger entities composed of multiple protocol instances.

Reliable multicast protocols play an especially important role in our vision, in that they are very useful as components of most distributed applications we have considered so far, and many protocols that are not classified as multicast protocols can in fact be viewed as variants of reliable multicast. For example, multicast could be used to reliably and consistently delegate tasks in a scalable role delegation system, keys or revocations in a security protocol, or lock requests in a locking protocol. At the same time, protocols such as consensus, transactions, or atomic commitment involve dissemination of data to agree upon, confirm or reject, and can be thought of as examples of reliable multicast with a particularly complex logic dictating what happens to the disseminated messages.

The importance of multicast as a building block and the similarity of certain types of commonly used protocols to multicast suggests that the feasibility of the live objects vision depends, to a large degree, on whether the model and platform can accommodate multicast objects cleanly, and whether it is possible to efficiently support the patterns of use that would emerge if our vision took off on a large scale. The key issue is related to

scalability, especially scalability in dimensions that have not been extensively explored in prior work, such as scalability in the number of multicast objects running simultaneously on overlapping sets of nodes. Addressing the scalability challenge for multicast objects, and exploring the practical architectural and performance questions revolving around the use of multicast protocols within our live objects framework is thus an important first step towards making our model useful. Indeed, we believe that techniques described in Chapter 3 and Chapter 4 in the context of multicast can be generalized; some of our ongoing work in this direction has been briefly discussed in Chapter 5.

1.4 Prior Work

In our treatment of prior work, we adopt a historical perspective: we divide the work into three functional areas, and within each, we identify the subsequent waves of research. Our account is not meant to be exhaustive: our main goal is to demonstrate how research in each of these domains has drifted into a direction that resulted in most sophisticated systems, but failed to address the fundamental needs of the developers, and how in the end, the most widely adopted solutions were often the least sophisticated ones.

1.4.1 Embedding

In this section, we focus on the problem of embedding distributed multiparty protocols into a modern object-oriented language and development environment.

The history begins around 1967 with Simula [89], the first object-oriented programming language. Together with the packet switching paradigm [26], Simula led to the development of the *Actor* model [148] of computation in 1973, the concept that every software artifact is an entity that communicates concurrently and asynchronously with its environment by exchanging messages. Smalltalk [127] took this concept to the ex-

treme, and proposed that every part of the computer program, including numbers and operators, classes, and stack frames, be an object. Our vision, in which every protocol is a live object within a larger system, and the message-passing interface we have chosen, are directly inspired by Smalltalk.

Experiments with distribution began early in the 1980s, and many *remote object* systems emerged, including Chorus [25], Argus [191, 190], Amoeba [223], Eden [10], TABS [277], Clouds [90], Emerald [167], Camelot [278], and Network Objects [47]. The key capabilities researchers focused on were location transparency, mobility, and support for nested transactions across groups of remote method invocations. More recently, modern remote object implementations such as *remote method invocation* (RMI) in Java, *remoting* in .NET, or *web services* and SOA, have incorporated some of these mechanisms, in particular the concept of making local and remote objects accessible through the same interface. Jini [303], built upon Java and RMI, has taken this idea further and added elegant and clean support for code mobility.

Strictly speaking, remote objects were not *distributed* in the sense of the word used in this dissertation: these objects resided in specific locations, and did not involve multiparty protocols. Another major line of research pursued in this decade, and perhaps somewhat closer in spirit to our model, was *distributed shared memory* (DSM). Ivy [188] was the first DSM system, and Linda [64], with its *tuple space*, was the first example of a "language embedding", essentially a crude form of content-based publish-subscribe. Linda was revived again a decade ago with JavaSpaces [114], as a part of the Jini initiative. Researchers loved Linda for its elegance and simplicity, but models based on DSM were problematic, in that they pretended that a distributed system is not really distributed, and were intrinsically fragile in settings where faults would occur [288]. The desire to provide transparency in terms of distribution, by hiding it from the programmer, resulted in a mechanism that is inherently non-scalable: implementing the

shared memory abstraction in a decentralized manner involves costly synchronization.

Most other models invented throughout the 1980s (surveys and bibliography can be found in [230, 77, 65, 102, 56]) shared the general characteristics of remote objects and DSM. A notable exception, and perhaps the first attempt to embed a truly "distributed" entity into the programming language was the work on *fault-tolerant objects*, which dates back to the Isis system [46] in 1985. However, the model used in Isis was limited to replication, did not have the concept of an *object type*, and while their objects could be replicated, the replicas were assumed to reside on clusters of servers, and were contacted by the clients remotely, in the traditional client-server fashion. Another work that followed the design philosophy advocated in this dissertation was the *fragmented objects* model [200], which proposed objects that have replicas in multiple locations, and internally use *connective objects* that encapsulate a multicast protocol and expose multiple communication endpoints. However, the model had many of the same limitations as fault-tolerant objects in Isis. Later, the Globe [297] system slightly relaxed the model, by supporting composition of the sort shown on Figure 1.2. Our work on live distributed objects is inspired by and extends the Isis, fragmented objects, and Globe approach by addressing several limitations of these early systems.

Meanwhile, throughout the late 1980s and 1990s, reliable multicast systems similar to Isis flourished. The *process group* [40, 76, 44] and *message bus* [236] mechanisms were popularized as general-purpose programming abstractions. Distributed ML [180] pioneered the idea of embedding these abstractions as first-class programming language constructs. The idea was to choose a single underlying communication paradigm, and then present it through a typed interface. In the following years, many middleware platforms incorporated various forms of message buses that followed the same approach.

The typed channel abstraction provided by these systems was useful, but the embedding was very superficial. The new language entity that represented a communica-

tion port, a multicast group, or a channel had the look and feel of an object, but did not support composition. Thus, for example, different types of reliable channels were not compositions of unreliable channels and higher-level protocol layers implementing some flavor of reliability. If the abstraction supported different types of reliable channels, the desired flavor of reliability was simply passed as a parameter, chosen from a fixed list of reliability modes. Also, the new distributed entity was limited to the concept of a communication channel, but not higher-level constructs. Thus, for example, it could not represent a shared document, or a composition of a reliable multicast protocol that depends on a membership service with a membership protocol. Accordingly, the notion of a distributed type was also shallow, limited to the type of the transmitted data, and perhaps parameterized with the type of reliability supported by the channel. In essence, from the point of view of a programmer, using a communication library through these typed channels did not differ much from accessing it through an ordinary API, and the embedding did not make writing distributed applications dramatically easier.

In the attempt to address this problem, many researchers tried to develop more transparent language embeddings that would completely hide the distributed aspects of the protocol from the developer; an example of this has been the fault-tolerant variant of CORBA [198] mentioned earlier. Our earlier argument against fault-tolerant CORBA's approach to distributed protocol embedding by transparency and hiding applies also to other, similar language-centric approaches, and illustrates what we believe is a general principle. Developers use entirely different methods to access a system depending on whether it is hosted on a single remote server [40], cloned for load-balancing on a cluster [198], or using state machine replication [267], and those are only a few of the many distributed computing paradigms. Each of these matches a distinct style of applications [229], and each has its own constraints and interfaces. This underlying diversity of styles runs contrary to transparency and uniformity, which perhaps explains why distributed

computing has remained a world of tools one either re-implements from scratch or accesses at a code level through esoteric interfaces. Hence, a successful marriage of the object-oriented and distributed computing paradigms cannot be achieved by hiding. To the contrary, the distributed nature of a protocol and its peculiar properties should be explicit, and reflected by the interface and type of the language entity representing it.

The recent trends seem to agree with this observation. Although the emerging family of web services specifications include standards for distributed paradigms such as publish-subscribe notification [134, 52], distributed transactions and other sorts of coordination [61, 87, 62], none of these mechanisms has been integrated as a programming language entity into either .NET or Java/J2EE. Writing applications that use standards such as WS-Notification [134] or WS-Eventing [52] thus generally requires either manually dealing with SOAP [138] messages, or using third-party tools such as WSRF.NET [159]. The lack of broad adoption certainly does not prove yet that these standards are not useful, but through our own experience and a dialogue with other users, we did discover these standards to be too inflexible to be a good match for many types of practical uses.

We conclude this section by comparing live objects with Jini [303], a recent initiative that, to a degree, shares the overall vision and some of the specifics of our proposal. Jini [303] introduced the concept of a single uniform Java-based Internet-wide "space" of services. Jini services could be local or remote, and could be implemented in hardware, yet to the clients, their location and physical access protocol are opaque. Jini services "live" within a global Internet-wide space of Java objects, and can be dynamically discovered and transparently accessed. Jini achieves this by dynamically retrieving the code that implements a service and encapsulates the client-server access protocol, and dynamically loading this code into the client process.

Our approach is inspired by Jini, in that we also perceive the Web as a global "space"

of protocols that "live" in it, and much like Jini, the code implementing these protocols can be dynamically loaded into the client process, in a transparent manner, and potentially with assistance of "infrastructure" objects, such as discovery or naming protocols.

However, our vision is driven by a fundamentally different philosophy. Jini is, at its core, a service-oriented technology, and is rooted in the client-server paradigm, similarly to the modern web services standards and SOA. In Jini and SOA, a service is an abstract entity with an ordinary procedure-call API, embedded in the programming language of the client, and the underlying protocols are simply a means of accessing it. In contrast, our vision focuses primarily on the protocols themselves: we attempt to *accommodate any protocol type as a usable building block*. Jini enthusiasts have proclaimed "the end of protocols" [304]. In our work, it is the protocols that are the very center of attention.

At a first glance, the difference may seem superficial. Jini's procedure-call API may consist entirely of asynchronous calls, and the dynamically loaded code for accessing a Jini "service" may, in fact, run a multiparty distributed protocol such as reliable multicast. Hence, Jini could, in theory, be used in a purely peer-to-peer manner. Nevertheless, it is evident throughout the standard that the platform is driven by a client-server design philosophy, and this has profound implications. Thus, while technically the peer-to-peer style of use is possible in Jini, the platform lacks explicit support for this scenario.

First, the standard lacks [220] a rigorous notion of a "group", and any notions dependent on it, such as consistency across multiple members of a group, or replication within a group. For example, Jini's lookup, discovery, and join specifications [286, 284, 285] lack notions such as a *consistent view of membership* or *synchronized state transfer*, and there is no discussion of problems such as consistent failure detection, which are fundamental in building reliable systems. Thus, although services in Jini can be grouped, and clients can discover multiple service instances, in practice the expressiveness of this mechanism is extremely limited and the model is not strong enough to associate a true

semantics with these groups, such as replication, fault-tolerance, etc. In particular, Jini's infrastructure does not appear to be capable of supporting the kinds of replicated objects described in Section 1.3.

More broadly, the notion of auto-discovery, a fundamental concept in Jini, is inherently rooted in the client-server paradigm, and makes no sense in the context of peer-to-peer replication. A client accessing a live object, such as a shared document or a virtual room, must bind to a specific protocol, and the naming and membership mechanism through which the clients discover each other must be deterministic and produce results that are consistent across the entire group. Auto-discovery introduces unpredictability that runs contrary to consistency. In our approach, we reject auto-discovery: protocols for accessing lookup services are themselves live objects, and are recursively embedded in the descriptions that clients use to instantiate object replicas. Thus, all actions that need to be performed by clients accessing a given object are consistent across the group.

Second, Jini's infrastructure mechanisms are inflexible. In many respects, these standards resemble the WS-* family of web service specifications, and the two approaches share many weaknesses. In particular, Jini's version of the publish-subscribe standard [255], with mechanisms such as *store-and-forward agents* and *notification filters*, resembles WS-Notification [134] and WS-Eventing [52]. In both Jini and SOA, the multicast infrastructure is a notification tree, does not support peer-to-peer recovery, and cannot provide strong end-to-end reliability guarantees needed to use multicast for replication, etc. In the same spirit, Jini's support for atomic transactions [287], much like the corresponding web services specifications, is limited to the two-phase commit protocol. Since its invention three decades ago, this protocol has been found to have serious limitations [31], and it has been superseded with a number of more resilient and better performing alternatives [269, 170], but as of this writing, the Jini standard does not appear to permit replacing this default protocol with a different one. The core of the problem is that,

unlike in our approach, Jini's eventing and transaction mechanisms are not "objects" in any real sense, and thus cannot benefit from the core object-oriented mechanisms, such as encapsulation, reusability, decoupling, and type-safe composition.

1.4.2 Composition

In the preceding section, we concluded that prior work on marrying object-oriented and distributed computing has been focused on the wrong set of issues. The main motivating factor has been the uniformity, transparency, and ease of use, but the desire for elegant, transparent language embedding is in conflict with the diversity of distributed protocols, and as we argued, we believe that the developers tend to reject rigid "one size fits all" solutions that may not precisely fit their specific needs. In this section, we focus on support for what we believe is the true essence of object-oriented programming: type-safe composition, encapsulation, extensibility, and reusability. Each of these concepts directly or indirectly depends on the notion of a *protocol type*, hence protocol "types", the ways of expressing them and the relations between them, are central to our discussion.

In most object-oriented programming languages, the hierarchy of types reflects two concepts: *structural inheritance* and *functional subtyping*. As noted in [276], the two concepts are independent: *inheritance* is focused on inheriting the *mechanism*, it saves the programming effort by allowing for code reuse, incremental development, and specialization. On the other hand, *subtyping*, as defined in [276, 12], is focused on inheriting the *behavior* of an object, hence it is focused on replacing the existing mechanism by another, equivalent mechanism, rather than reusing existing code. Behavior of a derived class, defined as a set of event traces or histories, may not be a subset of the behavior of the base class. Likewise, two classes that do not share code, and are not derived from one another, may behave identically. Thus, *inheritance* and *subtyping* are orthogonal.

Past approaches to associating "types" with distributed protocols fall roughly into

two categories, depending on which of these two approaches they opted for, and the systems that fall into these two categories have, in some sense, complementary sets of features. In what follows, we shall shed light on their relative strengths and weaknesses.

This part of the history begins in 1962 with Petri Net [250], a mathematical representation of a distributed system, in which computers communicate via messages. Petri Net was the first attempt to formally model the behavior of distributed protocols, and ironically, nearly five decades later, it might just be the most popular formalism for modeling the interaction between web services and web service composition [212]. Over the following three decades, several other such formalisms, sometimes referred to as *process calculi*, emerged: the Actor model [148], *trace theory* [203], Communicating Sequential Processes (CSP) [153], Calculus of Communicating Systems (CCS) [214], Algebra of Communicating Processes (ACP) [32], *history monoid* [268], π-calculus [216, 215], and Activities [179]. Based on those formalisms or their variants, several languages were created, such as Occam [264], LOTOS [55], and Wright [9] based on CSP, SAL [7] based on the Actor model, or dataflow languages Lucid [302] and Lustre [67].

This early work on process calculi is important, in that it has provided the foundations for formally modeling and reasoning about protocol behaviors, which, as we have argued previously, can be used as a basis for defining protocol *types*. In our work, we have also adopted this perspective: in Section 2.1.4, we define live object types in terms of the patterns of events that can flow between the protocol replicas and their environment.

However, the early process calculi focused on interactions between named sets of participants. As such, they were sufficient for expressing the behavior of a *business process*, but lacked the notion of a group and the expressive power to describe strong semantics, such as replication, consensus, atomic transactions, etc. Most recently, the web services family of specifications and related formalisms, such as WSFL [187], XLANG,

BPEL [161], JXTA [128], Web Services Conversation Language (WSCL) [24], SSDL [245], OWL-S [202], have been proposed for describing interactions between groups of services. Unfortunately, although these formalisms can capture logic such as conditional branching, preconditions, or concurrency, they all lack expressive power to support replication, much like the models they were based on (usually CSP or π-calculus).

Further work on process calculi, such as PEPA [96], ambient calculus [63], or fusion calculus [246], has focused on issues such as uncertainty or mobility, a recent survey can be found in [2]. As recently noted in [118], still none of the current process calculi can express the replication semantics, and they offer weak support for composition.

The second wave of research, starting with I/O automata [196] in 1987, and continuing into early 1990s with BAN logic [57], GNY logic [129], TLA [182], and various temporal logics [141], has produced very expressive formalisms that could capture very complex reliability and security properties of protocols. I/O automata and temporal logic have been successfully used to formally specify group membership and group communication [14, 39, 149, 171, 79]. For the first time, a solid foundation has been laid out for formally defining a behavioral type system for distributed protocols, and several researchers have made steps in this direction, e.g., in the work on type signatures for network stacks [38] and layers of the Ensemble group communication system [39, 149].

Probably the first example of use of formal behavioral protocol specifications in the context of protocol composition was the work on strongly-typed protocol stack composition in Horus [169] and Ensemble [294] in the late 1990s. Horus and Ensemble were not object-oriented programming environments, their protocols were not objects, and their specifications were not types in the strict sense. They had a number of limitations, such as support only for strictly vertical protocol stacks, and lacked a clean object-oriented embedding. Nevertheless, this work has preceded and inspired the work described in this dissertation, and although its objectives have been different, it represents an important

technical advance on the path towards realizing the live objects vision.

However, the early work on automatically verifying compositions of protocol layers in Horus and Ensemble has not evolved into a formal behavioral type system and language embedding. At that time, most researchers have focused on using formalisms such as I/O automata in the context of proving the formal properties of protocols rather than on using them as a foundation for behavioral distributed protocol type systems. The Promela [155] specification language, its descendants [224, 29, 30], and the model checker SPIN [156] targeting it, the work on Proof-Carrying Code (PCC) [226], and the use of theorem provers for dynamic protocol layer composition in Ensemble [194], have all focused on the relationship between the protocol code and its formal specification. While the use of theorem provers at the source code level is useful in *enforcing* typed contracts between components, it is completely orthogonal to the problem of *defining* such contracts and *comparing* them, and only the latter is relevant in the context of component integration.

Perhaps a part of the reason why researchers have not more actively pursued this direction at the time has been due to the fact that complete protocol specifications in formalisms such as I/O automata were excessively complex and detailed. While the protocol *type* should be essentially a *theorem* that specifies the protocol's behavior at an abstract level, the *code* can be thought of as an actual *proof* of this theorem at a very mechanical level. When comparing the behavior of two protocols to determine whether one is a subtype of another, such low-level implementation mechanics are irrelevant. Much work has been done to combat these problems, and in particular on improving the modularity of specifications in I/O automata and related formalisms [166, 39, 318, 147, 172, 115], but as stated earlier, the focus was mainly on *proving* protocol properties.

Independently, starting with STREAMS [258] in 1984, and continuing for the next two decades, many *protocol composition* frameworks have been proposed, initially only

for point-to-point communication. RTAG [15] modeled protocols as grammars. Subsequent work based on x-Kernel [160] introduced *protocol graphs* [237], and was the first to introduce principles such as encapsulation, reusability, and composability; our model is strongly influenced by this work. Other work from this period includes Consul [218] and ADAPTIVE [266]. Horus [296] was the first to use protocol composition in the context of distributed protocols. As mentioned earlier, however, Horus *microprotocols* could only be stacked vertically, and they all had identical interfaces.

BAST [121, 122] was the first framework in which distributed protocols could be arbitrarily composed and had strong types. Indeed, live objects and BAST share many similarities, e.g., BAST *strategies* are similar to our *proxies*. However, typing in BAST reflected the manner in which objects are constructed: a protocol "type" was simply the type of the class implementing it. Composition was achieved via inheritance, by creating new protocol libraries that inherit and override existing code. The creators of BAST concluded [122] that inheritance was not a good basis for defining a type system. We strongly agree with this statement. While this language-centric approach may seem natural, it is flawed in the sense that it severely limits extensibility. Where a protocol of type *A* is expected, only a protocol that was physically derived from *A* can be used, even though many independent implementations could have precisely the same behavior, and be functionally perfect replacements for *A*. Inspired by the work on BAST, we created a model is which there is no construct equivalent to inheritance or "importing" a library.

In the following years, many other protocol composition frameworks were proposed, e.g., a membership service in [152], Coyote [37], Cactus [151], and most recently Appia [217, 206]. However, the focus in those system had drifted to areas unrelated to type-safe composition. For example, in Coyote and Appia, all protocol layers had access to the same events, and registered for those they were interested in. This improved performance by avoiding the overhead of forwarding events between the protocol layers,

but it also went against object-oriented design principles, such as encapsulation.

Meanwhile, after major middleware providers incorporated message buses and other sorts of eventing substrates into their systems throughout the 1990s, their attention had turned to the concept of a typed event channel, where the channel type was parameterized by the type of events flowing through it. The idea was first explored in 1993 in Distributed ML [180], which incorporated virtually synchronous groups into ML [132]. Later examples of use of the typed channel concept and formal definitions of distributed types include Dπ [146]. Since around 2000, the typed channel idea was incorporated into several systems via mechanisms through which the programmer could control the type of received events, e.g., in MSMQ [207], COM+ [232], CORBA Event Service [234], TAO Event Service [144], CORBA Notification Service [233], ECO [140], Cambridge Event Architecture (CEA) [18], Java Message Service [283], and Distributed Asynchronous Collections (DAC) [106, 105, 104, 103]. Some of these systems were nothing more than just variants of content-based publish-subscribe, but some, like DAC, and earlier Distributed ML, were actual language embeddings. DAC went further, in that it provided an entire family of typed event channels with various reliability properties. However, much as was the case with BAST, the types of these channels were essentially the types of the protocols implementing them, and in this sense, DAC has not departed from the inheritance-based proposed earlier in BAST.

At the time of this writing, modern business process specifications, such as BPEL, JXTA, WSCL, or SSDL, and the earlier work on protocol layer specifications in Horus [169] and Ensemble [294], are the closest to our overall vision; they specify behaviors, and as standards created by developers, they are designed with properties such as simplicity, interoperability, and composability in mind.

Our work is an attempt to find a common framework, within which we could embed the many different past approaches, and leverage the earlier work on TLA, I/O automata,

temporal logic, and formal specifications such as [14, 39, 149, 171, 79], within a common strongly-typed and object-oriented protocol composition environment.

1.4.3 Replication

In this last section, we focus on replication mechanisms that might support live objects. Our discussion will center around reliable multicast, as a replication technology most relevant in this context. Although, as stated earlier, our overall vision is not limited to reliable multicast protocols, it is clear that whether our approach can be adopted on a massive scale depends to a large degree on whether we can implement reliable multicast in a way that scales in the major dimensions.

Work on replication dates back to the invention of the primary-backup scheme [11] and the invention of the *two-phase commit* protocol [135] in late 1970s. In the following years, researchers focused on devising mechanisms with strong semantics [124, 123, 35, 44, 1, 267, 136]. Most of these mechanisms were much stronger than what the live objects vision requires, yet at the same time, due to their complex and thus costly semantics, they were relatively slow, and their scalability was limited.

Having recognized the scalability challenge, many researchers turned their attention to combating these problems. It quickly became clear that even the most basic forms of reliability, such as ensuring that in a system without node crashes, all recipients receive the same data despite the possible packet loss in the network, are difficult to ensure in a scalable manner. While hardware mechanisms such as IP multicast could unreliably *disseminate* data to hundreds of nodes, reliability required *collecting* feedback from the recipients in a scalable manner, and for the latter, hardware offered no support.

Throughout the 1990s, many reliable multicast protocols have been proposed with the simple semantics mentioned above: RBP [74], MTP [116], MTP-2 [51], RMP [308], XTP [306], TMTP [320], LBRM [154], RAMP [176], RLM [204], SRM [113], RMTP

[248], RMTP-II [219], LMS [244], TRAM [78], SMFTP [213], RMDP [261], Digital Fountains [59], H-RMC [205], and ARM [321], among others. Comprehensive surveys and analysis of these approaches can be found in [251, 243, 185, 235, 142]. Here, we limit ourselves to a brief summary. The earliest implementations, which relied on the exchange of *acknowledgements* (ACK) or *negative acknowledgements* (NAK) between the sender and the recipients, suffered from what has been known as an ACK or NAK *implosion*: with a large group of recipients, the overhead of receiving and processing feedback at the sender was the main factor limiting performance. SRM pioneered the use of IP multicast to limit this overhead, by allowing a single node to report a network loss to the entire network, which prevented other nodes from doing the same. However, the technique was based on the assumption that losses are correlated, whereas in modern networks, losses occur mostly on the recipients, in the operating system. Packets are dropped due to buffer overfill, because the system sometimes becomes so overloaded that the applications are unable to process the incoming data. In such scenarios, losses are uncorrelated, and systems such as SRM exhibit what is sometimes referred to as a *crying baby syndrome*. One node with a flaky network interface can destabilize the entire network by issuing NAK requests for losses that are localized, thus causing unnecessary IP multicast retransmissions. RMTP pioneered a more practical approach, by having the recipients hierarchically aggregate ACK/NAK information, and perform peer-to-peer recovery. The latest of systems cited here focused on augmenting the *reactive* approaches based on ACK/NAK feedback with *proactive approaches* based on *forward error correction* (FEC) [259, 231, 162, 181], but this technique only makes sense on networks with high loss rates, e.g., wireless, or in applications where achieving low latency at the expense of other performance metrics, such as throughput, is important. In most systems, losses are negligible and tend to be bursty, and the use of FEC runs against the general engineering principle of optimizing for the *common case* at the expense of

the *rare* cases. Recently, some of these protocols were evaluated in the context of grid computing, and tree-based protocols such as RMTP, and similar techniques based on token rings, were reported as the most appropriate for this environment [184].

A common characteristic of these early approaches is that they focused on scalability in terms of the number of recipients, but ignored entirely the problem of scaling with the number of simultaneously active multicast protocols. It is not hard to see that if the live objects vision is massively adopted, each node might participate in tens to hundreds of multicast protocols corresponding to the different objects displayed in the user's browser, hence this dimension of scalability is very important. Most of the early systems that leveraged IP multicast were designed for use with a single multicast protocol at a time: some did not support multiple protocols at all, while others ran each protocol instance independently, and had overheads linear in the number of protocols.

Running each protocol instance independently is particularly problematic with protocols such as RMTP, which construct peer-to-peer tree-like or ring-like structures to aggregate ACK/NAK information and perform peer-to-peer recovery. Recall that the main reason for creating such structures is to prevent the sender, or any other node in the system, from having to interact with a large number of other nodes, a situation that would create a bottleneck similar to ACK/NAK implosion. However, if different protocol instances construct their ACK trees independently, and those ACK trees overlap on the same set of recipients, individual nodes may still have large numbers of neighbors: perhaps $\Theta(1)$ in each ACK tree, but $\Theta(m)$ in total with m overlapping trees (Figure 1.4). Furthermore, systems that use IP multicast, and allocate a separate IP multicast address for each protocol instance, also suffer from a problem known as a *state explosion*. One might naïvely think that scalability with IP multicast comes for free, but in reality, each IP multicast group requires a certain amount of resources, in particular each router has to maintain forwarding state proportional to the number of multicast trees passing through

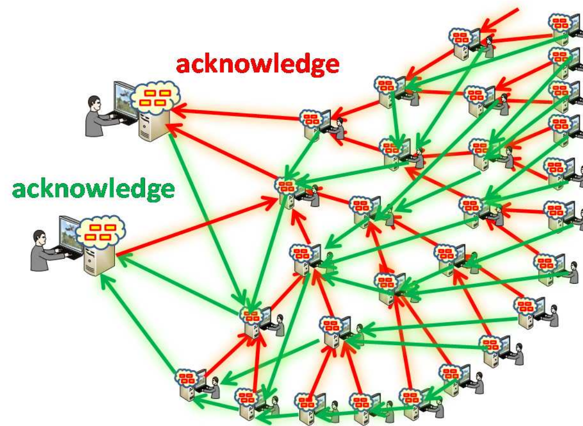

Figure 1.4: With overlapping ACK trees, nodes can have unbounded indegrees. Even if in each individual ACK tree, a node interacts with no more than k other nodes, this upper bound grows linearly with the number of protocols the node is participating in.

it [313]. Several techniques have been proposed to remedy this problem [253, 290, 109], but as of today, the problem still persists, and is recognized as a serious issue even in large data centers. One can verify this experimentally by having one machine A on a switched Ethernet subscribe to a random set X of multicast groups, and flood the network with packets, and another machine B, on the same switched network, subscribe to a random set Y of multicast groups, where $X \cap Y = \emptyset$, and observe that as a result of flooding, CPU usage on machine Y will rise. With 2 nodes and 1 high-performance switch, the author observed a linear growth of about 1% of CPU utilization per 1000 multicast groups, clearly indicating that hardware filtering is failing, and the operating system on machine B is involved. The study in [313] shows that in such scenarios, the router also becomes increasingly overloaded and eventually becomes lossy. In large data centers, with tens of thousands of nodes, the problem is potentially even more serious.

In this work, we propose a way to alleviate the problem by sharing workload across multicast protocols in scenarios where there is a correlation between the sets of nodes involved in running different protocols. In particular, as we shall argue, this is the case for some of the scenarios targeted by live objects. The overall concept is somewhat similar

to *shared ACK trees* [186], but it is used in an entirely different setting. More recently, a few multicast channeling and scalable overlay construction techniques, somewhat related to the problem we have described, but designed for unreliable multicast and using a very different approach, have been independently proposed in [292, 80].

Although no longer a part of the mainstream, work on IP multicast based techniques has been ongoing until today, mostly in the context of high-performance clusters and grid computing. Some of the recent protocols include MDP [197], DyRAM [199], and NORM [4]. Other approaches and surveys can be found in [157, 27, 28, 19, 150, 184]. Generally, these systems appear to share weaknesses with the systems discussed earlier.

Meanwhile, starting in 1993 with the Isis toolkit [125, 42], and later systems such as Totem [222], Transis [95], Horus [295], Ensemble [145], JGroups [20, 21], and most recently, the Spread toolkit [13], another group of researchers focused on building group communication systems with strong reliability properties. Unlike the former group of systems, which were designed for scalability in the number of nodes, group communication systems often did not scale with the number of nodes, but did explicitly support a simultaneous use of multiple multicast protocols. Nevertheless, scalability in this dimension was still very limited: although Isis and Spread can support large numbers of *lightweight* multicast groups [13, 125, 262], the groups seen by applications are an illusion; in actuality, there is just one physical multicast group. In Isis, the group consists of the union of the members of the lightweight groups. Spread uses a slightly different variant of this approach: client systems connect to a smaller set of servers, and each application-level multicast is relayed through one of these; each server multicasts the requests of its clients in the physical group, and the remaining servers actively filter each incoming message, accepting it only if they have any clients in the lightweight group to which it is addressed; finally, messages are unicast to clients. Both techniques incur a high cost. In Isis, all nodes are burdened by undesired traffic in lightweight groups to

which they do not belong, whereas in Spread the servers are a point of contention, and the indirect communication pathway results in high latency.

A decade of work on IP multicast based reliable multicast protocols has also led researchers to realizing that the scalability of reliable multicast is, to a large degree, limited by the ability of the protocol to sustain stable throughput in spite of minor perturbations, which are common in real systems. The issue was particularly nicely illustrated in the work on Bimodal Multicast [43], where it has been shown that artificially freezing a selected node to simulate effects such as an excessive load or flaky network interfaces has a dramatic effect on throughput, to a degree that grows with system size. Our experience confirms these observations, and we believe that understanding the phenomena that cause instabilities and mechanisms that can be used to prevent them is an essential step towards realizing our vision. Much research has been done to address these problems by leveraging probabilistic techniques such as gossip, e.g., in Bimodal Multicast [43] or Lightweight Probabilistic Broadcast [107]. However, while probabilistic techniques are amazingly efficient at curing instabilities, they did not leave us with a complete understanding of what the underlying cause is, in part because the throughputs these systems achieved, at the level of 100-200 messages/s, were nowhere close to the hardware limits, and the somewhat degenerate scenarios that they were tested against were not quite representative of a healthy network, where, as our experience shows, minor perturbations and instabilities are still a major factor affecting scalability. Other attempts at addressing the instability problem included the work on multicast congestion control [299, 260, 311, 317, 165, 310]. Here, the focus was mostly on incorporating TCP-like flow control mechanisms and achieving fairness between different protocol instances.

In the period from 2000 until 2005, problems with the adoption of IP multicast by the Internet service providers (ISP) [94] spurred another wave of research, focusing on replacing hardware-based dissemination techniques such as IP multicast with *application-*

level multicast (ALM) protocols, in which multicast forwarding trees are formed by the end-hosts subscribed to a particular multicast group. Systems in this category include e.g., Overcast [164], NICE [23], NARADA [82], Scribe [70], Bullet [177], BitTorrent [84], Splitstream [69], and Chainsaw [242]. ALM protocols can be remarkably scalable, and perform well in tasks such as video streaming, for which they have been highly optimized. However, this over-specialization is also a weakness. Many of these systems, e.g., Overcast or NICE, are designed to disseminate data one-way from a single source to a large number of users, and would not be very useful for implementing a virtual room or similar functionalities with multiple interacting endpoints. Others, such as Scribe or SplitStream, do not have any end-to-end loss recovery mechanisms. Bullet and BitTorrent offer recovery mechanisms that are probabilistic in nature, and it is not clear what their specific guarantees are, and how useful they might be in the context of replication. Moreover, in most ALM systems, messages follow circuitous routes from source to destination, incurring very high latency. For example, SplitStream and Chaisaw have delays measured in seconds.

While the prior work on ALM is important, and the lack of a widespread adoption of hardware-based techniques makes it clear that the future Web will have to leverage some sort of an application-level scheme, the ALM systems share two common weaknesses.

First, scaling in the number of simultaneously active protocols is still largely ignored. Existing ALM systems fall roughly into two categories: systems like Overcast, which run each protocol independently, and systems like NARADA, which build a backbone tree-like or mesh-like network, and run multicast over this common structure. In the former group, overheads are linear with the number of protocols, and nodes can have unbounded indegrees, much as for the RMTP-like protocols discussed previously. In fact, the issue is even more acute, because nodes can also have unbounded outdegrees, and may be forced to forward traffic to large numbers of destinations. On the other hand,

in the latter systems, the mapping of the actual protocol onto the backbone structure often results in some of the client nodes having to relay high rates of messages that do not interest them, which incurs overheads, and due to the symmetric, peer-to-peer nature of these systems, this burden may be randomly assigned to clients that may not have sufficient capacity to play this role.

Second, existing ALM protocols tend to overload under-provisioned clients and fail to utilize spare capacity of the better provisioned ones: most of these systems are insensitive to the heterogeneity of the Internet, and either spread the load uncontrollably, or attempt at an even distribution, neither of which is desirable. Reports of bad performance of popular peer-to-peer systems in such settings [36, 48] have spurred research on *capacity-aware* overlays and supporting tools that attempt to distribute node indegrees, forwarding overheads and other load factors proportionally to the declared nodes capacities [301, 298, 324]. Much in the same way, earlier experiences with the unpredictability of routing in DHTs, on which many peer-to-peer schemes are based, spurred research on a *topology-aware* and *proximity-aware* overlay construction [68, 305, 319, 323], which avoids creating direct peering relationships between nodes separated by large distances or network latencies. As already mentioned earlier, we believe that the work on *capacity-aware*, *topology-aware* and *locality-aware* overlays is a step in the right direction, but also that ultimately, what we need is component-oriented techniques of the sort we propose in Chapter 3 that would allow federation of independently developed, and locally optimized systems, into global, Internet-wide structures.

To conclude this section, we briefly mention one other line of relevant prior work: *publish-subscribe* systems. Its beginnings can be traced back to the first experiments with USENET [108] in 1979, and the creation of the NNTP [168] protocol in 1986. A year later, a news tool in the Isis system was the first form of publish-subscribe implemented over a multicast protocol. In 1993, the concept of a *message bus* [236] was

proposed, which evolved the concept into the form that was later used within a number of commercial middleware products. Early *topic-based* publish-subscribe systems, such as Tibco Rendezvous [291], were essentially multicast substrates, but optimized towards a slightly different model, where events were handed off to the communication medium, which served the role of a reliable event store, and might be delivered at a later time, persisted for fault-tolerance or logged for querying, etc. The end-to-end guarantees might be weak or non-existent. Nevertheless, techniques used to implement recent topic-based publish-subscribe systems, such as Herald [60] or Corona [254], are similar to those used in ALM, and the distinction between the two has become somewhat blurry. The comments we have made earlier about ALM apply to those systems, too. Today, most research in this area seems to focus on the *content-based* publish-subscribe model, in which recipients might specify complex queries, thus effectively selecting a subset of the data stream. Gryphon [279] was the first such system; more recent examples include Siena [66] and Onyx [93]. The content-based publish-subscribe model has the potential to be useful in building the future Web. Often, the user would want to observe only a subset of events related to a particular object, for example an avatar in a virtual space might only receive updates on objects located within a 90-degree viewing angle ahead of it, and at a distance smaller than 100m. Nevertheless, a scalable content-based publish-subscribe substrate still needs to face many of the same performance and scalability challenges as the reliable multicast systems discussed earlier, and introducing expressive queries, by itself, does not contribute in any way to alleviating these problems. The same could be said about expressive variants of publish-subscribe that permit temporal queries, such as Cayuga [92]. The problems these system solve, albeit important, are to a large degree orthogonal to the types of scalability we focus on in this dissertation.

1.5 Contributions

In this section, we briefly summarize the key contributions made by this dissertation. A much more comprehensive discussion can be found in Chapter 6.

In Chapter 2, we propose a novel programming model that combines the distributed computing and object-oriented paradigms, and we describe some of the techniques used to implement a prototype of the system. Our new model and the prototype implementation have several advantages over the prior work in the area.

- Our approach is unique in that it creates the opportunity to combine the "best of breed" from several previously disjoint lines of work on distributed object embeddings, protocol composition, and protocol specifications. We have combined ideas from systems as diverse as Jini, Smalltalk, Ensemble, x-Kernel, BAST, WS-* family of specifications, I/O automata, and Object Linking and Embedding (OLE), to create a seamless whole that is extremely natural, intuitive, and appealing to the developers, yet sufficiently powerful to accommodate a variety of protocols within a single unified object-oriented environment.
- Unlike earlier approaches, discussed in Section 1.4.1, our object-oriented embedding is not limited to any single class of distributed objects, such as *remote objects*, *replicated objects*, or *publish-subscribe objects*, and may be the first development platform that is genuinely protocol-agnostic, in the sense that it can support *any* type of a distributed protocol as a reusable component. Existing embeddings are "protocol agnostic" only in the limited sense that one could choose among several different protocols to implement their distributed primitives.
- In contrast to existing distributed object systems, which focus on one category of application objects, such as replicated data structures or front-end components, our object model extends from the user-interface layer down to the hardware level, and may be the first such system designed for distributed multiparty components.

Earlier systems based on a similar vision, such as Jini, were fundamentally rooted in the client-server paradigm, and offered no explicit support for distributed protocols: mechanisms such as discovery, transactions, and notification, defined as distinguished infrastructure services in Jini, are simply *objects* in our framework.

- Our model for the composition of distributed protocols uses a behavioral notion of a distributed protocol type and supports functional composition in a language-agnostic manner, in contrast to many earlier protocol composition frameworks that typically either lacked the notion of typed composition, or defined protocol types by the types of the implementing classes and achieved composition through inheritance. Horus and Ensemble used behavioral protocol specifications, but these systems were not general-purpose platforms for component integration; their applicability was mostly limited to composing reliable multicast protocol stacks.
- Unlike earlier language-centric approaches, ours supports a clean integration with legacy applications. Our prototype has been used to implement "live" spreadsheets, in which ranges of cells are connected to multicast protocols, and to interface to relational databases via triggers.
- In contrast to most existing language-centric component integration platforms that either require communicating components to import the same external library and agree on a common interface, or use expensive techniques such as serialization to move data between the two connected parties, our platform completely eliminates notions such as "static dependency", "external library" and "object inheritance", and has a (limited) ability to connect certain pairs of binary-incompatible components by dynamically generating efficient proxy code.

In Chapter 3, we propose a novel architecture for hierarchical protocol composition that allows a single instance of a reliable multicast protocol to leverage different hardware mechanisms and different implementation strategies in different parts of the network and

at different levels in the hierarchy. Our approach has numerous advantages listed below. We are currently not aware of any prior work that would offer a similar range of benefits.

- Our technique does not rely on proxies or other infrastructure, and can generate efficient overlays or use mechanisms such as IP multicast to move data directly between the clients. While our architecture does rely on some infrastructure components, these components play the role similar to DNS, and do not participate in data dissemination or in the recovery protocol
- We present a protocol construction that uses a new kind of a membership service. Our membership service maintains membership information in a distributed form in the network, and rather than routing all membership events into a single point or disseminating information to all clients, uses its distributed information to assign roles to nodes running the protocol in a decentralized manner. Unlike most protocols, which rely on global sequence of membership views, protocols in this model are controlled by a decentralized, but consistently evolving hierarchy of membership views maintained by a background infrastructure.
- Based on our experience with the model and initial evaluation, we believe that the approach can be used to construct protocols with very strong reliability properties, in particular various forms of state machine replication, and might be generalized to support an even broader class of protocols.

In Chapter 4, we describe our implementation of a subset of the architecture discussed in Chapter 3, a simple high-performance reliable multicast system optimized for use in enterprise LANs and data centers. We report on our experiences and discuss architectural insights gained by building this system. Our work is novel in the following respects.

- We outline a new approach to scaling reliable multicast with the number of simultaneously running protocols, by leveraging structural similarities between the sets

of nodes running different protocols to amortize work across different protocol instances. While our technique is not completely general and does not yield benefits in all scenarios, and we lack realistic traces to evaluate its usefulness in practice, it appears to be a good match for a range of applications.

- Our system leverages IP multicast and delivers messages directly from senders to receivers, and does not use filtering, yet uses relatively few IP multicast addresses, thus avoiding the state explosion problem discussed in Section 1.4.3.
- We describe a previously unnoticed connection between memory usage and local scheduling policy in managed runtime environment (.NET), and multicast performance in a large system.
- We discuss several techniques, such as priority I/O scheduling and "pull" protocol stack architecture, developed based upon these observations, that can increase performance by reducing instabilities causing broadcast storms, unnecessary recovery and other disruptive phenomena. While the techniques we used are themselves not new, their use in this context, and the insights behind it, are novel.
- Our system achieves exceptional performance with low CPU footprint. The maximum throughput decreases by only a few percent while scaling from 2 to 200 nodes and degrades gracefully with loss and other types of perturbations.

Chapter 2
Programming

In this chapter, we describe the live distributed objects programming model. We explain how live objects can be defined, constructed, and composed. We define live object types and the meaning of notions such as *object reference* in the new model, and we explain how the new concepts are embedded into the development environment.

The chapter is divided into two parts. In Section 2.1, we focus on aspects of the programming model independent of any concrete implementation, such as introducing the definitions and concepts, overall design philosophy, types, language abstractions, and composition. In Section 2.2, we focus on our prototype implementation, in particular on the embeddings of the concepts introduced in Section 2.1 into a concrete operating system and a concrete programming language. Although the techniques discussed in Section 2.2 were used in a specific context, we believe they are fairly general.

2.1 Programming Model

2.1.1 Objects and their Interactions

To remind the definition introduced earlier: a *live distributed object* (or *live object*) is an instance of a distributed protocol executed by a set of components that may reside on different nodes and communicate by sending messages over the network; the notion of a live object encompasses all internal aspects of the protocol's execution, such as internal state, processing logic, and any network messages exchanged internally. A live object communicates with its environment via typed events, as in Smalltalk.

For example, a running instance of a reliable multicast protocol might be executed by a set of processes that have linked to a communication library implementing this

protocol, and that have created instances of a reliable multicast protocol stack. These reliable multicast protocol stack instances correspond to the "software components" in the above definition: each is located on a different node, and each maintains its own local state, such as a sequence of messages sent, received, acknowledged, stable, delivered, or persisted, perhaps the last received checkpoint etc. The different reliable multicast protocol stack instances might leverage IP multicast to communicate with one another, and they might create point-to-point TCP connections. The entire infrastructure consisting of the protocol stack instances, their local states, and their network communication, represents a running live distributed object. This live object may communicate with its software environment via events such as *send*(m) and *receive*(m), passed between the protocol stack instances located on different nodes and the instances of application code linked to those protocol stacks. From the point of view of the software environment, the exact form of state maintained by the protocol stack instances on each node, and the patterns of physical communication over the network, are irrelevant. The only aspect that matters is the kinds of events that may be exchanged with the protocol stack instances, and the various guarantees that the protocol might make in terms of the patterns of those events. For example, we may know that the protocol accepts *send*(m) events, and that there is a certain form of a guarantee that each of these events is eventually followed by a corresponding *receive*(m) event on each node, and that *receive*(m) events are generated by the protocol in a manner that preserves some form of ordering. This type of characterization of a protocol instance through events it consumes or produces and their patterns, and ignoring any implementation aspects including the exact form of network communication or internal data structures, is a key defining feature of our approach.

When node Y executes live object X, we'll say that *a proxy of live object X is running on Y*. Thus, a live object is executed by the group of its proxies (Figure 2.1). Proxies correspond to the "software components" or "protocol stack instances" mentioned above.

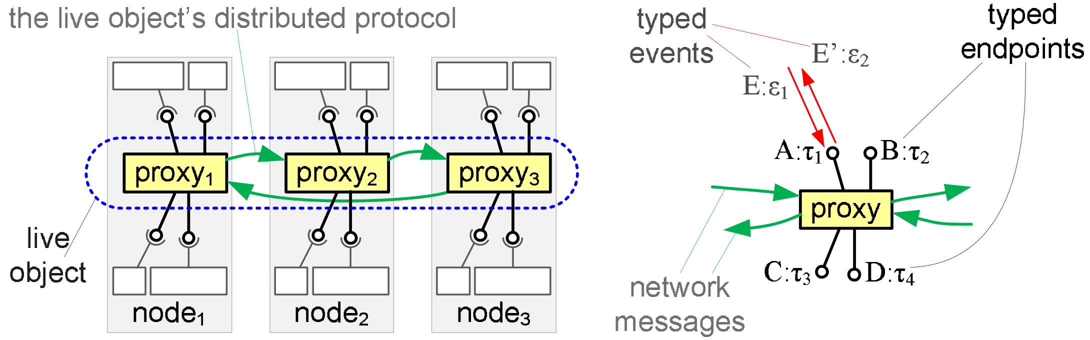

Figure 2.1: Live objects are executed by proxies that interact via endpoints. A proxy is a software component that runs a protocol instance on a node. To access a live object, a node starts the corresponding proxy. Proxies on different nodes can communicate by sending messages over the network. Proxies of different objects running on the same node can communicate via endpoints: strongly-typed, bidirectional event channels.

A live object proxy is a functional part of the object running on a node.

We model proxies in a manner reminiscent of I/O automata [196]. A proxy runs in a virtual context consisting of a set of *live object endpoints*: strongly-typed bidirectional event channels, through which the proxy can communicate with other software on the same node (Figure 2.1). Unlike in I/O automata [196], a proxy can use any external resources, such as local network connections, clocks, or the CPU. These interactions are not expressed in our model and they are not limited in any way. However, interactions of a live object proxy with any other component of the distributed system must be channeled through the proxy's endpoints.

All proxies of the same live object run that live object's *code*. Unlike in state machines [198, 267], we need not assume that proxies run in synchrony, in a deterministic manner, or that their internal states are identical. We *do* assume that each proxy of a live object X interacts with other components of the distributed system using the same set of endpoints, which must be specified as part of X's type. To avoid ambiguity, we sometimes use the term *instance of endpoint E at proxy P* to explicitly refer to a running event channel E, physically connected to and used by P.

Live objects have *behavioral types*, expressed in terms of the patterns of events their

proxies exchange with their software environment. Types will be discussed in more detail in the following section; here, we'll just limit ourselves to a brief example. Suppose that object *A* logs messages on the nodes where it runs, using a reliable, totally ordered multicast to ensure consistency between replicas. Object *B* might offer the same functionality, but be implemented differently, perhaps using a gossip protocol. As long as *A* and *B* offer the same interfaces and equivalent properties (consistency, reliability, etc.), we'll want to consider *A* and *B* to be implementations of the same type. The concept of behavioral equivalence is the key here; we define it more carefully in Section 2.1.2.

A natural concern to raise at this point is one related the notion of a live object's identity. One might expect that each live object has associated with it a unique identifier in some common global namespace. This is often the case for various types of distributed entities, such as multicast groups or web services, which are usually identified by some sort of a globally unique address. However, we do not make any assumptions about the existence of globally unique object identifiers. A live object in general might not have any sort of a global identifier at all, or it might have multiple independent identifiers.

To understand the reasons motivating this decision, consider a regular object in a desktop environment, for example a C++ object. The object resides in memory, hence one may think of it as being identified by its memory address. However, one may just as well create a C++ object in a section of shared memory mapped to different processes at different addresses, at which point an object's identity defined in this manner ceases to make sense. In a managed environment such as .NET, objects could be moved around, and their addresses might change. By modern standards, a system that uses memory addresses as identifiers would be considered buggy. In effect, regular objects lack any meaningful notion of a unique address or name. Yet despite this, we can still talk about their *identity*, in the sense of there being a single distinct individual entity that might encapsulate a logical unit of data or an internal thread of execution.

They key here seems to be the realization that an object reference is simply a description of how to access or interact with the same single entity that encapsulates the unit of data or processing, and the different references are all valid, functionally equivalent descriptions of this sort. A globally unique identifier or address is a special type of a description that relies on the existence of some common address space or a namespace.

Accordingly, we consider the live object's identity to be determined not by an identifier or by address of any sort, but by a complete description of the protocol instance, including the protocol's code as well as any parameters that might affect its execution, and that would distinguish it from different instantiations of the same protocol code. We will consider all software components executing this description at any point in time as running the same protocol instance, and thus representing parts of the same live object.

For example, an instance of a reliable multicast protocol that uses IP multicast and a membership service could be uniquely identified by the description of the protocol, including any parameters that might determine the precise semantics or control the built-in optimizations, plus parameters such as an IP multicast address used to transfer data, an IP address of the membership service, and the name of the multicast group within the namespace governed by this membership service. Any node that executes this description, with all the specific parameters embedded in it, will be appending messages to, and reading from the same message sequence. It is thus fair to say that all such nodes are executing the same protocol instance, or that they are running the same live object.

Accordingly, a *reference to a live object* X is simply a complete set of instructions for constructing and configuring a proxy of X on a node; this definition corresponds to the "complete description of the protocol instance" in the preceding discussion. Thus, when node Y wants to *access* live object X, node Y uses a reference to X as a recipe with which it can create a new proxy for X that will run locally on Y. The proxy then executes the protocol associated with X. For example, it might seek out other proxies

for X, transfer the current distributed state from them, and connect itself to a multicast channel to receive updates.

Unlike proxies, which can have state, references are just passive, stateless, portable recipes. In the context of what we have just discussed, it would make no sense for live object references to carry a mutable state; in order to ensure that all nodes constructing a proxy from the reference indeed follow the same protocol, we need to know that each of these nodes uses the exact same set of instructions. For the same reason, we do not use auto-discovery: as a description of a protocol instance, a live object reference cannot contain ambiguous elements that could be resolved differently depending on the network location or other local factors, thus resulting in different nodes taking inconsistent actions within the same protocol instance, using different instances of external services, or otherwise winding up with an inconsistent view of failures, membership, naming etc.

The instructions in a live object reference must be complete, but need not be self-contained. Some of their parts can be stored inside online repositories, from which they need to be downloaded. Rather than using a fixed API and a fixed method of identifying such repositories, such as URLs or other identifiers, we simply model the access protocols that need to be used to contact these repositories to download the missing part of the reference as live objects. The references to these access protocols are recursively embedded within the reference that uses this sort of indirection. This way, given a live object reference, a node can always *dereference* it without any prior knowledge of the protocol. An exception is thrown if dereferencing fails (for example, if a repository containing a part of the reference is unavailable).

Defining live objects through their references (complete descriptions of protocol instances) has profound implications. For example, we do not make any assumptions about "connectedness", i.e. the ability of nodes running the live object to reach one another. Indeed, we do not even assume that any communication takes place. In an

extreme example, a live object description may state that nodes executing it generate a certain type of an event and deliver it to the application at regular time intervals, without any synchronization. In this case, we will still think of all nodes executing this description as executing the same live object. Of course, this is a degenerate example; our goal is to potentially express more sophisticated protocols with complex behaviors.

One concern that may arise at this point is the question of membership. We have just stated that nodes executing the complete protocol description at any point in time are part of the same live object, but the set of such nodes might change, hence the definition might seem ambiguous. Again, our approach to modeling this aspect of protocol execution is to not make membership an explicit part of the model, but rather capture it as a dependency on another live object. Thus, we will not assume that the machines running the live object "know" about one another, or that their access to the protocol is somehow coordinated. If the knowledge of the set of components executing the live object or coordination of this sort are essential for the correct operation of the protocol, we will capture this as a dependency of this live object on another live object – one that represents an instance of a membership protocol. If the live object internally needs to make assumptions about connectivity, we will model this as a dependency on a membership protocol with specific guarantees in terms of handling partitioning failures. Similarly, we will not make any assumptions about the failure model. If the components running the protocol depend on such assumptions, these assumptions will again be captured as dependencies on other live objects that represent specific kinds of membership services or that encapsulate failure detection, discovery, and similar infrastructure protocols.

By now, it will not come as a surprise to the reader that we do not assume live objects to have replicated state. Thus, a live object could be an instance of a replicated state machine protocol tolerant of Byzantine failures, where local states evolve synchronously in a lock step manner, but it could also be an instance of a protocol where the local state

of protocol members is not synchronized in any way, an instance of a gossip protocol where these local states might be loosely synchronized, or an IP multicast channel, where no local state is kept.

Considering the generality of our definitions, the reader may worry that the model is not capable of expressing anything meaningful; after all, it is hard to express anything without making assumptions. Note, however, that in our approach, strong assumptions can indeed be expressed; assumptions about connectedness, synchrony, membership, failure models, state, and so on are not a built-in, core part of the basic model, but they could be expressed as dependencies on other live objects that represent protocols implementing the respective functionality. The given object is then defined as *requiring* those other objects for its correct operation, and its behavior in terms of failures, membership changes, partitioning events, and in various other such aspects, is then defined in terms of the patterns of events that the object exchanges with the objects it depends on.

This design is motivated by the desire to make our model flexible, and extend its applicability and benefits such as reusability and encapsulation as far towards the network and hardware as possible, to avoid imposing on the developers assumptions that might not fit their application scenarios. Also, it allows us to model aspects such as failure detection, network connectedness, membership, or synchrony using the same language, and the same behavioral type system.

To stress the fact that our model is designed to facilitate component integration and that it is designed to be used pervasively at all levels of application development, we adopt a somewhat radical perspective, in which the entire system, all applications and infrastructure, including low-level network services, such as failure detection, naming, membership, discovery, are live objects, typically expressed as compositions of, or connected to other, simpler objects. Accordingly, endpoints of a live object's proxy will be connected to endpoints exposed by proxies of other live objects running on the same

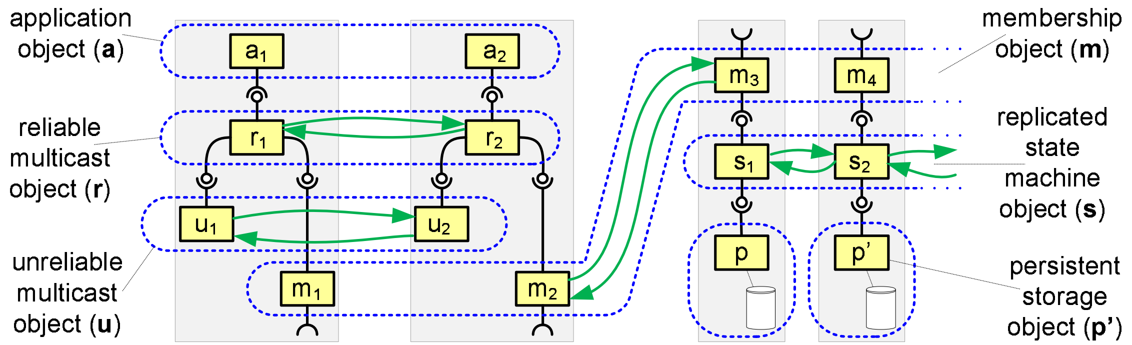

Figure 2.2: Applications are composed of interconnected live objects. Objects are "connected" if endpoints of some pair of their proxies are connected. The connected objects can then affect one-another by having their proxies exchange events through these endpoints. A single object can be connected to many other objects. Here, a reliable multicast object *r* is simultaneously connected to an unreliable multicast object *u*, a membership object *m*, and an application object *a*. The same object can be accessed by different machines in different ways. For example, membership object m is used in two contexts: by the reliable multicast object *r* (left), and by replicas of a membership service (right). The latter employs a replicated state machine s, which persists its state to replicas of a storage object p.

node (Figure 2.2). When proxies of two different objects *X* and *Y* are connected through their endpoints on a certain node *Z*, we'll say that *X and Y are connected on Z*. When two objects have proxies on overlapping sets of nodes, their respective proxies may connect and interact. We can think of the live objects as interacting through their proxies.

Example (a). Consider a distributed collaboration tool that uses reliable multicast to propagate updates between users (Figure 2.2) . Let *a* be an application object in this system that represents a collaboratively edited document. Proxies of *a* have a graphical user interface, through which users can see the document and submit updates. Updates are disseminated to other users over a reliable multicast protocol, so that everyone can see the same contents. The system is designed in a modular way, so instead of linking the UI code with a proprietary multicast library, the document object *a* defines a typed endpoint *reliable channel client*, with which its proxies can submit updates to a reliable multicast protocol (event *send*) and receive updates submitted by other proxies and propagated using multicast (event *receive*). Multicasting can then be implemented by a

separate object r, which has a matching endpoint *reliable channel*. Proxies of a and r on all nodes are connected through their matching endpoints. Similarly, object r may be structured in a modular way: rather than being a single monolithic protocol, r could internally use object u for dissemination and object m for membership tracking. Endpoints *unreliable channel* and *membership* serve as contracts between r and its internal parts u and m. Object m is an example of an "infrastructure" object; it provides r with a notion of a "group", which r uses internally to control its own configuration. ■

Figure 2.2 illustrates several features of our model. First, a pair of endpoints can be connected multiple times: there are multiple connections between different instances of the *reliable channel* endpoint of object r and the *reliable channel client* endpoint of a, one connection on each node where a runs. Since objects are distributed, so are the control and data flows that connect them. If different proxies of r were to interact with proxies of a in an uncoordinated manner, this might be an issue. To prevent this, each endpoint has a type, which constrains the patterns of events that can pass through different instances of the endpoint. These types could specify ordering, security, fault-tolerance or other properties. The live objects runtime will not permit connections between a and r, unless their endpoint types declare the needed properties.

A single object could also define multiple endpoints. One case when this occurs is when the protocol involves different roles. For example, the membership object m has two endpoints, for clients and for service replicas. The role of the proxy in the protocol depends on which endpoint is connected. In this sense, endpoints are like interfaces in object-oriented languages, giving access to a subset of the object's functionality. Another similarity between endpoints and interfaces is that both serve as contracts and isolate the object's implementation details from the applications using it. We also use multiple endpoints in object r, proxies of which require two kinds of external functionality: an unreliable multicast, and a membership service. Both are obligatory: r cannot

be activated on a platform unless both endpoints can be connected.

Earlier, we commented that not all live objects replicate their state. We see the latter in the case of the persistent store p. Its proxies present the same type of endpoint to the state machine s, but each uses a different log file and has its own state.

Our model promotes reusability by isolating objects from other parts of the system via endpoints that represent strongly typed contracts. If an object relies upon external functionality, it defines a separate endpoint by which it gains access to that functionality, and specifies any assumptions about the entity it may be connected to by encoding them in the endpoint type. This allows substantial flexibility. For example, object u in our example could use IP multicast, an overlay, or BitTorrent, and as long as the endpoint that u exposes to r is the same, r should work correctly with all these implementations. Of course this is conditional upon the fact that the endpoint type describes all the relevant assumptions r makes about u, and that u does implement all of the declared properties.

2.1.2 Defining Distributed Types

The preceding section introduced endpoint types, as a way to define contracts between objects. We now define them formally and give examples of how typing can be used to express reliability, security, fault-tolerance, and real time properties of objects.

Formally, the type Θ of a live object is a tuple of the form $\Theta = (E, C, C')$. E in this definition is a set of named endpoints, $E = \{(n_1, \tau_1), (n_2, \tau_2), , (n_k, \tau_k)\}$, where n_i is the name and τ_i is the type of the ith endpoint.

For example, if Θ is the type of reliable multicast objects like object r in Figure 2.2, it may define three endpoints, $E = \{e_r, e_u, e_m\}$, where e_r = ("*reliable channel*", τ_r) represents the reliable channel endpoint of type τ_r exposed to application objects such as a on Figure 2.2, e_u = ("*unreliable channel*", τ_u) represents the unreliable channel endpoint of type τ_u that needs to be connected to an unreliable multicast object such as u on

Figure 2.2, and finally, $e_m = (\text{"}\textit{membership}\text{"}, \tau_m)$ represents the membership endpoint of type τ_m that needs to be connected to a membership object such as m on Figure 2.2.

C and C' represent sets of constraints describing security, reliability, and other characteristics of the object (C), and of its environment (C'). C models constraints *provided* by the object, such as semantics of the protocol: guarantees that the object's code delivers to other objects connected to it. C' models constraints *required*, which are prerequisites for correct operation of the object's code. Constraints can be described in any formalism that captures aspects of object and environment behavior in terms of endpoints and event patterns. Rather than trying to invent a new, powerful formalism that subsumes all the existing ones, we build on the concepts of aspect-oriented programming [174], and we define C to be a finite function from some set A of aspects to predicates in the corresponding formalisms. For example, constraints $C = \{(a_1, \phi_1), (a_2, \phi_2), , (a_m, \phi_m)\}$ would state that in formalism a_1 the object's behavior satisfies formula ϕ_1, and so on.

Examples of various practically useful formalisms and constraints that can be expressed in these formalisms are discussed in the following section.

Type τ of an endpoint is a tuple of the form $\tau = (I, O, C, C')$. I is a set of *incoming events* that a proxy owning the endpoint can receive from some other proxy, O is a set of *outgoing events* that the proxy can send over this endpoint, and C and C' represent constraints provided and required by this endpoint, defined similarly to constraints of the object, but expressed in terms of event patterns, not in terms of endpoints (for example, an endpoint could have an event of type *time*, and with a constraint that time advances monotonically in successive events). Each of the sets I and O is a collection of named events of the form $E = \{(n_1, \epsilon_1), (n_2, \epsilon_2), \ldots, (n_k, \epsilon_k)\}$, where n_i is the name of the ith event and ϵ_i is its type.

For example, if τ_m is the type of a membership endpoint such as the one that r exposes to m in Figure 2.2, then I might include event $m_v = (\text{"}\textit{membership view}\text{"}, \epsilon_v)$

that proxies of r may receive from proxies of m, and O might include events m_j = ("*join group*", ϵ_j) and m_l = ("*leave group*", ϵ_l) that proxies of r can send to proxies of m. The three types of events would constitute the interface between r and m.

Event types can be value types of the underlying type system, such as .NET or Java primitive types and structures, or types described by WSDL [81], but not arbitrary object references or addresses in memory. We assume that events are serializable and can be transmitted across the network or process boundaries. As explained later, references to live objects are simply textual descriptions in a live object composition language, hence they are serializable, and can also be passed inside events. The subtyping relation on the event types is inherited from the underlying type system.

The purpose of creating endpoints is to connect them to other, matching endpoints, as described in Section 2.1.1 and illustrated on Figure 2.2. *Connect* is the only operation possible on endpoints. For endpoint types $\tau_1 = (I_1, O_1, C_1, C'_1)$ and $\tau_2 = (I_2, O_2, C_2, C'_2)$ we say that they *match*, denoted $\tau_1 \propto \tau_2$, when the following holds.

$$\tau_1 \propto \tau_2 \Leftrightarrow O_1 \rightsquigarrow I_2 \wedge O_2 \rightsquigarrow I_1 \wedge C_1 \stackrel{*}{\Rightarrow} C'_2 \wedge C_2 \stackrel{*}{\Rightarrow} C'_1 \qquad (2.1)$$

The relation $\rightsquigarrow$ between two sets of named events expresses the fact that events from the first can be understood as events from the second. Formally, we express this as follows:

$$E \rightsquigarrow E' \Leftrightarrow \forall_{(n,\epsilon)\in E}\ \exists_{(n,\epsilon')\in E'}\ \epsilon \lhd \epsilon' \qquad (2.2)$$

The operator $\lhd$ on types of any kind represents the subtyping relation in this dissertation. The types of events, and the corresponding subtyping relation, are inherited from the underlying object-oriented environment.

The relation $\stackrel{*}{\Rightarrow}$ between two sets of constraints expresses the fact that the constraints in the first set are no weaker than constraints in the second. Formally, we write this as:

$$C \stackrel{*}{\Rightarrow} C' \Leftrightarrow \forall_{(a,\phi')\in C'}\ \exists_{(a,\phi)\in C}\ \phi \stackrel{(a)}{\Rightarrow} \phi' \qquad (2.3)$$

Relation $\stackrel{(a)}{\Rightarrow}$ is simply a logical consequence in formalism a.

Intuitively, the definition of $\stackrel{*}{\Rightarrow}$ states that if C' defines a constraint ϕ' defined in some formalism a, then C must define a constraint ϕ that is no weaker than ϕ', also in formalism a. For example, if C' defines some reliability constraint expressed in temporal logic, then C must define an equivalent or stronger constraint, also in temporal logic, in order for $C \stackrel{*}{\Rightarrow} C'$ to hold.

The formal definition of $\tau_1 \propto \tau_2$ may be intuitively understood as imposing the following two conditions.

1. For each output event n of type ϵ of either endpoint, its counterpart must have an input event with the same name n, and with a type ϵ' such that $\epsilon \lhd \epsilon'$. This guarantees that if proxies of two live objects expose endpoints of type τ_1 and τ_2 and those endpoints are connected to one another, all events that originate in one of the proxies and are passed through its corresponding endpoint can be delivered through the other connected endpoint and correctly interpreted by the other proxy. This condition ensures that in a mechanical sense, communication between proxies connected by such endpoints is possible.
2. The provided constraints of each of the endpoints must imply (be no weaker than) the required constraints of the other. This ensures that the endpoints mutually satisfy each other's requirements. This will be further explained below.

For an example of the second condition, consider an application object a and a multicast channel object m connected through a pair of endpoints, both of which have the same name and types τ and τ', accordingly. Assume that the two types τ, τ' mechanically fit and define events through which proxies of a can issue multicast requests to proxies of m, and through which proxies of m can deliver multicast messages to proxies of a. Now, the type τ of a's endpoint might include a required constraint stating that messages are delivered at most once and in FIFO order, and a provided constraint declaring that its multicast requests will have unique identifiers and that they will never be issued twice.

The type τ' of m's endpoint might include a required constraint stating that multicast requests should have unique identifiers, and a provided constraint declaring that it will deliver messages at most once, but also at least once with probability 0.99, and in total order. The sets of constraints defined by τ and τ' are compatible: a satisfies the unique identifier requirement, and m provides the at most once and FIFO ordering semantics.

For a pair of endpoint types τ_1 and τ_2, the former is a subtype of the latter if it can be used in any context in which the latter can be used. Since the only possible operation on an endpoint is connecting it to another, matching one, the relation $\tau_1 \lhd \tau_2$ holds iff τ_1 matches every endpoint that τ_2 matches, i.e. $\tau_1 \lhd \tau_2 \Leftrightarrow \forall_{\tau'} ((\tau_2 \propto \tau') \Rightarrow (\tau_1 \propto \tau'))$, which after expanding the definition of $\propto$ can be formally expressed as follows:

$$\tau_1 \lhd \tau_2 \Leftrightarrow O_1 \rightsquigarrow O_2 \wedge I_2 \rightsquigarrow I_1 \wedge C_1 \overset{*}{\Rightarrow} C_2 \wedge C_2' \overset{*}{\Rightarrow} C_1' \tag{2.4}$$

Intuitively, $\tau_1 \lhd \tau_2$ if (a) τ_1 defines no more output events and no fewer input events than τ_2, (b) the types of output events of τ_1 are subtypes and the types of input events of τ_1 are supertypes of the corresponding events of τ_2, and (c) the provided constraints of τ_1 are no weaker and the required constraints of τ_1 are no stronger than those of τ_2.

Subtyping for live object types is defined in a similar manner. Type Θ_1 is a subtype of Θ_2, denoted $\Theta_1 \lhd \Theta_2$, when Θ_1 can replace Θ_2. Since the only thing that one can do with a live object is connect it to another object through its endpoints, this boils down to whether Θ_1 defines all the endpoints that Θ_2 defines, and whether the types of these endpoints are no less specific, and whether Θ_1 guarantees no less and expects no more than Θ_2. Formally, for two types $\Theta_1 = (E_1, C_1, C_1')$ and $\Theta_2 = (E_2, C_2, C_2')$, we define:

$$\Theta_1 \lhd \Theta_2 \Leftrightarrow E_1 \overset{*}{\preceq} E_2 \wedge C_1 \overset{*}{\Rightarrow} C_2 \wedge C_2' \overset{*}{\Rightarrow} C_1'. \tag{2.5}$$

Relation $\overset{*}{\preceq}$ between sets of named endpoints used above is defined as follows:

$$E \overset{*}{\preceq} E' \Leftrightarrow \forall_{(n,\tau') \in E'} \exists_{(n,\tau) \in E} \; \tau \lhd \tau' \tag{2.6}$$

The use of types in our platform is limited to checking whether the declared object contracts are compatible, to ensure that the use of objects corresponds to the developer's intentions. The live objects platform performs the following checks at runtime:

1. When a reference to an object of type Θ is passed as a value of a parameter that is expected to be a reference to an object of type Θ', the platform verifies that $\Theta \lhd \Theta'$.

2. When an endpoint of type τ is to be connected to an endpoint of type τ', either programmatically or during the construction of composite objects, the platform verifies that the two endpoints are compatible i.e. that $\tau \propto \tau'$.

Practical uses of endpoint matching and object subptyping relations in our platform are further discussed in Section 2.1.5.

We believe that in practice, this limited form of type safety is sufficient for most uses. For provable security, the runtime could be made to verify that live object's code implements the declared type prior to execution. Techniques such as Proof-Carrying Code (PCC) [226], and domain-specific languages with limited expressive power such as our Properties Language [238], could facilitate this.

2.1.3 Constraint Formalisms

In this section, we discuss different formalisms that can be used to express constraints in the definitions of object and endpoint types. The details of concrete formalisms and their concrete syntax are beyond the scope of this dissertation, and are an ongoing work. Our objective in building the model was not to lock into a specific formalism, such as temporal logic, but rather to provide an environment in which different existing formalisms can be embedded. An example of how such specifications fit into the live object definitions in our platform can be found in Section 2.2.2. We discuss our progress and plans

for future refinement in more detail in Section 6. Here, we will focus on explaining what types of formalisms can be embedded in our model, and how constraints in those formalisms can be defined in terms of proxies, endpoints, and events.

The issue is subtle because on the one hand, a type system will not be very helpful if it has nothing to check, but on the other hand, there are a great variety of ways to specify protocol properties. It is not much of an exaggeration to suggest that every protocol of interest brings its own descriptive formalism to the table. As noted earlier, many prior systems have effectively selected a single formalism, perhaps by defining types through inheritance. Yet when we consider protocols that might include time-critical multicast, IPTV, atomic broadcast, Byzantine agreement, transactions, secure key replication, and many others, it becomes clear that no existing formalism could possibly cover the full range of options; hence the need for extensibility. Our goal is to provide a framework, in which users can define custom formalisms, and then use those to annotate their objects.

As noted in Section 6, defining an ontology for formalisms is a future work at this point, but the general structure and an initial starting point at this stage is the following: the "mechanical" part of an object or endpoint type that lists the endpoints and events, along with their names, provides the basic structure. A constructed predicate that represents an endpoint or object constraint and that captures a certain aspect of behavior is then expressed in terms of this structure. A predicate can thus talk about endpoints being connected or not, it can talk about the occurrence of events flowing through specific endpoints, and although it cannot refer to specific instances of any given endpoint, it can potentially refer to the set of all instances of an endpoint, and express conditions such as "everywhere" (in all endpoint instances) or "somewhere" (in some of the endpoint instances), which correspond to general and existential quantifiers, as well as potentially to different notions of time, such as a global time, causality etc. This is perhaps best explained through examples.

For example, one formalism popular in the systems literature and possible to embed cleanly in our model is temporal logic [141, 79]. Here, we assume a global time and a set of locations, and a function that maps from time to events that occur at those locations. In the context of endpoint constraints, we can think of instances of the particular endpoint as the equivalent of "locations" or "nodes" in temporal logic formulas, and the endpoint's incoming and outgoing events, and the explicit *connect* and *disconnect* events assumed by our model, as the "events" of the temporal logic. Constraints are then expressed as formulas over these events, identifying the legal event sequences within the (infinite) set of system histories.

Example (b). Consider the *reliable channel* endpoint, exposed by the reliable channel r in the example in Section 2.1.1. The endpoint's type might define one incoming event *send*(m) and one outgoing event *receive*(m), parameterized by message body m. Constraints provided by the channel object r might include a temporal logic formula, which states that if event *receive*(m) is delivered by r through some of the instances of the endpoint sooner than *receive*(m'), then for any other instance of the endpoint, if both events are delivered, they are delivered in the same sequence.

Note how the formulas in this example have been constructed. They are built upon the following kinds of basic elements and patterns:

- event P occurs at ... instance of endpoint Q
- ... some of the instances of endpoint Q ...
- ... all of the instances of endpoint Q ...
- ... precedes ... at ... instance of endpoint Q
- if ... then ...

Note how each of these only refers to endpoints, occurrences of events at given endpoints, time and causality, and refers to the individual instances of endpoints either anonymously via an existential quantifier, or as a group via a general quantifier. ■

Readers familiar with group communication [40, 76] might wonder how to make the notion of *all instances of an endpoint* more rigorous. Indeed, the general quantifier refers to all instances of a given endpoint that might exist on any node in the network at any point in time, including the nodes running the object now, nodes that existed in the past, or nodes that will join the protocol and create a proxy and an instance of the endpoint in the future. Reliability properties, however, are usually expressed in terms of finite membership views with a fixed list of nodes. These views form a sequence, and new views are generated as nodes join or leave the protocol.

We have previously stated in Section 2.1.1 that to make the system flexible and extensible, we chose to not make strong assumptions about synchrony, failure models, or membership a part of the core model, but rather express them as dependencies on other live objects. An example of this was shown on Figure 2.2: there, the reliable multicast object r was connected to the membership object m. Proxies of r interacted through m with the membership service, registered as members of the group by passing events to proxies of m, and received membership views through their endpoints.

In order to express dependency on membership, the designer of the multicast object r will simply define a membership endpoint, and specify the requirements and guarantees made by the membership service as constraints on the endpoint type. The developer will then specify constraints on the occurrences of events between the communication and membership endpoints to express virtually synchronous semantics, much in the same way it is done in group communication specifications [79].

In the context of Example (b), the developer could thus define a constraint stating that if an event $send(m)$ occurs after event $view(k, s)$, but not after $view(k + 1, s')$ at any endpoint instance, then event $receive(m)$ cannot occur at any endpoint instance before $view(k, s)$, or after $view(k + 1, s')$. If the membership endpoint defines the appropriate conditions for events $view(k, s)$ to form a notion of a group, for example as in [79], then

this condition effectively constrains delivery of each message to a certain fixed and well defined set of participants. Note that this sort of a constraint still uses the same small number of elements and patterns introduced in the example; the only difference is that the condition now involves events on two types of endpoints. One can thus still express rigorous notions such as "group" in this model despite its apparent simplicity. Semantics in the presence of failures, partitioning, connectedness, and similar aspects can be captured in a similar way, through constraints that link events between two or more endpoints, one of which is an "infrastructure" endpoint that connects proxies of the object to proxies of an external membership, failure detector, or some other infrastructure object.

One issue with behavioral typing in general is the incompleteness of specifications, in a purely formal sense. Example (b) illustrates a safety property of a type for which temporal logic is especially convenient. In prior work [79], temporal logic has been used to specify a range of properties of reliable multicast protocols. However, the FLP impossibility result establishes that these protocols cannot guarantee liveness in traditional networks. Thus, while we can express liveness constraint in such a logic, no protocol could achieve it. In effect, one may question the usefulness of such a protocol type in real systems. We come back to this issue in Section 6.

Temporal logic is just one of many useful formalisms. Real-time and performance guarantees are conveniently expressed as probabilistic guarantees on event occurrences, e.g., in terms of predicates such as "at least p percent of the time, *receive*(m) occurs at all endpoint instances at most t seconds following *send*(m)," or "at least p percent of the time, *receive*(m) occurs at all different endpoint instances in a time window of at most t seconds". Note that these predicates are still confined to the basic structure laid out earlier, and talk about event occurrences at different endpoint instances using existential or general quantifiers, and only augmenting it with the probability of occurrence and the temporal distances between events at specific locations.

Yet another useful formalism would be a version of temporal logic that talks about the number of endpoint instances in time. For example, constraints of the sort “at most one instance of the *publisher* endpoint may be connected at any given time” could describe single-writer semantics and other similar assumptions made by the protocol designer. Constraints of this sort could also express various fault-tolerance properties, e.g., define the minimum number of proxies to maintain a certain replication level, etc. Again, the shape of these constraints again follows the basic pattern, this time augmenting the quantifiers with patterns such as “at most k endpoint instances” or “at least k endpoint instances”. Again, if these need to be made more rigorous and need to refer to numbers of endpoint instances within specific membership views, one can easily achieve that by defining such constraints in terms of events on endpoints connected to membership or other infrastructure objects.

In our work on a security architecture, still underway, we are looking into using a variant of BAN logic [57] to define security properties provided by live objects or expected from their environment. It seems that the structure laid out above, with endpoints, events, different kinds of quantification, defining events across two or more endpoints, and perhaps the use of time and causality, could be carried over to this setting as well.

In general, with formalisms like those listed above, type checking might involve a theorem prover, and hence may not always be practical. In practice, however, the majority of object and endpoint types would choose from a relatively small set of standard constraints, such as best-effort, virtually-synchronous, transactional, or atomic dissemination, total ordering of events, etc. Predicates that represent common constraints could be indexed, and stored as macros in a standard library of such predicates, and the object and endpoint types could simply refer to such macros. The runtime would then be able to perform type checking by comparing such lists, using cached “known” facts, for example that a virtually synchronous channel is also best-effort and ordered.

One might wonder why we do not simply compile a list of reliability, security, properties, and list all properties that a protocol satisfies. Following this idea, a virtually synchronous reliable multicast protocol would list *best effort* and *FIFO* among the many guarantees it provides. The reason for the more general definition is the desire to support new types of protocols, new types of guarantees, and indeed, even new types of formalisms, thus making the model extensible.

By taking advantage of late binding and reflection, features of .NET and Java platforms, it is easy to make these mechanisms extensible in a "plug and play" manner. We'll describe the way this is achieved in our platform in Section 2.2.2.

2.1.4 Language and Type System Embeddings

Our model has a good fit with modern object-oriented programming languages. There are two aspects of this embedding. On one hand, live object code can be written in a language like Java and C#, as shown in Section 2.2.2. On the other hand, live objects, proxies, endpoints, and connections between them are first-class entities that can be used within C# or Java code. The types that the runtime associates with these entities are based upon and extend the set of types in the underlying managed runtime environment. In this section, we'll discuss each of the new programming language entities we introduce. An example of their use is shown in Code 2.1.

A. References to Live Objects. Operations that can be performed on these references include reflection (inspecting the referenced object's type), casting, and dereferencing (the example uses are shown in Code 2.1, in lines 03, 05, and 06 accordingly). Dereferencing results in the local runtime launching a new proxy of the referenced object (recall from Section 2.1.1 that references include complete instructions for how to do this). The proxy starts executing immediately, but its endpoints are disconnected. A reference to the new proxy is returned to the caller (in our example it is assigned to a

```
    /* This is an example code of a handler for a proxy's incoming event. The
       event in this example is carrying a reference to some other live object. */
01  void ReceiveObject(ref<liveobject> ref_object)
02  {
     /* We can use reflection to inspect the referenced object's type at runtime,
        e.g., to make decisions based on reliability or security properties or needs. */
03    if (referenced_type(ref_object) is SharedFolder)
04    {
       /* Casting provides access to the desired set of endpoints. */
05      ref<SharedFolder> ref_folder := (ref<SharedFolder>) ref_object;

       /* Dereferencing a reference creates a local proxy, which immediately starts
          executing. At this point, custom wrappers might also be instantiated. */
06      SharedFolder folder := dereference(ref_folder);

       /* Communication with the proxy is only possible by connecting to one of
          its endpoints. We request the desired endpoint, by specifying its name.
          The exact type of the endpoint is pre-determined by the proxy's class. */
07      external<FolderClient> folder_ep := endpoint(folder, "folder");

       /* Unlike interfaces, endpoints do not expose methods; to communicate via
          the proxy's endpoint, we need to create a matching private endpoint. */
08      internal<FolderClient> my_ep := new_endpoint<FolderClient>();

       /* Here would be code that registers handlers for any incoming events that
          may arrive from the proxy through the newly created endpoint. */
09      my_ep.AddedElement += ...;

       /* After connecting, events start to flow in both directions, and callbacks on
          both sides of the connection are invoked to notify the communicating
          parties that the connection has been established. */
10      connection my_connection := connect(folder_ep, my_ep);

       /* The remaining code would store the connection reference for the duration
          of the session. Disposing the reference would terminate the connection. */
11      ...
12    }
13  }
```

Code 2.1: A live object event handler code in a C#-like language. We use a simplified syntax for legibility. Here, "ReceiveObject" is a handler of an incoming event of a live object proxy. The event is parameterized by a live object reference "ref_object". If the reference is to a shared folder, the code launches a new proxy to connect to the folder's protocol and attaches a handler to event "AddedElement" generated by this protocol, in order to monitor this folder's contents.

local variable *folder*). This reference controls the proxy's lifetime. When it is discarded and garbage collected, the runtime disconnects all of the proxy's endpoints and terminates it. To prevent this from happening, in our example code we must store the proxy reference before exiting (we would do so in line 11).

Whereas a proxy must have a reference to it to remain active, a reference to a live object is just a pointer to a "recipe" for constructing a proxy for that object, and can be discarded at any time.

An important property of object references is that they are serializable, and may be passed across the network or process boundaries between proxies of the same or even different live objects, as well as stored on in a file, etc. The reference can be dereferenced anywhere in the network, always producing a functionally equivalent proxy - assuming, of course, that the node on which this occurs is capable of running the proxy. In an ideal world, the environmental constraints would permit us to determine whether a proxy actually can be instantiated in a given setting, but the world is obviously not ideal. Determining whether a live object can be dereferenced in a given setting, without actually doing so, is probably not possible.

The types of live object references are based on the types of live objects, which we will define formally below. To avoid ambiguity, if Θ is a live object type, and x is a reference to an object of type Θ, we will write $ref{<}\Theta{>}$ to refer to the type of entity x.

The semantics of casting live object references is similar to that for regular objects. Recall that if a regular reference of type *IFoo* points to an object that implement *IBar*, we can cast the reference to *IBar* even if *IFoo* is not a subtype of *IBar*, and while as a result the type of the reference will change, the actual referenced object will not. In a similar manner, casting a live object reference of type $ref{<}\Theta{>}$ to some $ref{<}\Theta'{>}$ produces a reference that has a different type, and yet dereferencing either of these references, the original one or the one obtained by casting, result in the local runtime creating the same

proxy, running the same code, with the same endpoints. A reference can be cast to *ref*<Θ> without causing exception at runtime as long as the actual type of the live object constructed by this reference is a subtype of Θ.

B. References to Proxies. The type of a proxy reference is simply the type of the object it runs, i.e. if the object is of type Θ, references to its proxies are of type Θ. Proxy references can be type cast just like live object references. One difference between the two constructs is that proxy references are local and cannot be serialized, sent, or stored. Another difference is that they have the notion of a lifetime, and can be disposed or garbage collected. Discarding a proxy reference destroys the locally running proxy, as explained earlier, and is like assigning *null* to a regular object reference in a language like Java. The live object is not actually destroyed, since other proxies may still be running, but if all proxy references are discarded (and proxies destroyed), the protocol ceases to run, as if it were automatically garbage collected.

Besides disposing, the only operation that can be performed on a proxy reference is accessing the proxy endpoints for the purpose of connecting to the proxy. An example of this is seen in line 07, where the proxy of the shared folder object is requested to return a reference to its local instance of the endpoint named “folder”.

C. References to Endpoint Instances. There are two types of references to endpoint instances, *external* and *internal*. An external endpoint reference is obtained by enumerating endpoints of a proxy through the proxy reference, as shown in line 07. The only operation that can be performed with an external reference is to connect it to a single other, matching endpoint (line 10). After connecting successfully, the runtime returns a connection reference that controls the connection’s lifetime. If this reference is disposed, the two connected endpoints are disconnected, and the proxies that own both endpoints are notified by sending explicit *disconnect* events.

An internal endpoint reference is returned when a new endpoint is programmatically

created using operator *new* (line 08). This is typically done in the constructor code of a proxy. Each proxy must create an instance of each of the object's endpoints in order to be able to communicate with its environment. The proxy stores the internal references of each of its endpoints for private use, and provides external references to the external code per request, when its endpoints are being enumerated. Internal references are also created when a proxy needs to dynamically create a new endpoint, e.g., to interact with a proxy of some subordinate object that it has dynamically instantiated.

An internal reference is a subtype of an external reference. Besides connecting it to other endpoints, it also provides a "portal" through which a proxy that created it can send or receive events to other connected proxies. Sending is done simply by method calls, and receiving by registering event callbacks (line 09).

An important difference between external and internal endpoint references is that the former could be serialized, passed across the network and process boundaries, and then connected to a matching endpoint in the target location. The runtime can implement this e.g., by establishing a TCP connection to pass events back and forth between proxies communicating this way. This is possible because events are serializable.

Internal endpoint references are not serializable. This is crucial, for it provides isolation. Since any interaction between objects must pass through endpoints, and events exchanged over endpoints must be serializable, this ensures that an internal endpoint reference created by a proxy cannot be passed to other objects or even to other proxies of the same object. Only the proxy that created an endpoint has access to its "portal" functionality of an endpoint, and can send or receive events with it.

D. References to Connections. Connection references control the lifetime of connections. Besides disposing, the only functionality they offer is to register callbacks, to be invoked upon disconnection. These references are not strongly typed. They may be created either programmatically (as in line 10 in Code 2.1), or by the runtime during the

construction of a composite proxy. The latter is discussed in detail in Section 2.1.5.

E. Template Object References. Template references are similar to generics in C# or templates in C++. Templates are parameterized descriptions of proxies; when dereferencing them, their parameters must be assigned values. Template types do not support subtyping, i.e. references of template types cannot be cast or assigned to references of other types. The only operation allowed on such references is conversion to non-template references by assigning their parameters, as described in Section 2.1.5.

Template object references can be parameterized by other types and by values. The types used as parameters can be object, endpoint, or event types. Values used as parameters must be of serializable types, just like events, but otherwise can be anything, including *string* and *int* values, live object references, external endpoint references, etc.

Example (c). A channel object template can be parameterized by the type of messages that can be transmitted over the channel. Hence, one can, e.g., define a template of a reliable multicast stream and instantiate it to a reliable multicast stream of video frames. Similarly, one can define a template dissemination protocol based on IP multicast and parameterize in by the actual IP multicast address to use. A template shared folder containing live objects could be parameterized by the type of objects that can be stored in the folder and the reference to the replication object it uses internally. ■

F. Casting Operator Extensions. This is a programmable reflection mechanism. Recall that in C# and C++, one can often cast values to types they do not derive from. For example, one can assign an integer value to a floating-point type. Conversion code is then automatically generated by the runtime, and injected into this assignment. One can define custom casting operators for the runtime to use in such situations. Our model also supports this feature. If an external endpoint or an object reference is cast to a mismatching reference type, the runtime can try to generate a suitable wrapper, provided that such wrapper is available. Currently, the use of this feature in our system is very

limited; we'll comment on this further.

Example (d). Consider an application designed to use encrypted communication. The application has a user interface object *u* exposing a *channel* endpoint, which it would like to connect to a matching endpoint of an encrypted channel object. But, suppose that the application has a reference to a channel object *c* that is not encrypted, and that exposes a *channel* endpoint of type lacking the required security constraints. When the application tries to connect the endpoints of *u* and *c*, normally the operation would fail with a type mismatch exception. However, if the channel endpoint of *c* can be made compatible with the endpoint of *u* by injecting encryption code into the connection, the compiler or the runtime might generate such wrapper code instead. Notice that proxies for this wrapper would run on all nodes where the channel proxy runs, and hence could implement fairly sophisticated functionality. In particular, they could implement an algorithm for secure group key replication. In effect, we are able to wrap the entire distributed object: an elegant example of the power of the model. ■

The same can be done for object references. While casting a reference, the runtime may return a description of a composite reference that consists of the old proxy code, plus the extra wrapper, to run side by side (we discuss composite references in Section 2.1.5). In addition to encryption or decryption, this technique could be used to automatically inject buffering code, code that translates between "push" and "pull" interface, code that persists or orders events, automatically converts event data types, and so on.

Currently, our platform uses casting only to address certain kinds of binary incompatibilities, as explained in Section 2.2.2, but it is possible to extend the platform to support more sophisticated uses of casting, such as in the example above, and define rules for choosing among the available casting operators when more that one is found.

2.1.5 Construction and Composition

As noted in Section 2.1.4, a live object "exists" if references to it exist, and it "runs" if any proxies constructed from these references are active. Creating new objects thus boils down to creating references, which are then passed around and dereferenced to create running applications. Object references are hierarchical: references to complex objects are constructed from references to simpler objects, plus logic to "glue" them together. The construction can use four patterns, for constructing *composite*, *external*, *parameterized*, and *primitive* objects. We shall now discuss these, illustrating them with an example object reference that uses each of these patterns, shown in Code 2.2.

A. Composite References. A composite object consists of multiple "internal" objects, running side by side. When such an object is instantiated, the proxies of the internal objects run on the same nodes (like objects *r* and *u* in Figure 2.2). A composite proxy thus consists of multiple embedded proxies, one for each of the internal objects. A composite reference contains embedded references for each of the internal proxies, plus the logic that glues them together. In the example reference shown in lines 05 to 18 in Code 2.2, there is a separate section "**component** *name* : *reference*" for each of the embedded objects, specifying its internal name and reference. This is followed by a section of the form "**connection** *endpoint1 endpoint2*", for each internal connection. Finally, for every endpoint of some embedded internal object that is to be exposed by the composite object as its own, there is a separate section "**export** *endpoint*".

This would be a good moment to illustrate the use of formalisms discussed in Section 2.1.2. Recall that connecting a pair of endpoints of types τ and τ' is only legal if $\tau \propto \tau'$. Accordingly, a composite reference containing sections of the form "**connection** *endpoint1 endpoint2*" is only legal if the types of *endpoint1* and *endpoint2* match. These types can be determined based on the types of the corresponding objects, which can be either read directly off the reference, or deduced. In the live objects platform, the type

```
    /* an object based on a parameterized template */
01  parameterized object
     /* a template for label that can be collaboratively edited */
02    using template primitive object 0x3
03  {
04    parameter “Channel” :
       /* a complex object built from multiple component objects */
05      composite object
06      {
07        component “DisseminationObject” :
08          external object “MyChannel” as Channel
            /* parameters ‘‘Identifier’’ and ‘‘Channel’’ represent standard types */
09            from external object “QuickSilver” as Folder<Identifier, Channel>
              /* the standard, built-in, locally configured ‘‘registry’’ object */
10              from primitive object 0x2
11        component “ReliabilityObject” :
          /* specification of some loss recovery object, omitted for brevity */
12          . . .
        /* an internal connection between a pair of component endpoints */
13        connection
14          endpoint “UnreliableChannel” of “DisseminationObject”
15          endpoint “UnreliableChannel” of “ReliabilityObject”
        /* endpoints of the components to be exposed by the composite object */
16        export
17          endpoint “ReliableChannel” of “ReliabilityObject”
18      }
19  }
```

Code 2.2: An object reference using a shared document template. The template is parameterized by a reliable communication channel. The channel is composed of a dissemination object and a reliability object, connected to each other via their “UnreliableChannel” endpoints, much like objects *r* and *u* in Figure 2.2. The “ReliableChannel” endpoint of the reliability object is exposed by the composite object. The dissemination object reference is to be found as an object named “MyChannel” of type “Channel” in an online directory. The reference to the repository is retrieved, as an object named “QuickSilver” of type “Folder” containing channels, in yet another online repository, which is a standard object (the local *registry* object).

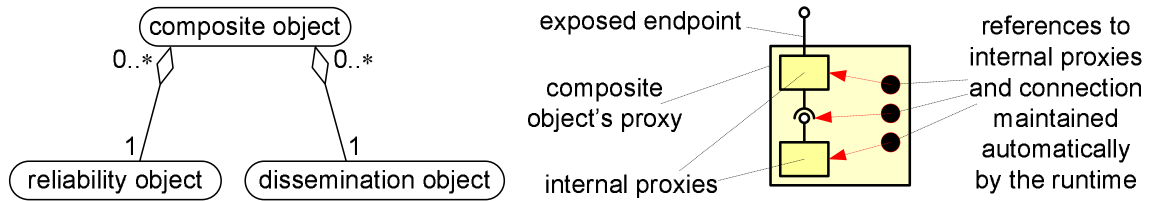

Figure 2.3: A class diagram and a proxy structure of a composite object. When constructing a proxy of a composite object, the runtime automatically constructs the proxies of all embedded objects, establishes connections between them, and stores all references for the embedded proxies and connections inside of a composite proxy. All embedded entities are garbage collected with the composite proxy. The composite proxy can expose some of the embedded proxy endpoints as its own.

check is performed statically, as well as dynamically. A static check is performed when the developer composes objects in a visual designer. A dynamic check is performed when a composite reference is dereferenced to create a proxy, or when two endpoints are connected programmatically.

When a proxy is constructed from a composite reference, the references to any internal proxies and connections are kept by the composite proxy, and discarded when the composite proxy is disposed of (Figure 2.3). The lifetimes of all internal proxies are thus connected to the lifetime of the composite. Embedded objects and their proxies thus play the role analogous to member fields of a regular object.

B. External References. An external reference is one that has not been embedded and must be downloaded from somewhere. It is of the form "**external object** *name* **as** *type* **from** *reference*", where *reference* is a reference to the live object that represents an access protocol for some online repository containing live object references, and *name* is the name of the object, the reference to which is to be retrieved from this repository. The type Θ of the retrieved object is expected to be a subtype of *type*, and the type of the external reference is *ref*<*type*>. One example of such a reference is shown in lines 08 to 10, and another (embedded in the first one) in lines 09 to 10.

The repository access protocol could be any object of type $\Theta \lhd$ *folder*, where type

folder is a built-in type of objects with a simple dictionary-like interface. Objects of this type have an endpoint with input event *get*(n) and with output events *item*(n, r) and *missing*(n). The condition $\Theta \lhd$ *folder* is another example of a type check that is performed statically, when an external reference is composed in a visual designer, as well as dynamically, when the runtime attempts to create a proxy of a protocol that will be used to download the missing information from the repository.

To retrieve an external reference, the runtime creates a repository access protocol proxy from the embedded reference, runs it, connects to its folder endpoint, submits the *get* event, and awaits response. Once the response arrives, the repository protocol proxy can be immediately discarded (or cached for future use). The reference retrieved from the repository is then type-checked against the *type* parameter in the "**as** *type*" clause. Its type *ref*<Θ> must be such that $\Theta \lhd$ *type*.

The "**as** *type*" clause allows the runtime to statically determine the type of the reference without having to engage in any protocol, which in turn enables static type checks at design time. In case of composite, parameterized, or primitive references, the runtime can derive the type right from the description, and this additional annotation is not needed. The "**as** *type*" clause can still be used in the other categories of references, however, as a form of an explicit type cast, e.g., to "hide" some of the object's endpoints.

The types in the reference (such as *Channel* in line 08 or *Folder*<*Id*, *Channel*>" in line 09) could either refer to the standard, built-in types, or they could be described explicitly using a language based on the formalisms in Section 2.1.2. To keep our example simple, we assume that all types are built-in, and we refer to them by names.

C. Parameterized References. These references are based on template objects introduced in Section 2.1.4. They include a section "**using template** *reference*", where *reference* is an embedded template object reference, and a list of assignments to parameter values, each in a separate section of the form "**parameter** *name* : *argument*",

where the *argument* could be a type description or a primitive value, e.g., an embedded object reference. For example, the reference in Code 2.2 is parameterized with a single parameter, *Channel*. The type of the parameter need not be explicitly specified, for it is determined by the template. In our example, the template expects a live object reference to a reliable communication channel. The specific reference used here to instantiate this template is the composite reference in lines 05 to 18.

Parameterized references are yet another example of a scenario that involves type checking. Whether we are dealing with a type parameter, or a value parameter that represents a live object or an endpoint reference, the runtime statically and dynamically checks that the type of the entity passed as a value of a given parameter is a subtype of the parameter type deduced from the embedded definition of the parameterized object.

D. Primitive References. The types of references mentioned so far provide means for recursively constructing complex objects from simple ones, but the recursion needs to terminate somewhere. Hence, the runtime provides a certain number of built-in protocols that can be selected by a "known" 128-bit identifier (lines 02 and 10 in Code 2.2). Of course even a 128-bit namespace is not infinite, and many implementations of the live objects runtime could exist, each offering different built-in protocols. To avoid chaos, we reserve primitive references only for objects that either cannot be referenced using other methods, or where doing so would be too inefficient. We will discuss two such objects: the *library* template and the *registry* object.

D1. Library. A library is an object of type *folder*, representing a binary containing executable code, from which one can retrieve references to live objects implemented by the binary. The library template is parameterized by URL of the location where the binary is located (see Code 2.3, lines 02 to 06). The binary can be in any of the known formats that allow the runtime to locate proxy code, object and type definitions in it, either via reflection, or by using an attached manifest (we show one example of this in

```
    /* my own, custom implementation */
01  external object "MyProtocol1" as MyType1
      /* an instance of the library template */
02    from parameterized object
        /* the global identifier of a built-in library template */
03      using template primitive object 0x1
04    {
05        parameter "URL" : "http://www.mystuff.com/mylibrary.dll"
06    }
```

Code 2.3: An example live object reference for a custom protocol. The protocol is implemented in a library downloadable from *http://www.mystuff.com/mylibrary.dll*. Objects running this protocol are of type "MyType1", and can be found in the library under name "MyProtocol1". The library template provides the folder abstraction introduced in Section 2.1.5

Section 2.1.2). After a proxy of a library is created, the proxy downloads the binary and loads it. When an object reference retrieved from a library is dereferenced, the library locates the corresponding constructor in the binary, and invokes it to create the proxy.

D2. Registry. The registry object is again a live object of type *folder*, i.e. a mapping of names to object references. The registry references are stored locally on each node, can be edited by the user, and in general, the mapping on each node may be different. Proxies of the registry respond to *get* requests by returning the locally stored references.

The registry enables construction of complex "heterogeneous" objects that can use different internal objects in different parts of the network, as follows.

Example (e). Consider a multicast protocol constructed in the following manner: there are two LANs, each running a local IP multicast based protocol to locally disseminate messages: local multicast objects x and y (Figure 2.4). A pair of dedicated machines on these LANs also run proxies of a tunneling object t, connected to proxies of x and y. Object t acts as a "repeater", i.e. it copies messages between x and y, so that proxies running both of these protocols receive the same messages. Now, consider an application object a, deployed on nodes in both LANs, and having some of its proxies

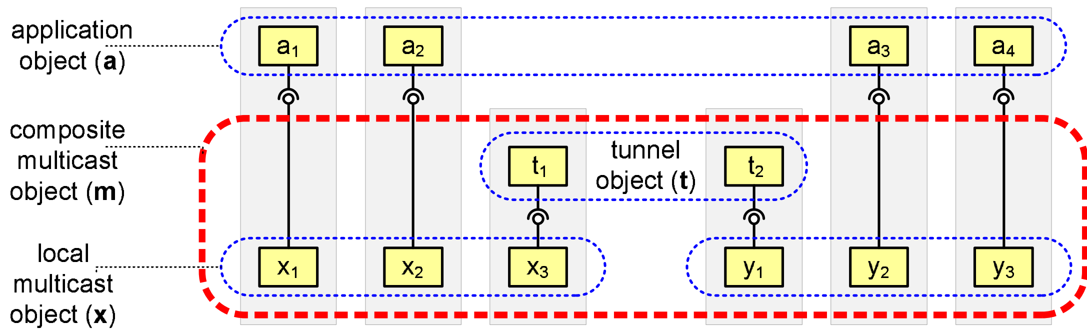

Figure 2.4: An example of a hybrid live object. The hybrid multicast object m is constructed from two local protocols *x*, *y* that disseminate data in two different regions of the network, e.g., two LANs, combined using a tunnel object *t* that acts as a repeater and forwards messages across the two LANs. Different proxies of the composite object *m*, running on different nodes, are configured differently. Some use an embedded proxy of object *x*, while others use an embedded proxy of object *y*.

```
01  external object “MyChannel” as Channel
02    from external object “MyPlatform” as Folder<Identifier, Channel>
        /* the registry object */
03      from primitive object 0x2
```

Code 2.4: A portable reference to the hybrid object of Figure 2.4. The use of registry allows for hiding the local configuration details by adding a level of indirection.

connected to *x*, and some to *y*. From the point of view of object *a*, the entire infrastructure consisting of *x*, *y*, and *t* could be thought of as a single, composite multicast object *m*. Object *m* is heterogeneous in the sense that its proxies on different machines have a different internal structure: some have an embedded object *x* and some are using *y*. Logically, however, *m* is a single protocol, and we’d like to be able to fully express it in our model. The problem stems from the fact that on one hand, references to *m* must be complete descriptions of the protocol, so they should have references to *x* and *y* embedded, yet on the other hand, references containing local configuration details are not portable. The registry object solves this problem by introducing a level of indirection (Code 2.4). ■

The reader might be concerned that the portability of live objects is threatened by use

```
01  parameterized object
02    using template external object “MyPlatform” as Folder<Identifier, Channel>
        /* from a binary downloaded from the url below */
03      from parameterized object
          /* the global identifier of a library template */
04        using template primitive object 0x1
05      {
06        parameter “URL” : “http://www.mystuff.com/mylibrary.dll”
07      }
08  {
09    parameter “LocalController” : “tcp://192.168.0.100:60000”
10  }
```

Code 2.5: An example of a correct use of the “registry” pattern. The registry object in this example is a locally configured multicast platform, which could then be used by external references like the one in Code 2.4. Here, the local instance of the communication platform is configured with the address of a node that controls a region of the Internet, from which other objects can be bootstrapped.

of the registry. References that involve registry now rely on all nodes having properly configured registry entries. For this reason, we use the registry sparingly, just to bootstrap the basic infrastructure. Objects placed in the registry would represent the entire products, e.g., “the communication infrastructure developed by company XYZ”, and would expose the *folder* abstraction introduce earlier, whereby specific infrastructure objects can be loaded. An example of such proper use is shown in Code 2.5.

2.1.6 Deployment Considerations

As mentioned before, object references are serializable and portable, hence “deploying” an object is as simple as disseminating its reference over the network. In Section 2.2, we explain how one can do so via a drag and drop interface.

In general, we want to think of a system as composed entirely of live objects. Thus, live object references would be stored inside, and used by, other live objects. Any live

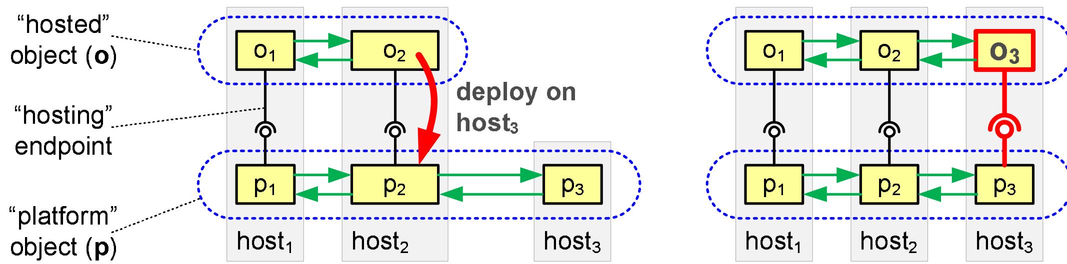

Figure 2.5: A live object dynamically controlling its own deployment. Objects may be able to dynamically create their own proxies, and control the lifetime of those proxies, by interacting with a "platform" object. The "platform" could expose a "hosting" endpoint that will be available to the proxies of objects "hosted" by the platform. Proxies could use this endpoint to request the instantiation of new proxies on other machines, or removing existing proxies. Objects could thus autonomously migrate between physical nodes, and "live in the network" in a fairly literal sense.

object that has a state, e.g., one based on a replicated state machine, a totally ordered reliable multicast protocol with state transfer such as virtual synchrony, etc., can keep serialized references to other objects as a part of that state. In Section 2.1.4, we also showed an example how live object code may dynamically instantiate proxies of other objects and connect to them. Hence, a live object that has state can also dynamically create and connect to other live objects, much in the same way as ordinary objects can create other objects, store them in their private member fields, or invoke their methods.

Not only can live objects dynamically spawn other objects, but they can also control their own deployment. To see this, consider a "platform" object p that runs on some set of nodes (Figure 2.5), and has a replicated internal state, in which it keeps references to various "hosted" objects and the sets of nodes on which they are to be deployed. The platform object p exposes a "hosting" endpoint, through which one could request a new proxy of some application object o to be deployed on some of the nodes on which p runs. In response to such request, p would update its replicated state, and the proxy of p running in the target location would spawn a proxy of o and maintain a reference to that proxy, thereby controlling this proxy's lifetime.

Suppose that p exposes the "hosting" endpoint to hosted object o, which can there-

fore issue its own hosting requests. Object *o* could now autonomously "migrate" between nodes as needed. For example, if *o* is a fault-tolerant backup object powered by some replication protocol, *o* can try to deploy its replicas on nodes that are using it frequently, or in proximity of such nodes, as well as control its replication level, and dynamically migrate out of nodes with flaky network interfaces, low on power, etc. In effect, live objects can create and store references to other objects, instantiate other objects and interact with them, and even autonomously "migrate" between nodes.

Although clearly useful, this capability also points to a potential security issue associated with the live objects framework, for it enables viral propagation by malfunctioning. or deliberately malicious objects. To prevent the abuse of this mechanism, in future large-scale deployments one would need a form of distributed resource control.

2.2 Prototype Implementation

Our implementation of the live object runtime runs on Microsoft Windows with .NET Framework 2.0. The system has two major components: an embedding of live objects into Windows drag and drop technologies, discussed here, and embedding of the new language constructs into .NET, discussed in Section 2.2.2.

2.2.1 OS Embedding via Drag and Drop

Our drag and drop embedding is visually similar to Croquet [272, 273], and mimics that employed in Windows Forms, tools such as Visual Studio (or similar ones for Java), and in the Object Linking and Embedding (OLE), XAML, and ActiveX standards used in Microsoft Windows to support creation of compound documents with embedded images, spreadsheets, drawings in vector graphic, etc. The primary goal is to enable non-programmers to create live collaboration applications, live documents, and business ap-

plications that have complex, hierarchical structures and non-trivial internal logic, just by dragging visual components and content created by others from toolbars, folders, and other documents, into new documents or design sheets.

Our hope is that a developer who understands how to create a web page, and understands how to use databases and spreadsheets as part of their professional activities, would use live objects to glue together these kinds of components, sensors capturing real-world data, and other kinds of information to create content-rich applications, which can then be shared by emailing them to friends, placing them in a shared repository, or embedding them into standard productivity applications.

Live object references are much like other kinds of visual components that can be dragged and dropped. References are serialized into XML, and stored in files with a "liveobject" extension. These files can easily be moved about. Thus, when we talk about emailing a live application, once can understand this to involve embedding a serialized object reference into an HTML email. On arrival, the object can be activated in place. This involves deserializing the reference (potentially running online repository access protocols to retrieve some of its parts), followed by analysis of the object's type. If the object is recognized as a user interface object, its proxy is then created and activated.

Live objects can also be used directly from the desktop browser interface. We configured the Windows shell to interpret actions such as doubleclick on "liveobject" files by passing the XML content of the file to the live objects runtime, which processes it as described above. Note that although our discussion has focused on GUI objects, the system also supports services that lack user interfaces.

We have created a number of live object templates based on reliable multicast protocols, including 2-dimensional and 3-dimensional desktops, text notes, video streams, live maps, and 3-dimensional objects such as airplanes and buildings. These can be mashed up to create live applications such as the ones on Figure 2.6.

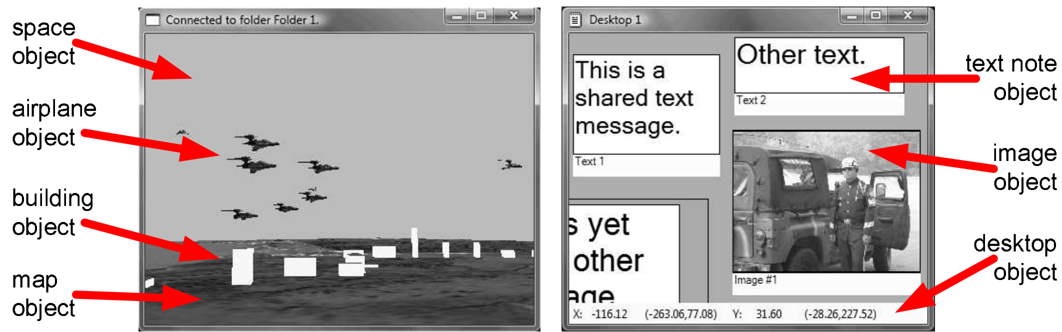

Figure 2.6: Screenshots of the live objects platform in action. The 3-dimensional spatial desktop, the area map embedded in this space, as well as each of the airplanes and buildings (left) are all separate live objects, with their own embedded multicast channels. Similarly, the green desktop and all text notes and images embedded in it are independent live objects. Each of the objects could be accessed from any location on the network, and separately embedded in other objects to create various web-style mashups, collaborative editors, online multiplayer games, and so on. Users create these by simply dragging objects into one another.

Although the images in Figure 2.6 are evocative of multi-user role-playing systems such as Second Life (SL) [189], live objects differ in important ways. In particular, live objects can run directly on client nodes, in a peer-to-peer fashion. In contrast, systems such as SL are tightly coupled to the data centers on which the content resides and is updated in a centralized manner. In SL, the state of the system lives in that data center. Live objects keep state replicated among users. When a new proxy joins, it must obtain some form of a checkpoint to initialize itself, or starts in a "null" state.

As noted earlier, live objects support drag and drop. The runtime initiates a drag by creating an XML to represent the dragged object's reference, and placing it in a clipboard. When a drop occurs, the reference is passed on to the application handling the drop. The application can store it as XML, or it can deserialized it, inspect the type of the dropped object, and take the corresponding action based on that. For example, in the spatial desktop on Figure 2.6, one is only allowed to drop objects with a 3-dimensional user interface. Likewise, the only types of objects that can be dropped onto airplanes are those that represent textures or streams of 3-dimensional coordinates. The decision

in each case is made by the application logic of the object handling the drop.

Live objects can also be dropped into OLE-compliant containers, such as Microsoft Word documents, emails, spreadsheets, or presentations. In this case, an OLE component is inserted with an embedded XML of the dragged object's reference. When the OLE component is activated (e.g., when the user opens the document), it invokes the live objects runtime to construct a proxy, and attaches to its user interface endpoint (if there is one). This way, one can create documents and presentations, in which instead of static drawings, the embedded figures can display content powered by any type of a distributed protocol. Integration with spreadsheets and databases is also possible, although a little trickier because these need to access the "data" in the object, and must trigger actions when a new event occurs.

As mentioned above, one can drag live objects into other live objects. In effect, the state of one object contains a reference to some other live object. This is visible in the desktop example on Figure 2.6. This example illustrates yet another important feature. When one object contains a reference to another (as is the case for a desktop containing references to objects dragged onto it), it can dynamically "activate" it: dereference, and connect to the proxy of the stored object, and interact with the proxy. For example, the desktop object automatically "activates" references to all visual objects placed on it, so that when the desktop is displayed, so are all objects, the references of which have been dragged onto the desktop.

By now, the reader will realize that in the proposed model, individual nodes might end up participating in large numbers of distributed protocol instances. Opening a live document of the sort shown on Figure 2.6 can cause the user's machine to join hundreds of instances of a reliable multicast protocol underlying the live objects embedded in the document. This leads to scalability concerns. Overcoming these problems has been the primary motivation for the work reported in Chapter 4.

2.2.2 Language Embedding via Reflection

Extending a platform such as .NET to support the new constructs discussed in Section 2.1.4 would require extending the underlying type system and runtime, thus precluding incremental deployment. Instead, we leverage the .NET reflection mechanism to implement dynamic type checking. This technique does not require modifications to the .NET CLR, and it should be possible to implement in other managed environments, such as Java. The idea is to use ordinary .NET types as "aliases" representing our distributed types. Whenever such an alias type is used in a .NET code, the live objects runtime understands that what is meant by the programmer is actually the distributed type.

The use of aliases is similar to the pseudo-code shown in Code 2.1, which resembles actual .NET code. The .NET type *SharedFolder* in lines 05 and 06 is an alias for a live object type, and the .NET type *FolderClient* in lines 07 and 08 is an alias for an endpoint type. When asked to perform operations on .NET objects of those types, such as casting in line 05, dereferencing in line 06, or connecting endpoints in line 10, the runtime uses its own metadata and the distributed type information it has previously collected to perform the operation. Thus, for example, during the casting process invoked in line 05, the live objects runtime will invoke its own subtyping code to check whether the casting is legal. In the process of performing this check, it will compare the lists of endpoints defined by the respective object types, and then it will compare the types of endpoints, by comparing the events, and finally checking whether the sets of constraints provided and required are compatible, exactly as it was described in Section 2.1.2. The only part of the type checking process that is not handled by the runtime is the comparison between individual constraints in the given formalism. To perform those checks, the runtime will invoke the custom code implementing those formalisms, which can be defined in an external library, much in the same way we can define custom live objects and custom live object and endpoint types. This will be discussed in more detail below.

```
   /* annotates ''IChannel'' as an alias for a live object type */
01 [ObjectTypeAttribute]
   /* defines a required constraint for the object type */
02 [ConstraintAttribute(ConstraintKind.Required, "0x1",
03   "at most one instance of Channel can be connected")]
04 interface IChannel<
   /* templates are modeled as .NET generics */
05   [ParameterAttribute(ParameterClass.ValueClass)] MessageType>
06 {
07   [EndpointAttribute("Channel")]
08   [ConstraintAttribute(ConstraintKind.Required, "0x2",
09   "event Send never precedes event Connect")]
10   [ConstraintAttribute(ConstraintKind.Provided, "0x2",
11   "event Send never precedes event Receive")]
12   EndpointTypes.IDual<
13     Interfaces.IChannel<MessageType>,
14     Interfaces.IChannelClient<MessageType>>
     /* returns an external reference to endpoint ''Channel'' */
15   ChannelEndpoint { get; }
16
17   [EndpointAttribute("Membership")]
18   [ConstraintAttribute(ConstraintKind.Required, "0x2",
19   "event Send never precedes event Connect")]
20   [ConstraintAttribute(ConstraintKind.Provided, "0x2",
21   "event Send never precedes event Receive")]
22   EndpointTypes.IDual<
23     Interfaces.IMembershipClient,
24     Interfaces.IMembership>
     /* returns an external reference to endpoint ''Membership'' */
25   MembershipEndpoint { get; }
26 }
```

Code 2.6: Declaring a live object type from an annotated .NET interface. The interface is associated with the live object type via an "ObjectType" attribute (line 01). The interface may then be used anywhere to represent the represented live object type. The live objects runtime uses reflection to parse such annotations in binaries it loads and build a library of built-in objects, object types and templates. Object and type templates are defined by specifying and annotating generic arguments (line 05). Constraints are also expressed by attributes (lines 02, 08, 10). The texts of those constraints (lines 03, 09, 11) is parsed by the user's pluggable implementation of the respective formalisms.

Alias types are defined by decorating .NET types with custom attributes, as shown in Code 2.6 and Code 2.7. For example, to define a new type of live objects, one defines a corresponding alias as a .NET interface annotated with *ObjectTypeAttribute* (line 01 in Code 2.6). When the runtime loads a new .NET library with custom code, it always first scans the binary for all .NET types annotated this way as aliases. It then uses .NET reflection mechanisms to analyze the structure of the .NET classes used as aliases, and parses all custom annotations to collect information about the distributed entities corresponding to those alias types. The collected metadata is then used by the runtime to extend the type system. It registers all newly encountered types of objects and endpoints, and definitions of objects and object templates, in its internal structures. When the next type check is performed, the newly collected types can be used, alongside with the built-in types of objects and endpoints. Indeed, the live objects runtime uses exactly the same mechanism to bootstrap its initial type system; it simply scans its own .NET libraries on startup to find the annotated classes and interfaces. Currently, the process is somewhat time consuming, and depending on the hardware, starting a new live object by double-clicking a ".liveobject" file can result in about 1s delay. However, once the process is started and the minimal typesystem has been bootstrapped, new libraries are loaded and analyzed on demand, and the process is much faster.

While the use of aliases is convenient as a way of specifying distributed types, alias types are, of course, not distributed, and the .NET runtime does not understand sub-typing rules defined in Section 2.1.2. The actual type checking is done dynamically. When the programmer invokes a method of an alias to request a type cast or to connect endpoints, the runtime uses its internal list of aliases to identify the distributed types involved and performs type checking itself. The physical .NET types of aliases are irrelevant. Indeed, if the runtime determines that two different .NET types are actually aliases for the same distributed type, it will inject a wrapper code, as explained further

in this section.

In the example of Code 2.6, the runtime would register *IChannel* as an alias for a type of live objects that have two endpoints named “Channel” and “Membership”. This information is derived from the structure of the alias. An alias for a live object type is expected to be a .NET interface that has only .NET properties as its members (lines 07-15 and 17-25), each property corresponding to one named endpoint that all live objects of this type will expose. The property has to be annotated with *EndpointAttribute* (lines 07 and 17), and it can only have a getter (lines 15 and 25), which must return a value of a .NET type that is already an alias for some endpoint type. In the example in Code 2.6, *EndpointTypes.IDual<Interface1*, *Interface2>* (lines 12-14 and 22-24) is expected to be an alias for a type of endpoints. When building its metadata structures for the live object type described by alias *IChannel*, the runtime will lookup its list of aliases to locate *EndpointTypes.IDual*, and if one does not exist, it will parse the .NET type *EndpointTypes.IDual* to create an alias based on its annotations (if the type is not annotated as an alias, a runtime exception will be thrown). In this example, the runtime will find that this alias describes a type of endpoints that is parameterized. It will have to first resolve aliases *Interface1* and *Interface2*, and then substitute the result to create the endpoint type. Finally, it will use the resulting endpoint type as the type of the endpoint “Channel” or “Membership”, accordingly, in the definition of the live object type that is going to be represented by alias *IChannel*.

We just saw that aliases can take parameters, as it was the case in the example with *EndpointTypes.IDual* above. Such aliases represent parameterized entities. For example, we could define a live object type template parameterized by the type of another live object. A practical use of this is a typed folder template, i.e. a folder that contains only references to live objects of a certain type. For example, an instance of this template could be a folder that contains reliable communication channels of a particular type.

Another good example is a "factory" object that creates references of a particular type, e.g., an object that configures new reliable multicast channels in a multicast platform.

The example alias in Code 2.6 also describes a parameterized template; instances of this template represent channels that can only transmit messages of particular types. Parameters of the represented live object type are modeled as generic parameters of the alias. Each generic parameter is annotated with *ParameterAttribute* (line 05), to specify the kind of parameter it represents. The classes of parameters supported by the runtime include *Value*, *ValueClass*, *ObjectClass*, *EndpointClass*, and a few others that are not important in our discussion. *Value* parameters are simply serializable values, including .NET types or references to live objects. The others represent the types of values, types of live objects, and types of endpoints. In this example, the generic parameter *MessageType* is annotated as a *ValueClass*, i.e. a type of serializable events. Note how *MessageType* is then used as an alias to parameterize the types of endpoints (lines 13, 14, 23, and 24 in Code 2.6). Indeed, if an object represents channel for messages of a particular type, then the endpoints exposed by proxies of this object will define *send*(*m*) and *receive*(*m*) events as carrying messages *m* of that type.

The last aspect of the live object type definition in Code 2.6 is the manner in which we specify constraints. Each constraint is defined as a custom attribute of type *ConstraintAttribute*, applied either to the definition of the interface (for constraints applied to object types, as in lines 02-03), or to the particular endpoint (for constraints applied to endpoint types, as in lines 08-09, 10-11, 18-19, and 20-21). When the runtime parses the annotations for an alias type, and finds an annotation with *ConstraintAttribute*, it uses it to extend the metadata of the respective live object or endpoint type with information representing a constraint. Constraint information is then used during type checking.

For example, in Code 2.6, in lines 08-09 there is a custom attribute specifying a required endpoint constraint. The attribute takes as an argument a 128-bit hexadecimal

number (0x2) that identifiers a particular formalism, and then a string that represents some constraint in this formalism (as in line 09). In this example, constraint strings are simply English sentences; as mentioned before, discussion of the exact syntax of any particular formalism is outside the scope of this dissertation.

When the runtime encounters such custom attribute, it looks for an external library that defines the formalism with this identifier (0x2). Definition of new formalisms resemble the definitions of alias types. Each such definition would be an ordinary .NET class annotated with a custom attribute *ConstraintClassAttribute*, which instructs the live objects runtime that will parse the library for annotations that the class annotated this way should be registered on the list of known formalisms, much in the same way aliases are registered on the list of known aliases.

The class defining the formalism should implement a method *Parse*, which takes as an argument a string representing a constraint, and the structural information about the endpoint, such as the list of incoming and outgoing events it defines. The constraint string is passed as the third argument to *ConstraintAttribute* (as in lines 03, 09, 11, 19, and 21), and the endpoint information is built by the runtime by analyzing the structure of the alias type, as explained earlier. The *Parse* method returns an object that represents a constraint. As mentioned before, the returned constraint object is then attached to the endpoint or object type, as a part of its description, and used during type checking.

Constraint objects are mostly opaque to the runtime, except a few standard methods they must all define, including *IsWeakerThan*. The method takes as an argument another constraint object representing a constraint in the same formalism. When the runtime needs to compare two endpoint types to determine if one is a subtype of or a match for another, it compares constraints in the matching formalisms by invoking the *IsWeakerThan* method on pairs of constraint objects to test whether one constraint implies another. The constraint objects are taken from the internal metadata of the types

being compared. The exact implementations of *Parse* and *IsWeakerThan*, and the language and syntax or constraints, are up to the developer who designs the respective formalism. The only thing different formalism have in common is that they build their constraint objects based on the same kind of structural information: a list of endpoints, events, their names, and types. Additional abstractions, such as time or causality, are concepts internal to the individual formalisms.

Having defined the object's type, we can define the object itself. This is again done via annotations. An example definition of a live object template is shown in Code 2.7. A live object template is again defined as a .NET class, the instances of which represent the object's proxies. The class is annotated with *ObjectAttribute* (line 01) to instruct the runtime to build a live object definition from it. This template has two parameters: the type parameter representing the type of messages carried by the channel (line 03), and a "value" parameter - the reference to the naming infrastructure object that this channel should use (lines 08-09). To specify the type of the live object, one inherits from an alias .NET interface representing a live object type (line 03). This forces our class to implement properties returning the appropriate endpoints (lines 19-25). The actual endpoints are created in the constructor (lines 11-12). While creating endpoints, we connect event handlers for incoming events (hooking up as in line 12, and implementing these handlers, as in line 27).

To conclude this section, we discuss one important aspect of the language embedding related to the use of aliases. We have previously stated that two .NET types can serve as aliases to the same distributed type, and that the runtime would treat them as interchangeable. The situation is actually quite common, since it relates to the way we support component integration: rather than having two live object implementations import a shared library, the developers must actually define the types of objects endpoints within their .NET libraries. For example, suppose that binary *Foo.dll* created

```
01 [ObjectAttribute("89bf6594f5884b6495f5cd78c5372fc6")]
02 sealed class MyChannel<
03   [ParameterAttribute(ParameterClass.ValueClass)] MessageType>
04 : ObjectTypes.IChannel<MessageType>, /* specifies the live object type */
05   Interfaces.IChannel /* we implement handlers to incoming events, see line 12 */
06 {
07   public MyChannel(
08     [Parameter(ParameterClass.Value)] /* also a parameter of the template */
09     ObjectReference<ObjectTypes.INaming> naming_reference)
10   {
11     this.myendpoint = new Endpoints.Dual<
12       Interfaces.IChannel, Interfaces.IChannelClient>(this);
13     ... /* the rest of the constructor would be similar to that in Code 2.1 */
14   }
15   /* this is our internal ''backdoor'' reference to the channel endpoint */
16   private Endpoints.Dual<
17     Interfaces.IChannel, Interfaces.IChannelClient> myendpoint;
18
19   EndpointTypes.IDual<
20     Interfaces.IChannel<MessageType>,
21     Interfaces.IChannelClient<MessageType>>
22   ObjectTypes.IChannel.ChannelEndpoint
23   {
24     get { return myendpoint; } /* returns an external endpoint reference */
25   }
26    /* this is a handler for one of the incoming events of the channel endpoint */
27   Interfaces.IChannel.Send(MessageType message) { ... } /* details omitted */
      /* the rest of the alias definition, containing all the other event handlers etc. */
28   ...
29 }
```

Code 2.7: Declaring a live object template by annotating a .NET class. The .NET class is decorated with "ObjectAttribute" (line 01). It can have generic parameters (line 03), as well as constructor parameters (line 08), all of which are parameters to the template. To specify the template live object's type, the class must implement an interface that is annotated to represent a live object type (line 04, referencing the definition shown in Code 2.6). In the body of the class, we create endpoints to be exposed by the proxy (created in lines 11-12, exposed in lines 19-25), handle incoming events (line 27) and send events through its endpoints.

by one developer defines an object type alias *IChannel* as in Code 2.6, and an object template alias *MyChannel* as in Code 2.7. Now, suppose that a different, unrelated binary *Bar.dll* created by another developer also defines an alias *IChannel* in exactly the same way, as in Code 2.6, and then uses this alias, e.g., in the definition of an application object that could use channels of the corresponding distributed type. If someone now creates a composite objects that uses components defined in *Foo.dll* and *Bar.dll* and connects their endpoints, both binaries are loaded by the live objects runtime, and we end up with two distinct and binary-incompatible .NET aliases *IChannel*, representing the same distributed type. When the runtime attempts to connect endpoints, or when the programmer makes an assignment between these two alias types, a binary mismatch occurs: even though both aliases represent the same endpoint type, their .NET types are not the same, and an assignment is not legal in .NET.

The mechanism used to resolve this problem is based on the casting extensions discussed previously in Section 2.1.4. In general, whenever the programmer casts one alias type X to another alias type Y, the runtime will not perform a simple .NET cast, but run a custom conversion. First, the runtime identifies the distributed types represented by the two aliases. Then, the subtyping code is invoked to test whether one distributed type is a subtype of another. In particular, if X and Y are aliases of the same distributed type, this test will always succeed. Then, the runtime generates a wrapper code that implements the .NET interface of alias Y, and then invokes its constructor, passing an object of type X as an argument. The wrapper initialized this way is returned as a result of casting. The wrapper now functions as if it were an entity of the type represented by alias Y. For example, if Y is the type of a live object, the wrapper will implement getters for properties representing individual endpoints. When methods of Y are invoked, the wrapper routes those calls to methods of X, perhaps rearranging or casting arguments and results accordingly. For example, the wrapper will implement a getter for a property

representing an endpoint by invoking the respective getter of a property defined by X, perhaps running a cast if necessary. In effect, the wrapper acts as a façade to a "backend" object of a .NET type X, exposing a "frontend" that looks like Y.

The code of the wrapper is dynamically generated, compiled, and loaded by the live objects runtime, so that the wrapper code can be efficient. In general, conversion between every pair of .NET types annotated as aliases requires different code because method calls might need to be forwarded differently, and the arguments and results of method calls might need to be adjusted differently. If the runtime were to generate a separate wrapper code for each pair of aliases, the space overhead of such implementation could be as high as $\Theta(N^2)$, where N is the total number of aliases. To avoid this, the wrapper code is separated into a *frontend* and a *backend*. In a wrapper that casts from X to Y, a Y-specific frontend code implementing interface Y takes an X-specific backend that encapsulates an object with interface X. The interface between the frontends and backends is standardized, so that each frontend code can be glued to each backend. For each alias type, frontend and backend code is generated in advance, at the time the library containing the alias is loaded. Later, when the runtime needs to create a wrapper, it simply picks the appropriate frontend and backend and assembles them together. The advantage of this solution is that its space overhead is $\Theta(N)$; there is no need for a custom wrapper for each pair of aliases, only a separate frontend/backend pair for each alias. Additionally, code generation in this scheme happens only once every time a new .NET library is loaded into the process, since all the frontends and backends are generated at once into the same source file, compiled together and loaded as a single .NET assembly. This allows the system to reduce overhead resulting from the need to represent each chuck of dynamically generated code as a separate .NET assembly, with all its metadata that occupies system memory.

Chapter 3
Extensibility

In this chapter, we describe an architecture that supports structural composition of the sort discussed in Section 1.3, and exemplified on Figure 1.3. The architecture is compatible with the programming model described in the preceding chapter, and indeed, it has been specifically designed to support it. The reader will easily recognize the similarity between concepts such as live object proxy from Chapter 2 and a protocol agent. On the other hand, the architecture described here can also be used independently, e.g., as a basis, on top of which one can design a new interoperability standard similar to the WS-* family of specifications. To maintain the "generic" flavor of this section, we will avoid direct references to live objects throughout most of the discussion. Instead, we will present the architecture as a general-purpose framework for reliable publish-subscribe eventing, and we will relate the discussion to existing web services standards in this category, specifically WS-Eventing and WS-Notification.

While the presentation is focused on multicast, it should be noted that our approach is more general, and can be used to run a broad variety of protocols. In Chapter 5, we give a brief overview of the Properties Framework, a new programming language that translates programs in a simple declarative rule-based language into code that runs on top of the architecture discussed here, and that can support protocols as diverse as virtual synchrony, consensus, and transactions, along with weaker models.

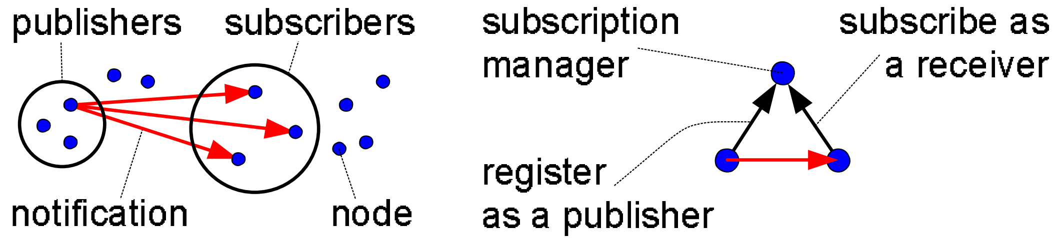

Figure 3.1: Nodes register for topics with a subscription manager. In general, there can be multiple publishers and multiple subscribers in each topic. The subscription manager is a logically separate entity that may be independent from the publishers.

3.1 Introduction

3.1.1 Terminology

In the following discussion, we employ the standard terminology found in the publish-subscribe literature, where *events* are associated with *topics*, produced by *publishers* and delivered to *subscribers*, and the prospective publishers and subscribers register with a *subscription manager* (Figure 3.1).

Unlike in some of the existing standards and middleware systems, here we assume that the distinction between publishers, subscribers, and subscription managers is purely functional. A single node might play more than one role, and different roles might be played by different nodes. In particular, publishers can be decoupled and independent from subscription managers, and there might exist multiple publishers for a single topic. In many applications, e.g., in replication, nodes could be thought of as simultaneously playing the roles of publishers and subscribers. In the context of replication, we'll sometimes use terms "group" and "topic" interchangeably. Formally, we define "*group X*" as "the set of subscribers for topic *X*".

Similarly, we'll treat "subscription manager" as a logical concept that does not need to refer to a single physical entity. It may be replicated to tolerate failures, or hierarchical, to scale. Different publishers or subscribers may contact different replicas or proxies

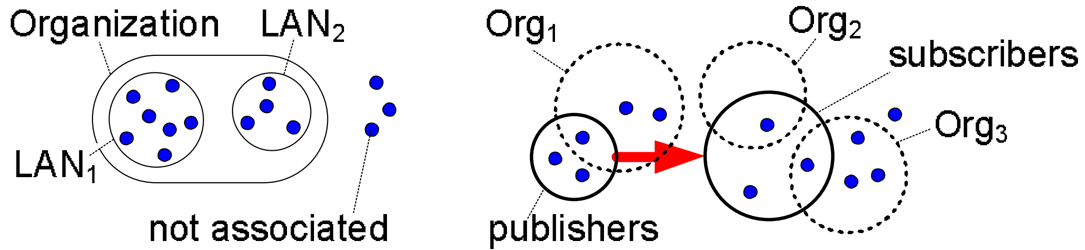

Figure 3.2: Nodes can be scattered across several administrative domains. These domains can be further hierarchically divided into administrative sub-domains.

of the subscription manager via different channels. A single manager may track publishers and subscribers for multiple topics. Many independent managers may coexist.

Nodes may reside in many *administrative domains* (LANs, data centers, etc.). It is often convenient to define policies, such as for message forwarding or resource allocation, in a way that respects domain boundaries, e.g., for administrative reasons, or because communication within a domain is cheaper than across domains, as it is often related to network topology. Accordingly, we will assume that nodes in the same domain are jointly managed, but nodes in different domains may have to be configured differently. Publishers and subscribers might be scattered across organizations, so these would need to cooperate in message dissemination. This presents a logistic challenge (Figure 3.2).

3.1.2 Technical Challenges

Our primary objective and the key technical challenge is to support efficient structural composition in a manner preserving a degree of genericity, so that the resulting infrastructure could support a broad variety of protocols. However, as an interoperability layer, our architecture is also aimed at addressing the specific weaknesses of the existing interoperability technologies for scalable eventing, such as WS-Notification and WS-Eventing. The latter technologies share the following limitations.

1. *No forwarding among recipients*. Many content distribution schemes build overlays within which content recipients participate in message delivery. In current web services notification standards, however, recipients are passive (limited to data reception). For example, given the tremendous success of BitTorrent for multicast file transfer, one could imagine a future event notification system that uses a BitTorrent-like protocol for data transfer. But BitTorrent depends on direct peer-to-peer interactions by recipients.

2. *Not self-organizing*. While both standards permit the construction of notification trees, such trees must be manually configured and require the use of dedicated infrastructure nodes ("proxies"). Automated setup of dissemination trees by means of a protocol running directly between the recipients is often preferable, but the standards preclude this possibility.

3. *Weak reliability*. Reliability in the existing schemes is limited to per-link guarantees resulting from the use of TCP. In many applications, end-to-end guarantees are required, and often of strong flavor, e.g., to support virtually synchronous, transactional or state-machine replication. Because receivers are assumed passive and cannot cache, forward messages or participate in multiparty protocols, even weak guarantees of these sorts cannot be provided.

4. *Difficult to manage*. It is hard to create and maintain an Internet-scale dissemination structure that would permit any node to serve as a publisher or as a subscriber, for this requires many parties to maintain a common infrastructure, agree on standards, topology and other factors. Any such large-scale infrastructure should respect local autonomy, whereby the owner of a portion of a network can set up policies for local routing, availability of IP multicast, etc.

5. *Inability to use external multicast frameworks*. The standards leave it entirely to the recipients to prepare their communication endpoints for message delivery.

This makes it impossible for a group of recipients to dynamically agree upon a shared IP multicast address, or to construct an overlay multicast within a segment of the network. Yet such techniques are central to achieving high performance and scalability, and can also be used to provide QoS guarantees or to leverage emergent technologies.

3.1.3 Design Principles

To meet the goals formulated in the preceding section, we adopted the following set of design principles:

1. *Programmable nodes*. Senders and recipients should not be limited to sending or receiving. They should be able to perform certain basic operations on data streams, such as forwarding or annotating data with information to be used by other peers, in support of local forwarding policies. The latter must be expressive enough to support protocols used in today's content delivery networks, such as overlay trees, rings, mesh structures, gossip, link multiplexing, or delivery along redundant paths.

2. *External control*. Forwarding policies used by subscribers must be selected and updated in a consistent manner. A node cannot predict a-priori what policy to use, or which other nodes to peer with; it must thus permit an external trusted entity or an agreement protocol to control it: determine the protocol it follows, install rules for message forwarding or filtering, etc.

3. *Channel negotiation*. The creation of communication channels should permit a handshake. A recipient might be requested to, e.g., join an IP multicast address, or subscribe to an external system. The recipient could also make configuration decisions on the basis of the information about the sender. For example, a LAN

domain asked to create a communication endpoint for receiving could select a well-provisioned node as its entry point to handle the anticipated load.

4. *Managed channels*. Communication channels should be modeled as contracts in which receivers have a degree of control over the way the senders are transmitting. In self-organizing systems, reconfiguration triggered by churn is common and communication channels often need to be reopened or updated to adapt to the changing topology, traffic patterns or capacities. For example, a channel that previously requested that a given source transmits messages to one node may notify the source that messages should now be transmitted to some two other nodes.

5. *Hierarchical structure*. The principles listed above should apply to not just individual nodes, but also to entire administrative domains, such as LANs, data centers or corporate networks. This allows the definition and enforcement of Internet-scale forwarding policies, facilitating cooperation among organizations in maintaining the global infrastructure. The way messages are delivered to subscribers across the Internet thus reflects policies defined at various levels (for example, policies "internal" to data centers, and a global policy "across" all data centers).

6. *Isolation and local autonomy*. A degree of a local autonomy of the individual administrative domains should be preserved; such as how messages are forwarded internally, which nodes are used to receive incoming traffic or relay data to other domains, etc. In essence, the internal structure of an administrative domain should be hidden from other domains it is peering with and from the higher layers. Likewise, the details of the subcomponents of a domain should be as opaque to it as possible.

7. *Reusability*. It should be possible to specify a policy for message forwarding or loss recovery in a standard way and post it into an online library of such policies as a contribution to the community. Administrators willing to deploy a given policy

within their administrative domain should be able to do so in a simple way, e.g., by drag-and-drop, within a suitable GUI.

8. *Separation of concerns*. Motivated by the end-to-end principle, we separate implementation of loss recovery and other reliability properties from the unreliable dissemination of messages, as well as from message ordering, security, and subscription management.This decoupling gives our system an elegant structure and a degree of modularity and flexibility unseen in existing architectures.

3.1.4 Our Approach

Our architecture supports structural composition by viewing instances of protocols running among sets of nodes as "objects". For a publish-subscribe topic X, and some subset S of nodes among all the publishers and subscribers to X, we can think of all the messages, state, and in general, any resources and activities pertaining to an instance of a protocol that disseminates events in topic X, as a live object O_S. In particular, if A is the set of all publishers and subscribers to X across the Internet, then O_A represents the "entire" Internet-wide publish-subscribe protocol instance for topic X, disseminating events from any publisher to any subscriber. Given such set of A, we now can partition it into subsets $B_1, B_2, \ldots, B_N$ of nodes residing in some N top-level administrative domains. Now, suppose that within each domain B_i, nodes run a "subset" of the protocol logic locally. For example, all nodes within a given B_i might construct a spanning tree, or perform local loss recovery only with other nodes within B_i (Figure 3.3). Thus, we could think of nodes within each B_i as running a "subprotocol" O_{B_i}, i.e. exchanging a subset of all the messages and performing a subset of all the actions of protocol O_A. In this sense, we can think of object O_A as being "composed of" all objects O_{B_i}. This may continue recursively, leading to a hierarchical decomposition of O_A, down to the level of individual nodes, which could be thought of as "leaf" objects that run "subsets" of the

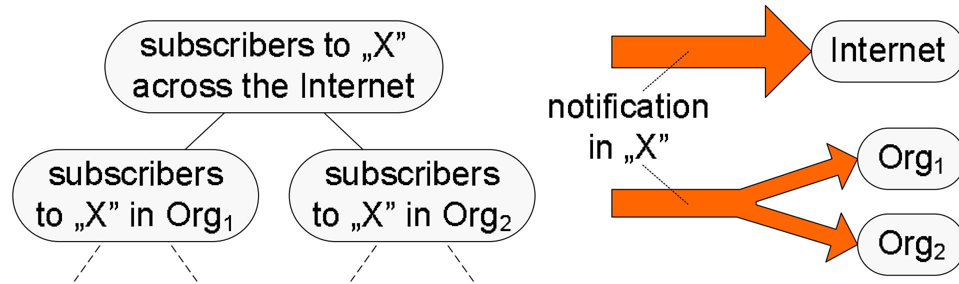

Figure 3.3: A hierarchical decomposition of the set of publishers and subscribers. The set of all nodes is partitioned into subsets residing in top-level administrative domains. These are further recursively sub-partitioned along the boundaries of the sub-domains.

protocol logic limited to local actions, such as interacting with the application.

Hierarchies of this sort have been previously exploited, e.g., in RMTP [248], and in the context of content-based filtering [22]. The underlying principle, implicit in many scalable protocols, is to exploit locality in terms of network topology, latency, or other metrics. Following this principle, sets of nodes, clustered based on proximity or interest, cooperate semi-autonomously in message routing and forwarding, loss recovery, managing membership and subscriptions, failure detection, etc. Each such set is treated as a single cell within a larger infrastructure. A protocol running at a global level connects all cells into a single structure, and scalability arises as in the *divide-and-conquer* principle. Additionally, the cells can locally share workload and amortize dissemination or control overheads, e.g., buffer messages from different sources and locally disseminate such combined bundles, etc. Thus, structural decomposition of the sort just described is already used quite pervasively, albeit usually very closely tied to the protocol logic. Our goals is to provide a generic framework that allows such constructions to be done in a generic manner, and where the management of node hierarchies can be automated.

In the architecture describe here we actually go a step further. Following our principles of isolation and local autonomy, each administrative domain should manage the registration of its own publishers and subscribers internally, and it should be able to decide how to distribute messages among them or how to perform loss recovery according

to its local policy, without regard to the way it is done in other administrative domains and at other levels in the hierarchy. Unlike in most hierarchical systems, where hierarchy and protocol are inseparable, and hence the "sub-protocols" used at all levels of the hierarchy are identical, in our architecture we decouple the creation of the hierarchy from the specific "sub-protocols" used at different levels of the hierarchy and we allow the "sub-protocols" to differ. Thus for example, our architecture permits the creation of a single, global dissemination scheme for a topic that uses different mechanisms to distribute data in different organizations or data centers. Likewise, it permits the creation of a single Internet-scale loss recovery scheme that employs different recovery policies within different administrative domains. Previously, this has only been possible with proxies, which can be costly, and which introduce latency and bottleneck. In this paper, we propose a way to do this efficiently, and in a very generic, flexible manner. The key elements of our approach involve:

1. A distributed scheme for maintaining a hierarchical decomposition of the set of all nodes involved in a protocol in a consistent manner. We propose a new type of a membership service that is decentralized and composed of autonomous elements managing different regions of the network, yet that at all times maintains a level consistency of its distributed data structures that is sufficient to implement protocols such as virtual synchrony. By avoiding collecting all information or routing all events into one location, the structure is able to limit the amount of information and processing in any of its parts to $\Theta(1)$, and can potentially scale without limits. Our membership service forms a "backbone" of the Internet, and plays the role somewhat similar to DNS, but unlike DNS, it is consistent, and its role is not limited to passive delivery of data: the service not only maintains data in a distributed manner, but also maks "decentralized" decisions.

2. A concept of a *protocol agent*, similar to a live object proxy, and an example of

how existing protocols can be modeled as hierarchies of interconnected agents.

3. A mechanism through which the decentralized membership service can construct and consistently maintain hierarchies of protocol agents collaboratively hosted by all publishers and subscribers involved in running the given protocol. While any dissemination, loss recovery, and other protocol-related tasks are performed directly by the protocol agents, and without the involvement of any external infrastructure, the membership service plays the role of a distributed "administrator", an external, God-like entity that provides all protocol members with subsets of a global, agreed-upon "view" of the world.

3.2 Architecture

3.2.1 The Hierarchy of Scopes

Our architecture is organized around the following concepts: *management scope*, *forwarding policy*, *channel* , *filter*, *session*, *recovery protocol*, *recovery domain*, and *protocol agent*.

A *management scope* (or simply *scope*) represents a set of jointly managed nodes. It may include a single node, span over a set of nodes residing within a certain administrative domain, or include nodes clustered based on other criteria, such as common interest. In the extreme, a scope may span across the entire Internet. We do not assume a 1-to-1 correspondence between administrative domains and the scopes defined based on such domains, but that will often be the case. A LAN scope (or just a LAN) will refer to a scope spanning all nodes residing within a LAN. The reader might find it easier to understand our design with such examples in mind.

A scope is not just *any* group of nodes; the assumption that they are *jointly managed* is essential. The existence of a scope is dependent upon the existence of an infrastructure

that maintains its membership and administers it. For a scope that corresponds to a LAN, this could be a server managing all local nodes. In a domain that spans several data centers in an organization, it could be a management infrastructure, with a server in the company headquarters indirectly managing the network via subordinate servers residing in every data center. No such global infrastructure or administrative authority exists for the Internet, but organizations could provide servers to control the Internet scope in support of their own publishers or to manage the distribution of messages in topics of importance to them. Many global scopes, independently managed, can co-exist.

Like administrative domains, scopes form a hierarchy, defined by the relation of *membership*: one scope may declare itself to be a *member* (or a *sub-scope*) of another. If X declares itself to be a member of Y, it means X is either physically or logically a part (subset) of Y. Typically, a scope defined for a sub-domain X of an administrative domain Y will be a member of the scope defined for Y. For example, a node may be a member of a LAN. The LAN may be a member of a data center, which in turn may be a member of a corporate network. A node may also be a member of a scope of an overlay network. For a data center, two scopes may be defined, e.g., *monitoring* and *control* scopes, both covering the data center, with some LANs being a part of one scope, or the other, or both. The corporate scope may be a member of several Internet-wide scopes.

The generality in these definitions allows us to model various special cases, such as clustering of nodes based on interest or other factors. Such clusters, formed, e.g., by a server managing a LAN and based on node subscription patterns, could also be treated as (virtual) scopes, all managed by the same server. Nodes would thus be members of clusters, and clusters (not nodes) would be members of the LAN. As we shall explain below, every such cluster, as a separate scope, could be locally and independently managed. For example, suppose that we are building an event notification system that needs to disseminate events reliably, and is implemented by an unreliable multicast mechanism

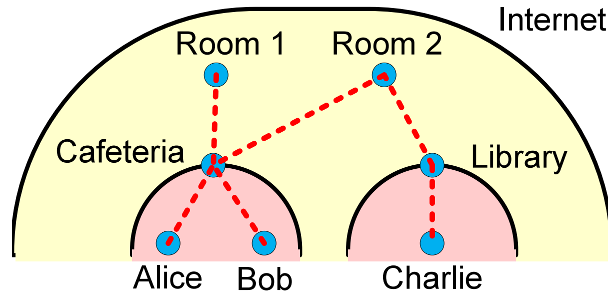

Figure 3.4: An example hierarchy of management scopes in a game. The combined hierarchy for all topics may in general not be a tree.

coupled to a reliability layer that recovers lost packets. In the proposed architecture, different clusters could run different multicast or loss recovery protocols; this technique is used in the platform described in Chapter 4. Thus, if one cluster happens to reside within a LAN that permits the use of IP multicast, it could use that technology, while a different cluster on a network that prohibits IP multicast, or consisting of a large number of nodes across the Internet, could instead form an end-to-end multicast overlay.

The scope hierarchy need not necessarily be a tree (Figure 3.4). There may be many global scopes, or many super-scopes for any given scope. However, a scope decomposes into a tree of sub-scopes, down to the level of nodes. The *span of a scope* X is the set of all nodes at the bottom of the hierarchy of scopes rooted at X. For a given topic X, there always exists a single global scope responsible for it, i.e. such that all subscribers to X reside in the span of X. Publishing a message to a topic is thus always equivalent to delivering it to all subscribers in the span of some global scope, which may be further recursively decomposed into the sets of subscribers in the spans of its sub-scopes.

Suppose that Alice and Bob are sitting with their laptops in a cafeteria, while Charlie is in a library. Both the cafeteria's wireless network and the local network in the library are separately managed administrative domains, and they define their own management scopes. Alice's and Bob's laptops are also scopes, both of which become members of the cafeteria's scope. Now suppose that Alice opens her virtual realities browser and enters a

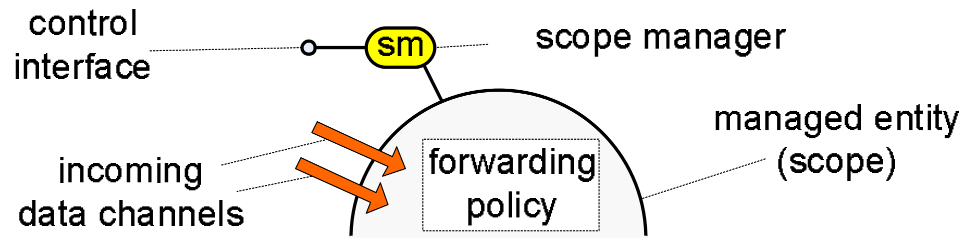

Figure 3.5: A scope is controlled by a separate entity called scope manager. The manager may reside outside of the scope. It exposes a standardized control interface, and may create on demand incoming data channels that serve as entry points for the scope.

virtual place "Room1" in the virtual reality, while Bob and Charlie enter "Room2". Each of the virtual rooms defines a global, internet-wide scope that could be thought of as "the scope of all users in this room, wherever they are". If the networking support for Alice and Bob is still provided by the cafeteria and library wireless networks respectively, when the students enter the rooms, the cafeteria's and library's scopes become members of these global scopes (Figure 3.4).

3.2.2 The Anatomy of a Scope

The infrastructure component administering a scope is referred to as a *scope manager* (SM). A single SM may control multiple scopes. It may be hosted on a single node, or distributed over a set of nodes, and it may reside outside of the scope it controls. It exposes a *control interface*, a web service hosted at a well-known address, to dispatch control requests (e.g., "subscribe") directed to the scopes it controls (Figure 3.5).

A scope maintains communication *channels* for use by other scopes. A *channel* is a mechanism through which a message can be delivered to all those nodes in the span of this scope that subscribed to any of a certain set of topics. In a scope spanning a single node, a channel may be just an address/protocol pair; creating it would mean arranging for a local process to open a socket. In a distributed scope, a channel could be an IP multicast address; creating it would require all local nodes to listen at this address. It

could also be a list of addresses, if messages are to be delivered to all of them, or if addresses are to be used in a random or a round-robin fashion. In an overlay network, for example, a channel could lead to a small set of nodes that forward messages across the entire overlay. In general, a scope that spans a set of nodes can thus be governed by a *forwarding policy* that determines how messages originating within the scope, or arriving through some channel, are disseminated internally within the scope.

Continuing our example, the cafeteria and the library would host the managers of their scopes on dedicated servers, and each of the student laptops would run a local service that serves as a scope manager for the local machine. The library's SM may be on a campus network with IP multicast enabled. When the cafeteria's SM requests a channel from the library's SM, the latter might, e.g., dedicate some machine as the entry point for all messages coming from the cafeteria, instruct it to retransmit these messages to the IP multicast address, and instruct all other laptops to join the IP multicast address. Similarly, the cafeteria's SM might setup a local forwarding tree. In most settings, a scope manager would live on a dedicated server. It is also conceivable to offload, in certain scenarios, parts of the SM's functionality to nodes currently in the scope (e.g., to Alice's and Bob's laptops), but a single "point of contact" for the scope must still exist.

The control interfaces "exposed" by scopes (interfaces exposed by their scope managers) are "accessed" by other scopes (i.e. by their SMs). When interacting, scopes can play one of a few standard roles, corresponding to the three principal interaction patterns: *member-owner*, *sender-receiver*, and *client-host* (see Figure 3.6). A *member* registers with an *owner* to establish a relation of *membership*, i.e. to "declare" itself as a sub-scope of the owner. After such a relationship has been established, the member may then register with the owner as a publisher or subscriber in one or more topics. Depending on the topics for which the member has registered and its role in these topics, it may be requested by the owner to perform relevant actions, e.g., by forwarding messages or

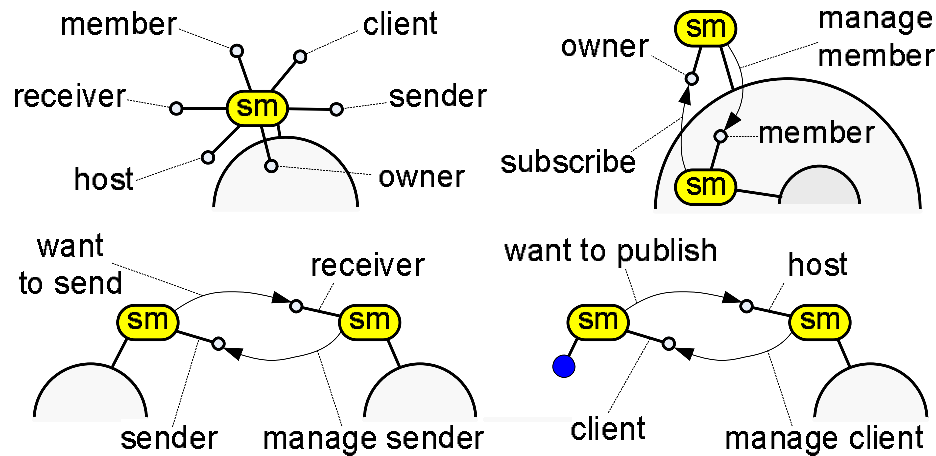

Figure 3.6: Standard roles and patterns of interaction between scopes. These patterns would form the basis of our interoperability standard.

participating in the construction of an overlay structure. At the same time, the owner is responsible for tracking the health of its members, and in particular for detecting failures and performing local reconfiguration.

The *client-host* relationship is similar to member-owner, in that a *client* can register with a *host* as a publisher or as a subscriber. However, unlike the member-owner relationship, which involves a mutual commitment, the client-host relationship is more casual, in the sense that the host cannot rely on the client to perform any tasks for the system, or even to notify the host when it leaves, and similarly, the client cannot rely on the host to provide it with the same quality of service and reliability properties as the regular members. Thus for example, while a member can form a part of a forwarding tree, a client will not; the latter will also typically get weaker reliability guarantees, because the protocols run by the scope members will not be prevented from making progress when transient clients are unreachable. The same applies to publishers. A long-term publisher that forms a part of a corporate infrastructure will register as a member, but handheld devices roaming on a wireless network will usually register as clients.

The *sender-receiver* relationship is similar to a publisher registering as a client or member, in that the *sender* registers with the *receiver* to send data. However, whereas

a member registers with its owner to publish to the set of all subscribers in a topic, including nodes inside as well as nodes outside of the owner, a sender will register with a receiver to establish a communication channel between the two to disseminate messages only within the scope of the receiver. When a publisher registers as a member with an owner scope that is not the global scope for the given topic, the owner may itself be forced to subscribe with its super-scope. The sender-receiver relationship is horizontal; no cascading subscriptions take place. On the other hand, while a member scope never exchanges data with the owner (instead, it is "told" by the owner to form part of a dissemination structure that the owner controls), the sender-receiver relationship serves exactly this purpose; the two parties involved in the latter will negotiate protocols, addresses, transmission rates, etc.

The reader should recognize in our construction the design principles we articulated earlier. Scopes, whether individual nodes, LANs or overlays, are externally controlled using the control interfaces exposed by SMs, may be programmed with policies that govern the way messages are distributed internally, forwarded to other scopes, etc., and transmit messages via managed communication channels, established through a dialogue between a pair of SMs, and dynamically reconfigured.

3.2.3 Hierarchical Composition of Policies

Following our hierarchical approach, we now introduce a structure, in which forwarding policies defined at various levels are merged into a hierarchical, Internet-wide overlay.

Each scope is configured with a policy dictating how messages are forwarded among its members, based on criteria such as the topic the message was published on, or where the message *locally originated*. We will say that a message "locally originated at X" within scope Y if X is a member of Y such that either it contains a publisher, or that contains a node that received the message from outside of X. Depending on the for-

warding policy at the local or at its parent, messages may locally originate in multiple locations. A policy defined for a particular scope X is always defined at the granularity of X's members (not individual nodes). The way a given sub-scope Y of X delivers messages internally is a decision made autonomously by Y, and opaque to X. Similarly, while X's policy may specify that Y should forward messages to a certain scope Z, it is up to Y's local forwarding policy to determine how to perform this task, e.g., which of its members should be selected to establish channels to Z.

For example, a policy governing a global scope might determine how messages in topic T, originating in a corporate network X, are forwarded between the various organizations. A policy of a scope of the given organization's network might determine how to forward messages among its data centers, and so forth. A global policy may request that organization X forward messages in topic T to organizations Y and Z. A policy governing X may then determine that to distribute messages in X, they must be sent to LAN_1, which will forward them to LAN_2. The same policy might also specify which LANs within X should forward to Y and Z, and finally, the policies of these LANs will delegate these forwarding tasks to individual nodes they own.

When all the policies defined at all the involved scopes are combined together, they yield a global *forwarding structure* that completely determines the way messages are disseminated (Figure 3.7). Although in the examples above, the forwarding policies are simply graphs of connections, i.e. each message is always forwarded along every channel, in general this does not have to be the case. A channel could be constrained with a *filter* that decides, on a per-message basis, whether to forward or not, and may optionally tag the message with custom attributes (more details are in Section 3.2.4). This allows us to express many popular techniques, e.g., using redundant paths, multiplexing between dissemination trees, etc.

Every scope manager maintains a mapping from topics to their forwarding policies.

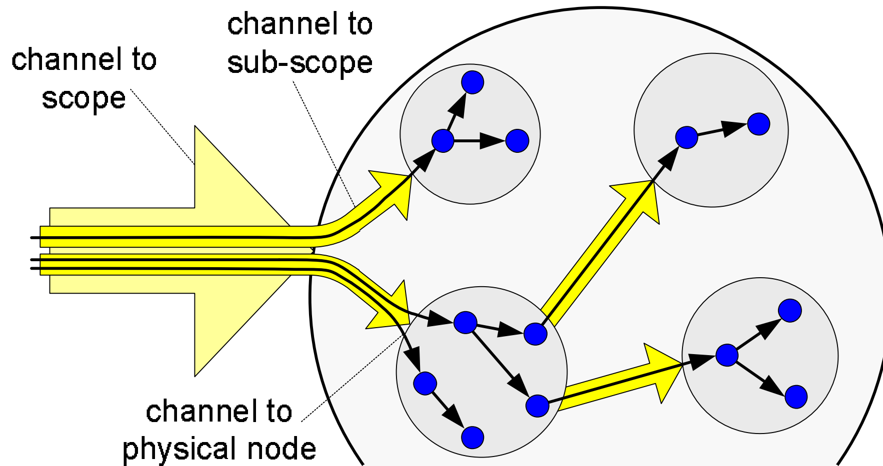

Figure 3.7: Channels created in support of policies defined at different levels. Despite the fact that the individual policies are defined independently from each other, our approach ensures that a single, globally consistent structure always emerges as a result.

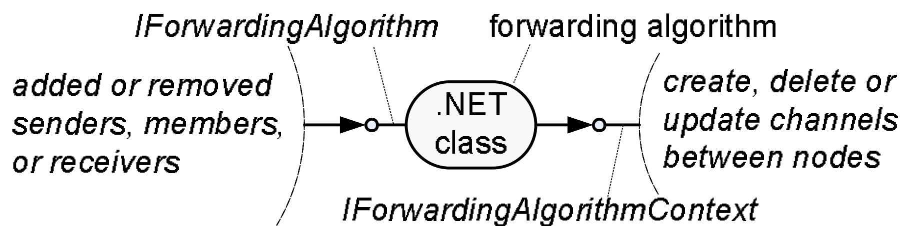

Figure 3.8: A forwarding policy as a code snippet. The code runs in an abstract context and may be packaged as a live object. The interface between the policy and its context can be defined, e.g., in WSDL [81], and compiled down, e.g., to a Java or .NET interface

A forwarding policy is defined as an object that lives within an *abstract context*, and that exposes a certain fixed set of standardized *events* to which members must be prepared to react, and *operations* and *attributes* the policy may inspect. A scope manager might thus be thought of as a *container* for objects representing *policies*. The interfaces that the container and the policies expose to each other and interact with are standardized, and may include, for example, an event informing the policy that a new member has been added to the set of members locally subscribed to the topic, or an operation exposed by the container that allows the policy to request a member to establish a channel to another member and instantiate a given type of filter (Figure 3.8). A scope manger thus provides a *runtime environment* in which forwarding policies can be hosted. This allows

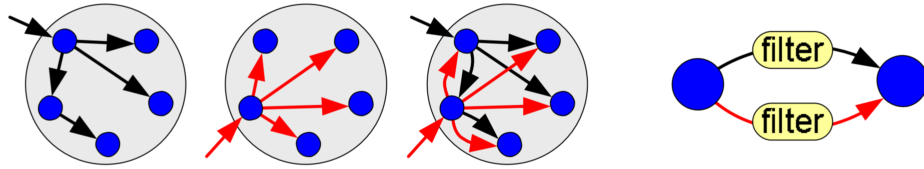

Figure 3.9: Forwarding graphs for different topics are superimposed. Two members of the scope could be linked by multiple channels created by the policy in support of dif-ferent topics, each with a different filter.

policies to be defined in a standard way, independent not only of the platform, but also, to a degree, independent of the type of the administrative domain. Naturally, different types of physical environments might offer different classes of resources, e.g., certain environments might support IP multicast, while others might not. Still, one can abstract the description and management of communication resources through a standardized interface, and decouple policies from the physical assets they might be instantiated over.

For example, one can imagine a policy that uses some novel mesh-like forwarding structure with a sophisticated adaptive flow and rate control algorithm. Our architecture would allow that policy to be deployed, without any modifications, in the context of a LAN, data center, corporate network, or a global scope. In effect we have separated the policy from the details of how it should be implemented in a particular domain. Policies may be implemented in any language, expose and consume WSDL-like APIs, stored online in *protocol libraries*, downloaded as needed, and executed in a secure manner.

Graphs of connections for different topics, generated by their respective policies, are superimposed by the SM (Figure 3.9). SM maintains an aggregate structure of all channels and filters created by its forwarding policies, and issues requests to the SMs of its members to create channels, instantiate filters, etc. If multiple policies request channels between the same pair of nodes, the SM will not create multiple channels, but rather a single channel, for multiple topics simultaneously. To avoid the situation where every member talks to every other member, the SM may use a single forwarding policy for sets of topics, to ensure that channels created for different topics overlap. The system

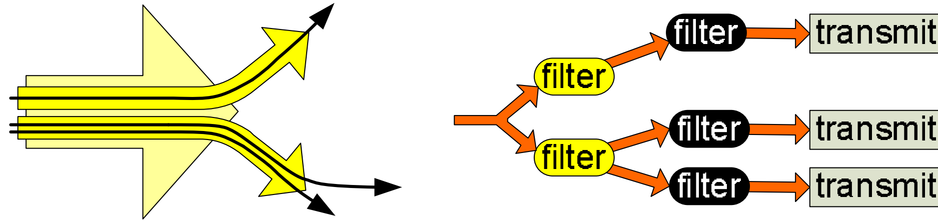

Figure 3.10: A channel split into sub-channels and a corresponding filter tree. Most of these filters would be extremely simple and could be implemented at a hardware level.

described in Chapter 4 could take advantage of this property.

3.2.4 Communication Channels

Consider a node X, which is a member of a scope Y that, based on a forwarding policy at Y, has been requested to create a communication channel to scope Z to forward messages in topic T. Following the protocol, X asks the SM of Z for the specification of the channel to Z that should be used for messages in topic T. The SM of Z might respond with an address/protocol pair that X should use to send over this channel. Alternatively, a forwarding policy defined for T at scope Z may dictate that, in order to send to Z in topic T, scope X should establish channels to members A and B of Z, constrained with filters α and β. After X learns this from the SM of Z, it contacts SMs of A and B for further details. Notice that the channel to Z decomposes into sub-channels to A and B through a policy at a target scope Z. This procedure will continue hierarchically, until the point when X is left with a tree with filters in the internal nodes and address and protocol pairs at the leaves (Figure 3.10). Now, to send a message along a channel constructed in this way, X executes filters, starting from the root, to determine recursively which sub-channels to use, proceeding until it is left with just a list of address and protocol pairs, and then transmits the message. Filters will usually be simple, such as modulo-n; hence X can perform this procedure very efficiently. Indeed, with network cards becoming increasingly powerful and easily programmable, such functionality may even

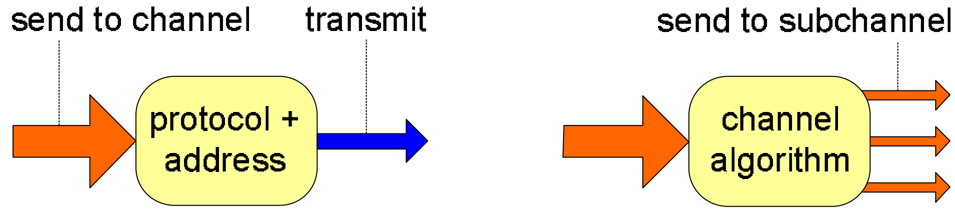

Figure 3.11: A channel may specify a protocol or decompose into sub-channels. In the latter case, an additional policy determines how messages are multiplexed across the sub-channels of a channel to achieve the desired type of load balancing or redundancy.

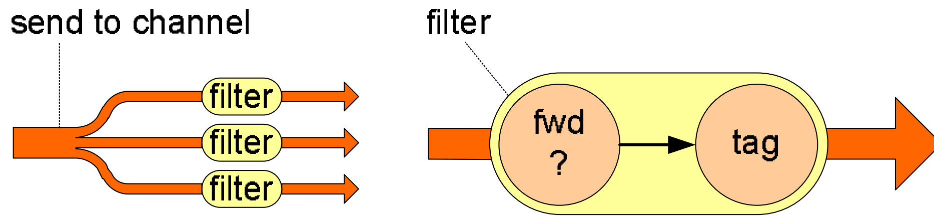

Figure 3.12: Channel policies are realized by per-subchannel filters. At runtime, each filter decides whether to forward and might optionally tag the message in transit.

be offloaded to hardware, for example using the approach proposed in [307].

Accordingly, to support the hierarchical composition of policies described in the preceding section, we define a channel as one of the following: an address/protocol pair, a reference to an external multicast mechanism, or a set of sub-channels accompanied by filters. In the latter case, the filters jointly implement a multiplexing scheme that determines which sub-channels to use for sending, on a per-message basis (Figure 3.11, Figure 3.12)). Consider now the situation where scope X, spanning a *set of nodes*, has been requested to create a channel to scope Y. Through a dialogue with Y and its sub-scopes, X can obtain a detailed channel definition, but unlike in prior examples, X now spans a set of nodes, and as such, it cannot *execute* filters or *send* messages. To address this issue, we now propose two simple, generic techniques: *delegation* and *replication* (Figure 3.13). Both of them rely on the fact that if X receives messages in a topic T, then some of its members, Z, must also receive them (for otherwise X would not declare itself as a subscriber and it would not be made part of the forwarding structure for topic

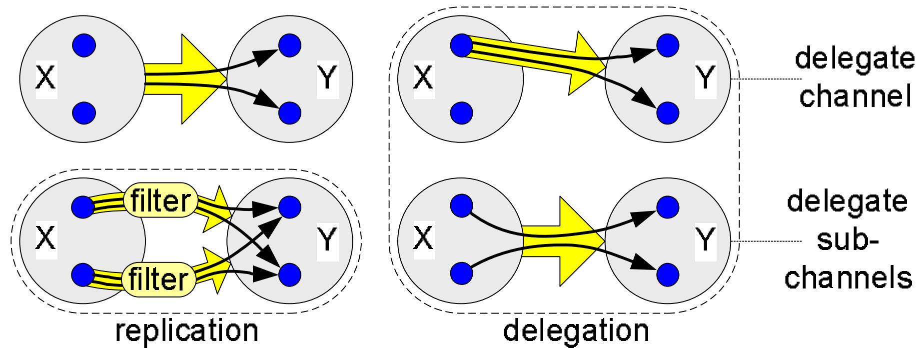

Figure 3.13: A distributed scope delegates or replicates channels to its members. In the case of delegation, full responsibility for either the entire channel, or its sub-channels, is passed on to a sub-scope. In the case of replication, a channel or sub-channel is replicated, each replica is constrained with a modulo-k filter, an the different replicas are then individually delegated to different sub-scopes.

T by X's super-scope). In case of *delegation*, X requests such a sub-scope Z to create the channel on behalf of X, essentially "delegating" the entire channel to its member. The problem can be recursively delegated, down to the level where a single physical node is requested to create the channel, and to forward messages it receives along the channel. A more sophisticated use of delegation would be for X to delegate sub-channels. In such case, X would first contact Y to obtain the list of sub-channels and the corresponding filters, and for each of these sub-channels, delegate it to its sub-scopes. In any case, X delegates the responsibility for forwarding over a channel to its sub-scopes.

Our approach is flexible enough to support even more sophisticated policies. An example of one such policy is a *replication* strategy, outlined here. In this scheme, scope X could request that n of its sub-scopes create the needed channel, constraining each with a modulo-n filter based on a message sequence number. Hence, while each of the sub-scopes would create the same channel on behalf of its parent scope (hence the name "replication"), sub-scope k would only forward messages with numbers m such that $m \bmod n = k$. By doing this, X effectively implements a round-robin policy of the sort proposed in protocols such as SplitStream [69]. Although all sub-scopes would create the same channel, the round-robin filtering policy would ensure that every

message is forwarded only by one of them. This technique could be useful, e.g., in the cases where the volume of data in a topic is so high that delegation to a single sub-scope is simply not feasible.

Our point is not that this particular way of decomposing functionality should be required of all protocols, but rather that for an architecture to be powerful enough and flexible enough to be used with today's cutting-edge protocols and to support state-of-the-art technologies, it needs to be flexible enough to accommodate even these kinds of "fancy" behaviors. Our proposed architecture can do so. The existing standards proposals, in contrast, are incredibly constraining. Each only accommodates a single rather narrowly conceived style of event notification system.

3.2.5 Constructing the Dissemination Structure

A detailed discussion of how forwarding policies can be defined and translated to filter networks is beyond the scope of this paper. We describe here just one simple, yet fairly expressive scheme.

Suppose that the forwarding policies in all scopes define forwarding trees on a per-topic basis and possibly also depending on the location at which the message locally originated. The technique described here implicitly assumes that messages do not locally originate in more than one location, i.e. that the forwarding policy at each level is a tree, not a mesh. The scheme may be extended to cover more general cases with redundant paths and randomized policies, but we omit it here, for it would unnecessarily complicate our example.

A scope manager thus maintains a graph, in which the scope members are linked by edges, labeled with constraints such as β:X, μ:Y, ..., meaning that a message is forwarded along the edge if it was sent in topic β and locally originates at X, or if it was sent in topic μ and locally originates at Y, and so on. We shall now describe, using a

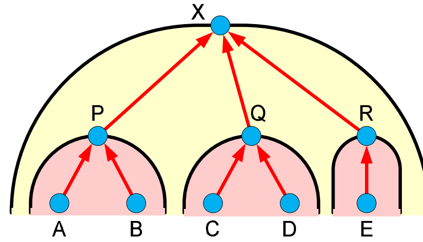

Figure 3.14: An example hierarchy of scopes with cascading subscriptions. Subscribe requests issued by student laptops cause cascading requests that propagate up to the root scope *X* that represents the entire campus.

simple example, how a structure of channels is established based on scope policies, and how filters are instantiated. We shall then explain how messages are routed.

Consider the structure of scopes depicted on Figure 3.14. Here *A*, *B*, *C*, *D*, and *E* are student laptops. *P*, *Q*, and *R* are three departments on a campus, with independent networks, hence they are separate scopes. *X* represents the entire campus. All students subscribe to topic *T*. Topic *T* is local to the campus, and *X* serves as its root. The scopes first register with each other. Then, the laptops send requests to subscribe for topics to *P*, *Q* and *R*. Laptop *A* requests the publisher role, all others request to be subscribers. None of *P*, *Q* or *R* are roots for the topic, hence they themselves subscribe for topic *T* with *X*, in a cascading manner. Now, all scopes involved create objects that represent local forwarding policies for topic *T*, and feed these objects with the "new member" events. The policy at *P* for messages in *T* originating at *A* creates a channel from *A* to *B*. Similarly, the policy at *X* for messages in *T* originating at *P* creates channels *P* to *Q* and *Q* to *R* (Figure 3.15). Each channel has a label of the form "*X*-*Y*, *T*:*Z*", meaning that it is a channel from *X* to *Y*, for messages in *T* originating at *Z*. Note that no channels have been created in scope *Q*. Until now, *Q* is not aware of any message sources because neither *C* nor *D* is a publisher, and because no other scope has so far requested a channel to *Q*, hence there is no need to forward anything. Channels are now delegated to individual nodes, as described in Section 3.2.4. *P* delegates its channel to *B*, and *Q* delegates to *D* (Figure 3.16 delegated channels are in blue). Channel labels

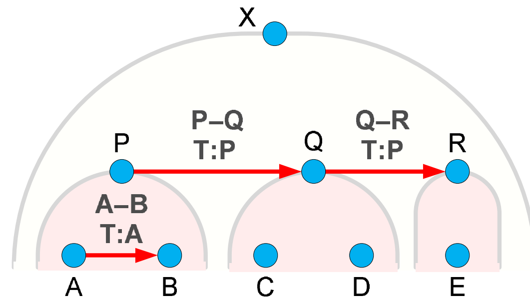

Figure 3.15: Channels created by the policies based on subscriptions.

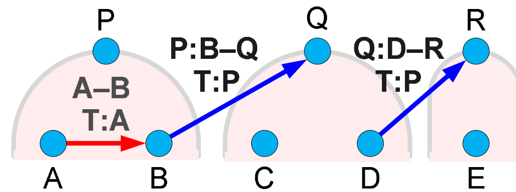

Figure 3.16: Channels are delegated.

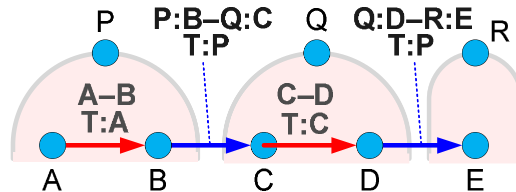

Figure 3.17: *B* and *D* contact *Q* and *R* to create channels. *Q* and *R* select *C* and *E* as entry points. *Q* now has a local source and creates its own local channel, from *C* do *D*.

are extended to reflect the fact that they have been delegated, e.g., *P*-*Q* becomes *P*:*B*-*Q*, which means that *P* has delegated its source endpoint to its member *B*, etc. At this point, *B* and *D* contact the destinations *Q* and *R*, and request the channels to be created. *Q* and *R* select *C* and *E* as their *entry points* for these channels. Now *Q* also has a local source of messages (node *C*, the entry point), so now it also creates an instance of a forwarding policy, which determines that messages in *T*, locally originating at *C*, are forwarded to *D* (Figure 3.17). The entire forwarding structure is complete.

A message in transit is a tuple of the form (T, k, r), where T is the topic, k is the identifier that may include the source name, one or more sequence numbers, etc., and r is a *routing record*. The routing record is an object of the form ((X_K-Y_K), (X_{K-1}-Y_{K-1}), . . .,

$(X_1$-$Y_1))$, in which every pair of elements (X_i, Y_i) represents the state of dissemination in one scope; X_i is the member of the scope that the message locally originated from and Y_i is the member of the scope that the message is moving inside of or that it is entering. Pair (X_1, Y_1) represents individual nodes, and for each i, scopes X_i and Y_i are a level below X_{i+1} and Y_{i+1}, respectively, and Y_i is always a member of Y_{i+1}. This list of entries does not need to "extend" all the way up to the root. If entries at some level are missing, they are filled up when the message jumps across scopes, as explained below.

When a message arrives at a node, the node iterates over all of its outgoing channels, and matches channel filters against the routing record. Message $(T, k, ((X_K$-$Y_K), (X_{K-1}$-$Y_{K-1}), \ldots, (X_1, Y_1)))$ matches channel $(P_L{:}P_{L-1}{:}\ldots{:}P_1$-$Q_L{:}Q_{L-1}{:}\ldots{:}Q_1, T{:}R)$ when $(K \geq L \wedge X_L = R \vee K < L \wedge P_L = R)$ holds. This condition has two parts. If $K \geq L$, then in the scope in which the channel endpoints P_L, Q_L and R are members, the message originated at X_L and is currently at Y_L. According to our rules, the message should be forwarded iff $X_L = R$ (and $Y_L = P_L$, but this is always true). If $K < L$, then the routing record does not carry any state for the scope at which P_L, Q_L and R are members. This means the message must have originated in this scope (the recovery record is filled up when the message is forwarded, as explained below), hence the condition $P_L = R$. Note that there might be several channels originating at the node; the message is forwarded across each one it matches. Now, when the message is forwarded, its routing record is modified as follows. First, the record is extended. If $K < L$, then for each i such that $i < K$ and $i \leq L$, we set $X_i = P_i$ and $Y_i = P_i$. Then, the details internal to the scope that the channel is leaving are replaced with the details internal to the scope the channel is entering, so for each i such that $i < L$, we set $X_i = Q_i$ and $Y_i = Q_i$. Finally, the current scope at the level at which the channel is defined is updated, we set $Y_L = Q_L$. The original value of X_L, once set, is not changed until the message is forwarded out of the scope of which X_L is the member. Entries for $i > L$ also remain unchanged.

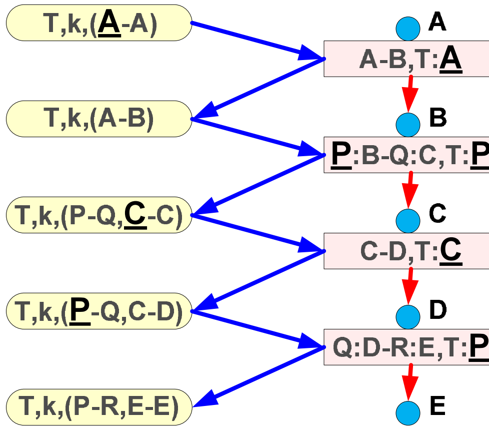

Figure 3.18: The flow of messages (rounded rectangles, left), and the channels (square rectangles, right) in the scenario of Figure 3.17. Elements compared against each other are shown as black, bold, and underlined.

The flow of messages and matching in the example of Figure 3.17 is shown on (Figure 3.18). When created on the publisher A, the message initially has a routing record A-A, since we know that it originated locally at A and that it is still at A. No entries are included at this point for routing in the scopes above A. There is no need for doing so since no routing has been done so far at those higher levels, so there is no state to maintain. Now the routing record is compared against channel A-B, T:A. The local origin A does match the channel's constraint A, so the message is forwarded along A-B, with a new record A-B to reflect the fact that it is now entering B. While matching this record to channel P:B-Q:C at B, we find that the record lacks routing information for the scope at which this channel is defined. This means that the message must have originated at the channel's source endpoint (which we know is P from the channel's description). We find that the channel's constraint P is the same as the channel's source endpoint P, so we again forward the message. While doing so, we extend its routing record with a new entry P-Q to record the fact that we made progress at this higher level. We also replace information local to P (the entry A-B) with new information local to Q (the new entry C-C). Forwarding at C and D is similar to how it worked on A and B.

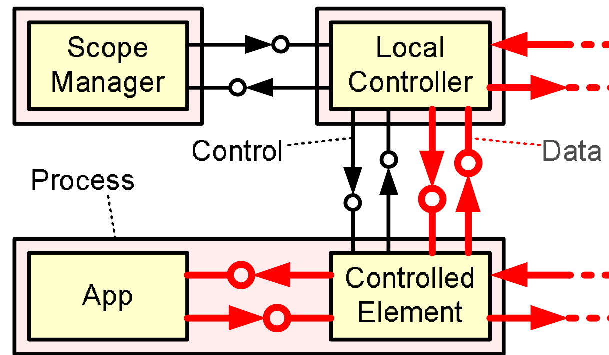

Figure 3.19: The architecture of a leaf scope in the most general scenario, with a local *scope manager*, a thin *controlled element* linked to the application, and with a *local controller* to handle peer-to-peer aspects, e.g., forwarding.

3.2.6 Local Architecture of a Dissemination Scope

So far we focused in our description on peer-to-peer aspects of the dissemination framework, but we have not described how applications use this infrastructure, and how leaf scopes are internally organized. Of a number of possible architectures of the leaf scopes, the most general one involves the following components: *scope manager*, *local controller*, and *controlled element* (Figure 3.19). The *controlled element* is a very thin layer that resides in the application process. Its purpose is to serve as a hookup to the application process that allows for controlling the way the application communicates over the network. As such, it serves two principal purposes: (a) passing to the local controller requests to subscribe/unsubscribe, and (b) opening send and receive sockets in the process per request from the controller. The controlled element does not forward messages, nor does it include any other peer-to-peer functionality, and it is not a part of the management network. This simplicity allows it to be easily incorporated into legacy applications. It can be implemented in any way, for as long as it exposes a *control endpoint* (Figure 3.19 , thin, black), a standardized web service interface, and as long as it can create *communication endpoints* (thick, red) to receive or connect to remote endpoints to send. The local controller implements all the peer-to-peer functionality such

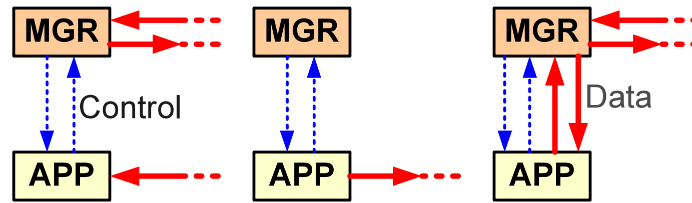

Figure 3.20: Example architectures with the scope manager and the local controller merged into a single local system "daemon".

as forwarding, and hosts such elements as channels or filters. It may also create communication endpoints, and may send data to or receive it from the controlled element. The controller is not a part of the management network, and it does not interact with any scope mangers besides the local one. These interactions are the job of the local SM, which, on the other hand, never sends or receives itself any messages.

In general, the three components can live in separate processes, or even physically on different machines, and may communicate over local sockets or over the network. However, in the typical scenario, the scope manager and local controllers are located in a single process ("manager"). The manager runs as a system service and may control multiple applications, either local or running on subordinate devices managed by the local node, through the control elements embedded in these applications. Within this scenario, we can distinguish three basic sub-scenarios, or three patterns of usage (Figure 3.20, depending on whether applications participate in sending or receiving directly, or only through the manager.

In the first scenario, the applications only receive data, directly from the network (Figure 3.20, left). When the manager is requested to create an incoming channel to the scope, it may either arrange for all applications to open the same socket to receive messages directly from the network (if the applications are all hosted one the same machine), or it may make them all subscribe to the same IP multicast address (if they are running on multiple subordinate devices), or it may have them create multiple endpoints, in which case the definition of the local channel endpoint will include a set of addresses

rather than just one. The manager does not sit on the critical path; hence we avoid bottleneck and latency. If the local scope is required to forward data to other scopes, the manager also creates an endpoint (opens the same socket, or extends the receive channel definition to include its own address), to receive the data that needs to be forwarded.

In our example, the library's server might act as a leaf scope, and students' laptops might act as subordinate devices that do not host their own local scope managers and do not forward messages. Applications on the laptops would have the embedded controlled elements that communicate with the local controller on the library server via a standard web interface. When students subscribe to a local topic, such as an online lecture transmitted by another department, the server chooses an IP multicast address and has all the laptops subscribe to it. The data arriving on the local network is received by all devices, without using a proxy. If the library needs to forward messages, the server also subscribes to the IP multicast address and creates all the required channels and filters.

In the second scenario, the applications act as publishers (Figure 3.20, center). There is no need to forward data, hence the manager does not create any send or receive channels. In order to support this scenario, the controlled element must allow transmitting data to multiple IP addresses; embed various headers provided by the local controller, etc. There can be different "classes" of controlled elements, depending on what functionality they provide; this scenario might be feasible only for some such classes. This scenario avoids proxies, thus it could be useful, e.g., in streaming systems.

In the third scenario, the applications communicate only with the local controller, which acts as a proxy (Figure 3.20, right). Unlike in the first two scenarios, this introduces a bottleneck, but since the controlled elements do not need to support any external protocols or to be able to embed or strip any external headers, this scenario is always feasible. This scenario could also be used to interoperate with other eventing standards, such as WS-Eventing or WS-Notification. Here, the manager could act as the publisher

or a proxy from the point of view of the application, and provide or consume messages in the format defined by all those other standards, while using our general infrastructure “under the hood”. And similarly, for high-performance applications that reside on the server, an efficient implementation is possible using shared memory as a communication channel between the manager and the applications, and permitting applications to deserialize the received data directly from the manager’s receive buffers, without requiring extra copy or marshaling across domains.

3.2.7 Sessions

We now shift focus to consider architectural implications of reliability protocols. Protocols that provide stronger reliability guarantees traditionally express them in terms of what we shall call *epochs*, corresponding to what in group communication systems are called *membership views*. In group communication systems, the lifetime of a topic (also referred to as a *group*) is divided into a sequence of *views*. Whenever the set of topic subscribers changes after a “subscribe” or an “unsubscribe” request, or a failure, a new view is created. In group communication systems, the corresponding event initiates a new epoch. Subscribers are notified of the beginnings or endings of epochs, and of the membership of the topic for each epoch. One then defines consistency in terms of which messages can be delivered to which subscribers and at what time relative to epoch boundaries. The set of subscribers during a given epoch is always fixed.

Whereas group communication views are often defined using fairly elaborate models (such as virtual synchrony, a model used in some of our past research, or consensus, the model used in Lamport’s Paxos protocol suite), the architectural standard proposed here needs to be flexible enough to cover a range of reliability protocols, include many that have very weak notions of views. For example, simple protocols, such as SRM or RMTP, do not provide any guarantees of consistent membership views for topics.

In developing our architectural proposal, we found that even for protocols such as these two, in which properties are not defined in terms of epochs, epochs can still be a very useful, if not a universal, necessary concept. In a dynamic system, configuration changes, especially those resulting from crashes, usually require reconfiguration or cleanup, e.g., to rebuild distributed structures, release resources, or cancel activities that are no longer necessary. Most simple protocols lack the notion of an epoch because they do not take such factors into account and do not support reconfiguration. Others do address some of these kinds of issues, but without treating them in a systematic manner. By reformulating such mechanisms in terms of epochs, we can standardize a whole family of behaviors, making it easier to talk about different protocols using common language, to compare protocols, and to design applications that can potentially run over any of a number of protocols, with the actual binding made on the basis of runtime information, policy, or other considerations.

Our design includes two epoch-like concepts: *sessions*, which are global to the entire topic, across the Internet, are shared by different frameworks (reliability, ordering, etc.), and which we discuss in this section, and *local views*, which are local to scopes, and which are discussed in Section 3.2.10.

A *session* is a generalization of an epoch. In our system, sessions are used primarily as means of reconfiguring the topic to alter its security or reliability properties, or for other administrative changes that must be performed online, while the system is running.

The lifetime of any given topic is always divided into a sequence of sessions. However, session changes may be unrelated to membership changes of the topic. Since introducing a new session involves a global reconfiguration, as described further, it is infeasible for an Internet-scale system to introduce a new session, and disseminate membership views, every time a node joins or leaves. Instead, such events are handled locally, in the scopes in which they occur, without creating a new, global, Internet-wide epoch.

Session numbers are assigned globally, for consistency. As explained before, for a given topic, a single global scope ("root") always exists such that all subscribers to that topic reside within its span. Although, as we shall explain, dissemination, reliability and other frameworks may use different scope hierarchies, the root scope is always the same and all the dissemination, reliability, and other aspects of it are normally managed by a single scope manager. This top-level scope manager maintains the topic's metadata; it is also responsible for assigning and updating session numbers. Note that local topics, e.g., internal to an organization, may be rooted locally, e.g., in the headquarters, and managed by a local SM, much in a way local newsgroups are managed locally. Accordingly, for such topics, sessions are managed by a local server (internal to the organization).

To conclude, we explain how sessions impact the behavior of publishers and subscribers. After registering, a publisher waits for the SM to notify it of the session number to use for a particular topic. A publisher is also notified of changes to the session number for topics it registered with. All published messages are tagged with the most recent session number, so that whenever a new session is started for a topic, within a short period of time no further messages will be sent in the previous session. Old sessions eventually quiesce as receivers deliver messages and the system completes flushing, cleanup and other reliability mechanisms used by the particular protocol. Similarly, after subscribing to a topic, a node does not process messages tagged as committed to session k until it is explicitly notified that it should receive messages in that session. Later, after session $k + 1$ starts, all subscribers are notified that session k is entering a *flushing* phase (this term originates in *virtual synchrony* protocols, but similar mechanisms are common in many reliable protocols; a protocol lacking a flush mechanism simply ignores such notifications). Eventually, subscribers report that they have completed flushing and a global decision is made to cease any activity and *cleanup* all resources pertaining to session k, thus completing the transition.

3.2.8 Incorporating Reliability and Other Strong Properties

As mentioned earlier, we rooted our design in the principle of separation of concerns; hence, we implement tasks such as reliability, ordering, security or scope management independently from dissemination. In Section 3.2.6, we explained how the management and the dissemination infrastructures interact in our system. Aspects of a protocol instance running for the given topic that are related to stronger properties, such as reliability, security and ordering, are decoupled from dissemination, and implemented by separate frameworks. We will refer to those frameworks as frameworks of different *flavors*. Each of these frameworks is hierarchically decomposed, much in the same manner as it was the case for dissemination, and includes the three base components listed earlier: the *controlled element* that lives in the application processes and implements only base functionality, related to sending or receiving from the applications, but none of the peer-to-peer or management aspects; the *local controller* that may live outside of the application process, and where all the peer-to-peer aspects are implemented; and the *scope manager* that implements the interactions with other scope managers, but that is not involved in any activities related to the data flows, such as forwarding, calculating recovery state, managing private keys, ordering messages, etc.

In general, frameworks of *dissemination*, *reliability*, *security*, *ordering* and all other flavors, each have a logically distinct hierarchy of differently-flavored scopes, and a distinct network of scope managers. For example, reliability scopes isolate and encapsulate the local aspects related to reliability, such as loss recovery, and hide their internal details from other scopes, just like dissemination scopes manage local dissemination and hide the local aspects of message delivery. Often, the different scopes would physically overlap; this will normally be the case, e.g., with scopes local to a node, where a single local service would act as a scope manager for scopes of all flavors. The same would typically be the case for the servers that control certain types of administrative domains,

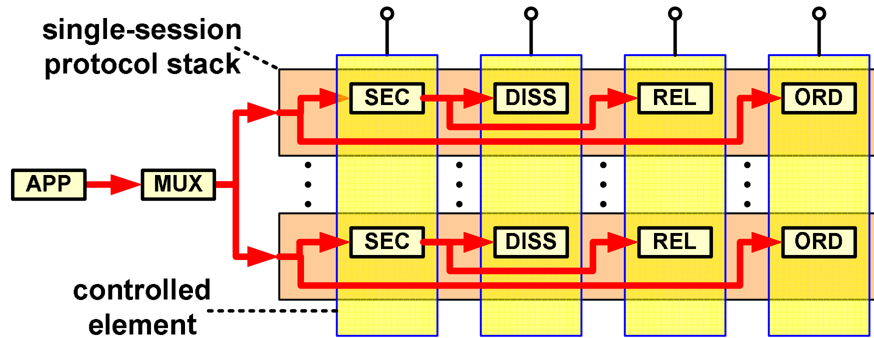

Figure 3.21: Internal architecture of the application process.

such as a departmental LAN, a wireless network in a cafeteria, a data center, a corporate network, or a computational cluster. Scopes of all flavors will often physically overlap, and may be managed by a single SM.

Irrespective of whether components of different "flavors" overlap, or are physically hosted on the same machine or in the same process, the frameworks always logically converge in the application, where local actions performed by elements of differently-flavored frameworks are synchronized. For example, a message flow diagram on Figure 3.21) shows the coordination between controlled elements of the security (SEC), dissemination (DISS), reliability (REL), and ordering (ORD) frameworks. Each session maintains a separate protocol stack containing four layers corresponding to the four different flavors (although in general, different sessions might differ in the properties provided, and could involve different sets of flavors). An application interacts with a multiplexer (MUX), which assigns messages to individual sessions, and forwards them to the respective protocol stacks (rows on Figure 3.21). Elements of the same flavor across all protocol stacks are owned and controlled by their respective controlled elements (columns on Figure 3.21), each of which exposes a standard interface required for interaction with its corresponding local controller.

Now, when the application sends a message, it is first assigned to a session by the multiplexer, and assigned a local sequence number within the session. It is then passed

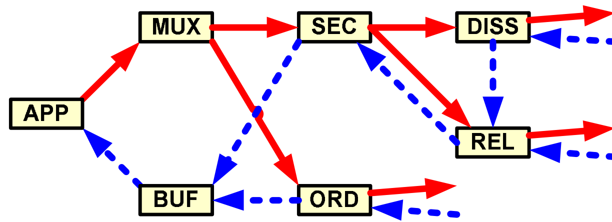

Figure 3.22: Processing messages on the send path (red and solid lines) and on the receive path (blue and dotted lines).

to the appropriate per-session protocol stack, simultaneously to the layers that handle security and ordering. Each of these processes the message independently and concurrently; the layer may encrypt and sign it, if necessary, and then passes it further to the dissemination layer for transmission, and independently, to the reliability layer to place it in a local cache for the purpose of retransmission, forwarding, etc., and to update the local structures to record the fact that the message was created.

At the same time, the ordering layer, also working in parallel with the dissemination and reliability layers, records the presence of the message in its own structures, which are used later by the ordering infrastructure to generate *ordering requests*, small records of the form (p, t, k, S) that contain publisher identifier p , topic identifier t, and session identifier k, and a set S of sequence numbers of messages transmitted by the publisher. Ordering requests are satisfied by generating *orderings*, records containing entries of the form (p, t, k, S, x, O) that map the specified set of messages to numbers O within a certain sequence x that is totally ordered within a topic, across topics, or using whatever ordering scheme is in use. Collecting ordering requests, generating orderings, and delivering orderings to the interested parties is handled by the ordering framework, and entirely independently of dissemination and reliability.

On the receive path, the process would look similar (Figure 3.22). Messages may arrive either through the dissemination framework, in the normal case, or via the reliability framework if they were initially lost and have been later recovered. Messages

that arrive from the dissemination framework are routed via the reliability sub-layer, so that they are registered, can be cached, or so that delivery can be suppressed. When ordering arrives from the ordering framework, messages can be decrypted, placed in a buffer (BUF), and delivered in the appropriate order to the application.

The exact manner in which the subtasks performed by the four layers are synchronized may vary. On the send path, messages may not be transmitted until the entire protocol stack for a given session can be assembled, i.e. if the dissemination framework learned of a new session, but the reliability framework has not, the transmission might be postponed until the information about the new session propagates across the reliability framework as well, to avoid problems stemming from such misalignments. On the receive path, the decryption of the message might be postponed until the ordering is known, to avoid maintaining two copies of the message in memory, one for the purpose of loss recovery (encrypted) and one for delivering to the application (decrypted), etc. The exact synchronization patter can be defined on a per-session basis.

In the remainder of this chapter, we will describe only the part of the architecture related to the recovery framework. Other flavors, such as ordering, can be implemented in the same way; indeed, they are all expressible within the Properties Language mentioned in Chapter 5 that targets the reliability framework described in this chapter. Examples of frameworks that could be constructed in this manner include infrastructure that implements flow, rate, and congestion control, negotiates transmission rates, manages leases on bandwidth, keys, certificates, grants or revokes access, performs failure detection, health monitoring, and self-verification or detection of inconsistencies such as partitions or invalid routing. In each case, the necessary logic can be partitioned between layers in a hierarchy, encapsulated within an independent framework, and combined at the end hosts (or, in some cases, as with flow or congestion control, at one or more intermediate layers), within a per-session protocol stack.

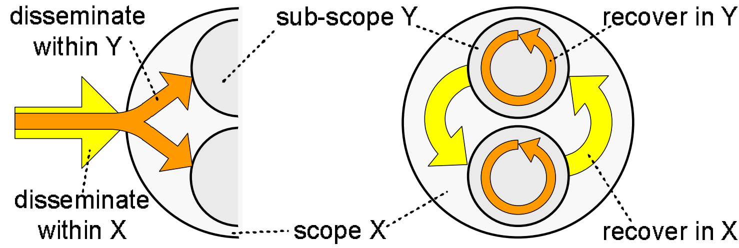

Figure 3.23: The similarities between a hierarchical dissemination (left) and hierarchical recovery (right).

3.2.9 Hierarchical Approach to Reliability

Our approach to reliability resembles our hierarchical approach to dissemination. Just as channels are decomposed into sub-channels, in the reliability framework we decompose the task of loss repair and providing other reliability goals. Recovering messages in a given scope is modeled as recovering within each of its sub-scopes, concurrently and independently, then recovering across all the sub-scopes (Figure 3.23). For example, suppose that scope X has members Y_1, Y_2, ..., Y_K. For the purpose of this example, we assume that this structural decomposition does not change, and membership of X is static (our approach does support scenarios with dynamically changing membership as well, but to keep this example simple, we shall postpone the discussion of dynamic scenarios until later). A simple reliability property $P(X)$ requiring that “if some node x in the span of scope X receives a message m, then for as long as either x or some other node keeps a copy of it, every other node y in the span of X will also eventually get m”, can be decomposed as follows. First, we ensure $P(Y_i)$ for every Y_i, i.e. we ensure that in each sub-scope Y_i of scope X, if one node has the message, then so eventually do the others. The protocols that lead to this goal can run in all these sub-scopes independently and concurrently. Then, we run a protocol across sub-scopes Y_1, Y_2, ..., Y_K, to ensure that if any of them has in its span a node x that received message m, then each of the other Y_i also eventually has a node that received m. When these tasks, i.e. recovery in each Y_i plus the extra recovery across all Y_i, are all performed for sufficiently long, $P(X)$

is eventually established.

Coming back to our example, assume that students with their laptops sit in university departments, each of which is a scope. Suppose that some, but not all of the students received a message with a problem set from their professor sitting in a cafeteria. We would like to ensure that the problem set is reliably delivered to all students. In our system, this would be achieved by a combination of protocols: a protocol running in each department would ensure that internally, for every pair x, y of students, if x got the message then so eventually does y, and likewise a protocol running across the departments ensures that for each pair of departments x, y, if some students in x got the message, so eventually do some students in y. In the end, this yields the desired outcome.

Just like recovery among individual nodes, recovery among LANs might also involve comparing their "state" (such as aggregated ACK/NAK information for the entire LAN) or forwarding lost messages between them. We give an example of this in Section 3.2.11. As mentioned earlier, in our architecture, different recovery schemes may be used in different scopes, to reflect differences in the network topologies, node or communication link capacities, the availability of IP multicast and other local infrastructure, the way subscribers are distributed (e.g., clustered or scattered), etc.

For example, in one department, the machines of the students subscribed to topic T could form a spanning tree. The property we mentioned above could be guaranteed by making neighbors in the tree compare their state, and upon discovering that one of them has a message m that the other is missing, forwarding m between the two of them. The same approach may also be used across the departments, i.e. departments would form a tree, the departments "neighboring" on that tree could compare what their students got, and perhaps arrange for messages to be forwarded between them. For the latter to be possible, the departments need a way to calculate "what their students got", which is an example of an aggregated, "department-wide" state. Finally, some departments could

use a different approach. For example, a department enamored of gossip protocols might require that student machines randomly gossip about messages they got; a department that has had bad experiences with IP multicast and with gossip might favor a reliability protocol that runs on a token ring instead of a tree, and a department with a site-license for a protocol such as SRM (which runs on IP multicast) might favors its use, where the option is available. In each department, a different protocol could be used locally. As long as each protocol produces the desired outcome (satisfies the reliability property inside of the department), and as long as the department has a way to calculate aggregate "department-wide" state needed for inter-department recovery, these very different policies can be simultaneously accommodated.

Just as messages are disseminated through channels, forming what might be termed *dissemination domains*, reliability is achieved via *recovery domains*. A recovery domain D in scope X may be thought of as an instance of a distributed recovery protocol running among some nodes within X that performs recovery-related tasks for a certain set of topics.

For example, when some of the students sitting in a library subscribe to topic T, the library might create a "local recovery domain for topic T". This domain could be "realized", e.g., as a spanning tree connecting the laptops of the subscribed students and running a recovery protocol between them. The library could internally create many domains, e.g., many such trees of student's laptops.

The concept of a recovery domain is dual to the notion of a dissemination channel; here we present the analogy.

- Just like a channel is created to disseminate messages for some topics T_1, T_2, ..., T_k in scope X, a recovery domain is created to handle loss recovery and other reliability tasks, again for a specific set of topics, and in a specific scope. Just like there could exist multiple channels to a scope, e.g., for different sets of topics,

there could also exist multiple recovery domains within a single reliability scope, each ensuring reliability for different sets of topics.

- Just as channels may be composed of sub-channels, a recovery domain D defined at a scope X may be composed of *sub-domains* D_1, D_2, ..., D_n defined at sub-scopes of X (we will call them the *members* of D). Each such sub-domain D_i handles recovery for a set of subscribers in the respective sub-scope, while D handles recovery across the sub-domains. The hierarchy of recovery domains reflects the hierarchy of scopes that have created them, just as channels are decomposed in ways that reflect the hierarchy of scopes that have exposed those channels.
- Just as channels are composed of sub-channels via applying filters assigned by forwarding policies, a recovery domain D performs its recovery tasks using a *recovery protocol*. Such a protocol, assigned to D, specifies how to combine recovery mechanisms in the sub-domains of D into a mechanism for all of D. Recovery protocols are defined in terms of how the sub-domains "interact" with each other. We explain how this is done in more detail in Section 3.2.9.
- Just like a single channel may be used to disseminate messages in multiple topics, a recovery domain may run a single protocol to perform recovery simultaneously for a set of topics. In both cases, reusing a single mechanism (a channel, a token ring, a tree, etc.) may significantly improve performance due to the reduction in the total number of control messages and other such optimizations. Indeed, we use this idea in the system described in Chapter 4.

Each individual node is a recovery domain on its own. In a distributed scope such as a LAN, such as the library in our example, on the other hand, many cases are possible. In one extreme, a single domain may cover the entire LAN. All internal nodes could thus form a token ring, or gossip randomly to exchange ACKs for messages in all topics simultaneously, and use this to arrange for local repairs. In the other extreme, separate

domains could be created for every individual topic; subscribers to the different topics could thus form separate structures, such as separate rings and trees, and run separate protocol instances in each of them, exchanging state and the lost messages. In our system, recovery domains actually handle recovery for specific *sessions*, not just specific topics. Each recovery domain created internally by a scope performs recovery for some set of sessions, and these sets are such that for each session k in which the scope has subscribers, there is a recovery domain in the scope that performs recovery for k.

A recovery domain D of a data center could have as its members recovery domains created by the LANs in that data center (by the SMs of these LANs). Note that in this case, members of D would themselves be distributed domains, i.e. sets of nodes. A recovery protocol running in D would specify how all these different sets of nodes should exchange state and forward lost messages to one another. Note the similarity to a forwarding policy in a data center, which would also specify how messages are forwarded among sets of nodes. As explained in Section 3.2.11 and Section 3.2.13, our recovery protocols are implemented through delegation, just like forwarding. A concept of a *recovery protocol* is, to some degree, an analogue of a *forwarding policy*.

3.2.10 Building the Hierarchy of Recovery Domains

Before we show how the hierarchical recovery scheme can be implemented, we need to explain how domains created at different scopes are related to each other. As explained in the preceding section, domains are organized by the relation of membership: domains in super-scopes can be thought of as containing domains in sub-scopes as members. Just as was the case for scopes, a given domain can have many parents, and there may be multiple global domains, but for a given topic, all domains involved in recovery for that topic always form a tree. Domains know their *members* (sub-domains) and *owners* (super-domains), and through a mechanism described below, also their *peers*

(other domains that have the same parent). This knowledge of *membership* allows the scopes that create those domains to establish distributed structures in an efficient way.

A consistent view of membership is the basis for many reliable protocols, and could benefit many others that do not assume it. Knowing the members of a topic helps to determine which nodes have crashed or disconnected. In existing group communication systems, this is usually achieved by a Global Membership Service (GMS) that monitors failures and membership changes for all nodes, decides when to "install" new membership views for topics, and notifies the affected members of these new views, including the lists of topic members. Nodes then use those membership views to determine, e.g., what other nodes should be their neighbors in a tree, who should act as a leader, etc.

In our framework, the manager of the root scope for a given topic is responsible for creating the top-level recovery domain, announcing when sessions for that topic begin or end, and so forth. However, if the root SM, which in case of an Internet-wide scope would "manage" the entire Internet, had to process all subscriptions, and respond to every failure across the Internet, it would lead to a non-scalable design: beyond a certain point the system would be constantly in the state of reconfiguration, trying to change membership or install new sessions, and hence unable to make useful progress. It would also violate the principle of isolation: the higher-level scopes would process information that should be local, e.g., a corporate network would have to know which nodes in data centers are subscribers, whereas according to our architectural principles, the administrator of a corporate network, and the policies defined at this level, should treat the entire data centers as black boxes.

To avoid the problem just mentioned, rather than collecting all information about membership in a topic T and processing it centrally, we distribute this information across all scope managers in the hierarchy of scopes for topic T (recall this hierarchy defined in Section 3.2.1). Each SM thus has only a partial membership view for each topic and

session. This scheme is outlined below.

In the reliability framework, if a scope X subscribes to a topic T, it first selects or creates a local recovery domain D that will handle the recovery for topic T locally in X, and then sends a request to one of its parent scopes, some Y, asking to subscribe this specific domain, to topic T. At this point, it is not significant which of its parent scopes X directs the request to. X may be manually setup by an administrator with the list of parent scopes, and to send requests in all topics that have names matching a certain pattern to a given parent, or it could use an automated, or a semi-automated scheme to discover the parent scopes that it should subscribe with. Exactly how such a discovery scheme can be most efficiently constructed is beyond the scope of this paper, but we do hope to explore the issue in a future work.

The super-scope Y processes the X's subscription request jointly with requests from other of its sub-scopes, e.g., batching them together for efficiency. It then either joins X and other sub-scopes to an existing recovery domain or creates a new one, some D'. When joining an existing recovery domain, Y follows a special protocol, some details of which are given in Section 3.2.14. In any case, scope Y informs all scopes of the new membership of domain D'. So for each recovery domain D'' that is a member of D', the sub-scope of Y that owns this domain will be notified by Y that domain D' changed membership, and will be given the list of all sub-domains of D', together with the names of the scopes that own those sub-domains. Finally, if domain D' has just been created, and scope Y is not the root scope for the topic, Y itself sends a request to one of its parent scopes, asking to subscribe domain D' to the topic. Topic subscriptions thus travel all the way to the root scope for the topic, in a cascading manner, creating a tree of recovery domains in a bottom-up fashion.

In our example, when a student A sitting in a library L enters a virtual room T, A's laptop creates a local recovery domain D_A, and sends a request to the library server.

Suppose that there are other students in the library that are already subscribed to T, so the library server already has a recovery domain D_L that performs recovery in topic T. In this case all that is left to do for the library server is to update the membership of D_L, and to inform student A's laptop, as well as the laptops of the other students subscribed to T, of the new membership of D_L, so that the laptops can update the distributed recovery structure, and build a new distributed structure. For example, if laptops used a spanning tree, or a token ring, to compare the sets of messages they have received and exchange messages between neighbors, the spanning tree or the ring may be updated to include the new laptop. On the other hand, suppose that student A is the first in the library to subscribe to T. In this case, the library server creates a new recovery domain D_L, with D_A as its only member, and sends its own request, in a cascading manner, to a campus server C, asking to subscribe D_L to topic T, and so on.

The above procedure effectively constructs a hierarchy of sub-domains, with the property that for each topic T, the recovery domains subscribed to T form a tree. At the same time, a membership hierarchy is built in a distributed manner. Specifically, for each domain μ in some scope S, S will maintain a list of the form $\{ X_1{:}\beta_1, X_2{:}\beta_2, \ldots, X_K{:}\beta_K \}$, in which $\beta_1, \beta_2, \ldots, \beta_K$ are the *members*, i.e. sub-domains of domain μ, and $X_1, X_2, \ldots, X_K$ are the names of the scopes that own those sub-domains (i.e. that created them). In the process of establishing this structure, each of the scopes X_i receives the list, along with any future updates to it. This information is not "pushed" all the way down to the leaf nodes. Instead, every scope maintains the membership of the domains it created (e.g., scope S maintains the list mentioned above), plus a copy of the membership of all super-domains of the domains it created (e.g., each X_i has a copy of the list above), but not the membership of any domains created below it (so, for example, S would not track the membership of any of the domains $\beta_1, \beta_2, \ldots, \beta_K$), more than one level above it (e.g., while scope S would know what are the peers of its own domain μ, scopes X_1, X_2,

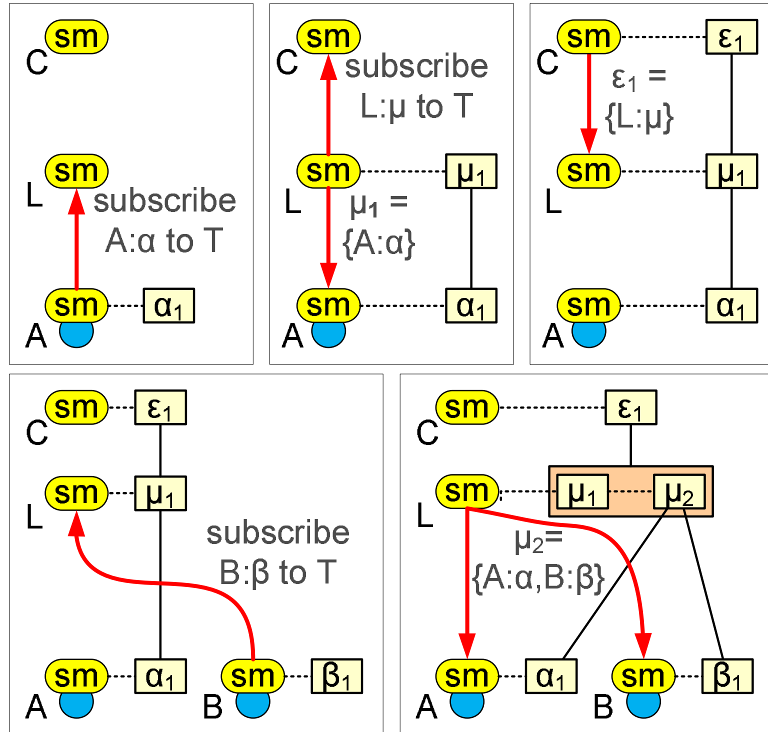

Figure 3.24: Membership information and view numbers are passed one level down (never up) the hierarchy. Node A subscribes to topic T with the library L. Library L subscribes with campus C.

…, X_K would not know those peers, because this information is, logically, two levels "above"), or any internal details of its peers (e.g., while all of X_1, X_2, …, X_K would know the membership of μ, i.e. the entire list $\{ X_1{:}\beta_1, X_2{:}\beta_2, \ldots, X_K{:}\beta_K \}$, they would not know the membership of any of the domains $\beta_1, \beta_2, \ldots, \beta_K$ besides their own; they only know the names of their peer domains).

Figure 3.24 shows an example of the structure the system would construct in a scenario similar to the one above. Laptop A creates a new domain α, which may have a version number, like any other domain, say α_1. Then, A directs a request "subscribe $A{:}\alpha$ to T" to the library L. Note that the version number of a was not included in the request to L. This is a detail internal to A that L does not need to know about. Now, the library creates a new domain μ, and gives it a version number, say μ_1, and directs subscribe "$L{:}\mu$ to T" to the campus C (omitting the version number). Concurrently, L notifies A

that $A{:}\alpha$ is now a member of μ_1. This means that a domain μ has been created at L, and version (view) number 1 of μ has just a single member $A{:}\alpha$. Similarly, C creates a new domain ϵ with initial version ϵ_1 that includes a single member $L{:}\mu$ and notifies L. Later, another laptop B in the library L also joins topic T. This time, no request is sent to campus C. The library handles the request internally. A new version (view) μ_2 of domain μ is created, with two members $A{:}\alpha$ and $B{:}\beta$, and both A and B are notified of this new view. A and B undergo a special protocol to "transition" from recovery domain μ_1 to recovery domain μ_2 in a reliable manner, and the protocol running for μ_1 eventually quiesces. The protocols that run at higher levels are unaffected. Domain ϵ_1 still has only a single member $L{:}\mu$, and the view change that occurred internally in domain μ is transparent to C, and to the protocols that run at this level, and handled internally in the library L, between nodes A and B.

By keeping the information about the hierarchy of domains distributed, and by limiting the way in which this information is propagated to only one level below, we remain faithful to the principles of isolation and local autonomy laid out earlier. At the same time, this enables significant scalability and performance benefits. Because parent domains are oblivious to the membership of their sub-domains, and reconfiguration can often be handled internally, as in the example above, churn and failures in lower layers of the hierarchy do not translate to churn and failures in higher layers. A failure of a node, or a mobile user with a laptop joining or leaving the system, does not need to cause the Internet-wide structure of recovery domains, potentially spanning across tens of thousands of nodes, to fluctuate and reconfigure.

As stated earlier, a single recovery domain may perform recovery for multiple topics (sessions), simultaneously. Additionally, recall that the domain hierarchy, with multiple topics, may not be a tree. We now present an example of how and why this could be the case. Suppose that nodes in a certain scope L are clustered based on their interest.

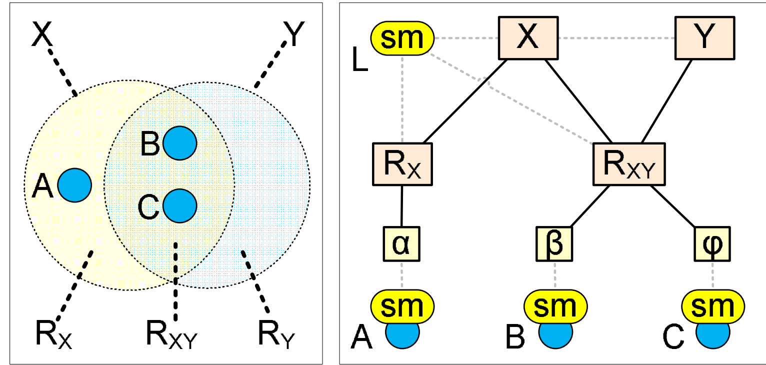

Figure 3.25: A hierarchy of recovery domains in a system that clusters nodes based on interest and maintains recovery scopes corresponding to each of the clusters.

Nodes A and B would be in the same cluster if A and B subscribed to the same topics. Clusters are thus defined by sets of topic names, e.g., cluster R_{XY} would include nodes that have subscribed to topics X and Y, and that have not subscribed to any other topics besides these two. The set of nodes subscribed to each topic would thus include nodes in a certain set of clusters. We might even think of as topics "including" clusters, and the clusters including the individual nodes (Figure 3.25). Accordingly, a scope manager in L might create a separate recovery domain for each cluster, and then a separate recovery domain for each topic. If node A subscribes to topic X, and nodes B and C subscribe to topic Y, then L could create a recovery domain R_X for cluster R_X and a recovery domain R_{XY} for cluster R_{XY}. Domain R_X would have a single member A:α, while R_{XY} would have two members, B:β and C:ϕ. Scope L would also create a local domain L:X for topic X and L:Y for topic Y. Domain L:X would have members L:R_X and L:R_{XY} while domain L:Y would have a single member L:R_{XY}. Domains L:X and L:Y defined at scope L could themselves be members of some higher-level domains, defined at a higher-level scope C, and so on. Now, the protocol running in domain R_{XY} at scope L, for example, would perform recovery simultaneously for topics X and Y. As said earlier, the protocol running in R_{XY} would also be used to calculate aggregate information about domain R_{XY}, to be used in the higher-level protocols. In our example, the information collected by

the protocol running in $L{:}R_{XY}$ would be used by two such protocols, a protocol running in domain $L{:}X$ and a protocol running in $L{:}Y$.

While the structure just described may seem complex, the ability to perform recovery in multiple topics simultaneously is important in systems like virtual worlds, where the number of topics (virtual rooms) may be very large.

To complete the discussion of recovery hierarchy, we now turn to sessions. As explained earlier, in our architecture recovery is always performed in the context of individual sessions, not topics, because whenever a session changes, so can the reliability properties of the topic. The creation of the domain hierarchy, outlined above, is mostly independent of the creation of sessions. The only case when these two processes are synchronized arises when the last member, across the entire Internet, leaves the topic, or when the first member rejoins the topic after a period when no members existed, for in such cases, it is impossible to handle the event via a local reconfiguration between members (such as transferring state from some existing member to the newly joining one, or "flushing" changes from the departing member to some of the existing members). Such an event will force an existing session to be flushed or a new session to be created.

In any case, sessions for a topic T are created by the scope that serves as the root for T. The root maintains topic metadata and the information about sessions in persistent storage. It assigns new session number whenever a new session is created, and then *installs* the new session and *flushes* the existing session in the top-level recovery domain that it created to perform recovery in topic T. More on how the installing and flushing of a session are realized will be explained in Section 3.2.13. Now, concurrently with installing of a new session and flushing of the old session in the top-level recovery domain, the scope manager passes the session change event further, to the sub-domains that are members of this global recovery domain, by communicating with the scope managers that created those sub-domains. This notification travels down the hierarchy of recovery

domains in a cascading manner, until the session change events are disseminated across the entire structure.

3.2.11 Recovery Agents

The reader will have noticed by now that the structure of recovery domains we have just described "exists" only virtually, inside the scope managers. These recovery domains are "implemented" by physical protocols running directly between physical nodes, the publishers and subscribers, in a manner similar to how we implemented channels between scopes, by *delegating* the tasks that the recovery domains are responsible for to physical nodes. Just as channels between scopes are "implemented" by physical connections between nodes that can be constrained with filter chains and that are "installed" in the physical nodes by their super-scopes, in the reliability framework recovery domains are "implemented" by *agents*. Similarly to filters, these agents are also small, "downloadable" components, which are installed on physical nodes by their super-scopes. Before going into details of how precisely this is done, however, we first explain how existing recovery protocols can be modeled in a hierarchical manner that is compatible with our architecture.

3.2.12 Modeling Recovery Protocols

The *reliability framework* is based on an abstract model of a scalable distributed protocol dealing with loss recovery and other reliability properties. In this model, a protocol such as SRM, RMTP, virtual synchrony, or atomic commit, is defined in terms of a group of cooperating *peers* that exchange control messages and can forward lost packets to each other, and that may perhaps interact with a distinguished node, such as a sender or some node higher in a hierarchy, which we will refer to as a *controller*(Figure 3.26). The con-

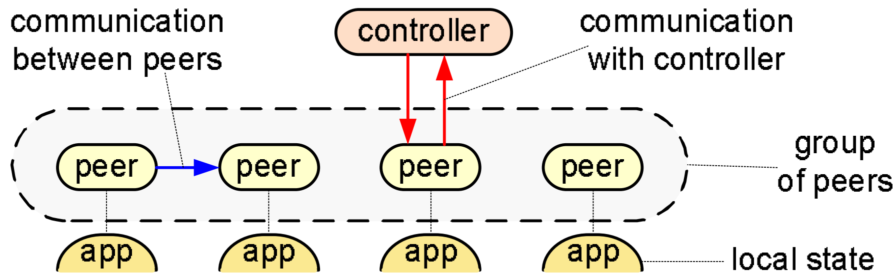

Figure 3.26: A group of peers in a reliable protocol.

troller does not have to be a separate node; this function could be served by one of the peers. The distinction between the peers and the controller may be purely functional. The point is that the group of peers, as a whole, may be asked to perform a certain action, or calculate a value, for some higher-level entity, such as a sender, a higher-level protocol, or a layer in a hierarchical structure. Examples of such actions include retransmitting or requesting a retransmission for all peers, reporting which messages were successfully delivered to all peers, which messages have been missed by all peers, etc. Irrespectively of how exactly the interaction with the controller is realized, it is present in this form or another in almost every protocol run by a set of receivers. We shall refer to the possible interactions between the peers and the controller as the *upper interface*. Notice that some reliability protocols are not traditionally thought of as hierarchical; we would view them as supporting only a one-level hierarchy. The benefit of doing so is that those protocols can then be treated side by side with protocols such as SRM and RMTP, in which hierarchy plays a central role.

Each peer inspects and controls its *local state*. Such state could include, e.g., a list of messages received, and perhaps copies of those that are cached (for loss recovery), the list and the order of messages delivered, etc. Operations that a peer may issue to change the local state could include retrieving or purging messages from cache, marking messages as deliverable, delivering some of the previously missed message to the application, etc. We refer to such operations, used to view or control the local state of a

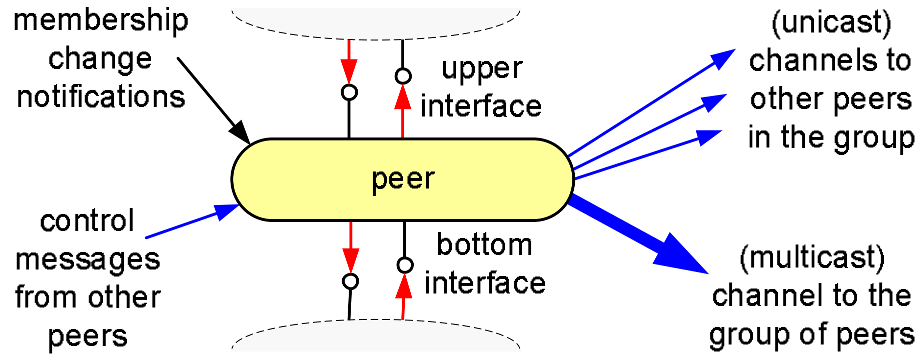

Figure 3.27: A peer modeled as a component living in abstract environment (events, interfaces, etc.).

peer, as a *bottom interface*.

In protocols offering strong guarantees, peers are typically given the membership of their group, received as a part of the initialization process, and subsequently updated via *membership change* events. Peers send control messages to each other to share state or to request actions, such as forwarding messages. Sometimes, as in SRM, a multicast channel to the entire peer group exists.

To summarize, in most reliable protocols, a peer could be modeled as a component that runs in a simple environment that provides the following interface: a *membership view* of its peer group, *channels* to all other peers, and sometimes to the entire group, a *bottom interface* to inspect or control local state, and an *upper interface*, to interact with the sender or the higher levels in the hierarchy concerning the aggregate state of the peer group (Figure 3.27). In some protocols, certain parts of this interface might be unavailable, e.g., in SRM peers might not know other peers. The bottom and upper interfaces also would vary.

This model is flexible enough to capture the key ideas and features of a wide class of protocols, including virtual synchrony. However, because in our framework protocols must be reusable in different scopes, they may need to be expressed in a slightly different way, as explained below.

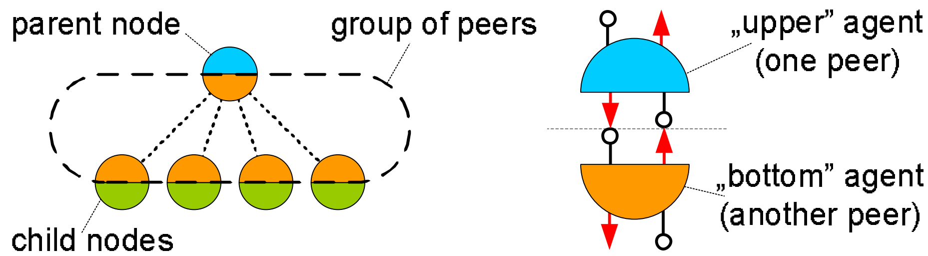

Figure 3.28: RMTP expressed in our model. A node hosts "agents" playing different roles.

In RMTP, the sender and the receivers for a topic form a tree. Within this tree, every subset of nodes consisting of a parent and its child nodes represents a separate local recovery group. The child nodes in every such group send their local ACK/NAK information to the parent node, which arranges for a local recovery within the recovery group. The parent itself is either a child node in another recovery group, or it is a sender, at the root of the tree. Packet losses in this scheme are recovered on a hop-by-hop basis, either top-down or bottom-up, one level at a time. This scheme distributes the burden of processing the individual ACKs/NAKs, and of retransmissions, which is normally the responsibility of the sender. This improves scalability and prevents ACK implosion.

There are two ways to express RMTP in our model. One approach is to view each recovery group consisting of a parent node and its child nodes as a separate group of peers (Figure 3.28). Since internal nodes in the RMTP tree simultaneously play two roles, a "parent" node in one recovery group and a "child" node in another, we could think of each node as running two "agents", each representing a different "half" of the node, and serving as a peer in a separate peer group. In this perspective it would be not the nodes, but their "halves" that would represent peers. Every group of peers, in this perspective, would include the "bottom agent" of the parent node, and the "upper agents" of its child nodes. When a node sends messages to its child nodes as a result of receiving a message from its parent, of vice versa, we may think of those two "agents" as interacting with each other through a certain interface that one of them views as upper,

and the other as bottom. These two types of agents play different roles in the protocol, as explained below.

The bottom agent of each node interacts via its bottom interface with the local state of the node. It also serves as a distinguished peer in the peer group, composed of itself and the upper agents of the child nodes. A protocol running in this peer group is used to exchange ACKs between child nodes and the parent node and arrange for message forwarding between peers, but also to calculate collective ACKs for the peer group, i.e. which messages were not recoverable in the group. This is communicated by the bottom agent, via its upper interface, to the upper agent. The upper agent of every node interacts via its bottom interface with the bottom agent. What the upper agent considers as its "local state" is not the local state of the node. Instead, it is the state of the entire recovery group, including the parent and child nodes, that is collected for the upper agent by the bottom agent though the protocol that the bottom agent runs with the upper agents in child nodes. Such interactions, between a component that is logically a part of a "higher layer" ("upper agent") with components that reside in a "lower layer" ("bottom agent"), both components co-located on the same physical node, and connected via their upper and bottom interfaces, are the key element in our architecture.

At the top of this hierarchy is the sender, the root of the tree. The bottom agent of the sender node collects for the upper agent the state of the top-level recovery group, which subsumes the state of the entire tree, and passes it to the upper agent through its upper interface. The upper agent of the sender can thus be thought of as "controlling" through its bottom interface the entire receiver tree.

The second way to model RMTP, which builds on the concepts we just introduced, captures the very essence of our approach to combining protocols. It is similar to the first model, but instead of the "upper" and "bottom" agents, each node can now host multiple agents, again connected to each other through their "bottom" and "upper" interfaces.

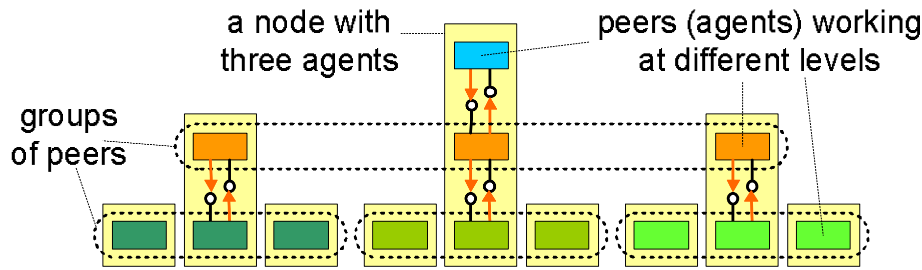

Figure 3.29: Another way to express RMTP. Each node hosts multiple ”agents” that act as peers at different levels of the RMTP hierarchy.

Each of these agents works at a different level. We may think of every node as hosting a “stack” of interconnected agents (Figure 3.29). In this structure, the sender would not be the root of the hierarchy any more. Rather, it would be treated in the very same way as any of the receivers. The same structure could feature multiple senders, and a single recovery protocol would run for all of them simultaneously.

Focusing for a moment on a concrete example, assume that nodes reside in three LANs, which are part of a single data center (Figure 3.29). Each of these administrative domains is a scope. Each node hosts, in its “agent stack”, a “node agent” (bottom level, green, Figure 3.29). The “local state” of the node agent, accessed by the node agent through its bottom interface, is the state of the node, such as the messages that the node received or missed, etc. Now, in each LAN, the node agents of all nodes in that LAN form a peer group and communicate with each other to compare their local state, or to arrange for forwarding messages between them. One of these node agents in each LAN serves as a leader (or “parent”), and the others serve as subordinates (or “children”). The leader collects the aggregate ACK/NAK information about the LAN from the entire peer group. The node that hosts the leader also runs anther, higher-level component that we shall call a “LAN agent” (middle level, orange, Figure 3.29). The LAN agent accesses, through its bottom interface, the aggregated state of the LAN that the “leader” node agent, co-located with it on the same “leader” node, calculated. The LAN agent can therefore be thought of as controlling, through its bottom interface, the entire LAN,

just like a node agent was controlling the local node. Now, all the LAN agents in the data center again form a peer group, compare their state (which are aggregate states of their LANs), arrange for forwarding (between the LANs), and calculate aggregate ACK/NAK information about the data center. Finally, one of the nodes that host the LAN agents hosts an even higher-level component, a "data center agent" (top level, blue, Figure 3.29). The aggregate state of the data center, collected by the peer group of LAN agents, is communicated through the upper interface of the "leader" LAN agent to the data center agent; the latter can now be thought as controlling, through its bottom interface, the entire data center. In a larger system, the hierarchy could be deeper, and the scheme could continue recursively.

Note the symmetry between the different categories of agents. In essence, for every entity, be it a single node, a LAN, or a data center, there exists exactly one agent that collects, via lower-level agents, the state of the entity it represents, and that acts on behalf of this entity in a protocol that runs in its peer group. The agent that represents a distributed scope is always hosted together with one of the agents that represent sub-scopes. By now, the reader should appreciate that this structure corresponds to the hierarchy of recovery domains we introduced in Section 3.2.10. In our design, every recovery domain is represented, as a part of some higher-level domain, by an agent that collects the state of the domain, which it represents, and acts on behalf of it in a protocol that runs among the agents representing other recovery domains that have the same parent (Figure 3.30). In order to be able to do their job, these agents are updated whenever a relevant event occurs. For example, they receive a membership change notification, with the list of their peer agents, when a new recovery domain is created. To this end, each agent maintains a bi-directional channel from the scope that created the recovery domain represented by this agent, down to the node that hosts the agent, along what we call an agent "delegation chain".

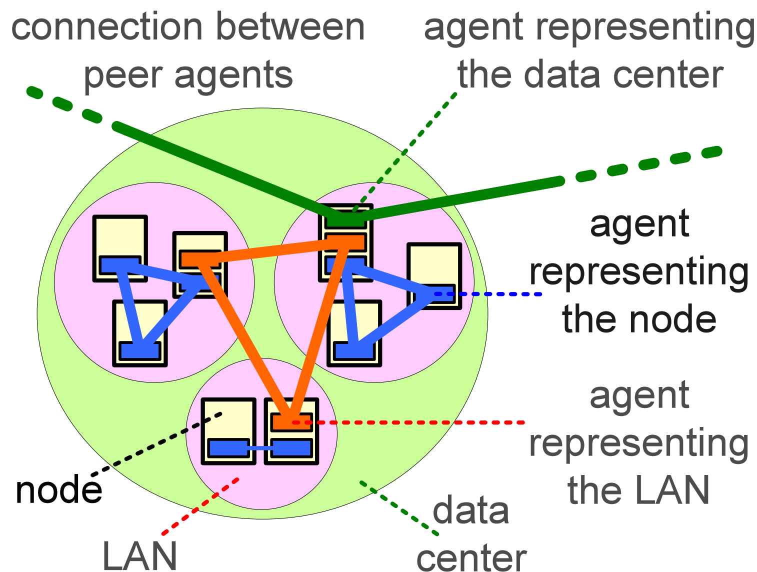

Figure 3.30: A hierarchy of recovery domains and agents implementing them.

Note also that as long as the interfaces used by agents to communicate with one-another are standardized, each group of agents could run an entirely different protocol, because the only way the different peer groups are connected with each other is through the bottom and upper interfaces of their agents. For example, in smaller peer groups with small interconnect latency agents could use a token ring protocol, whereas in very large groups over a wide area network, with frequent joins and leaves and frequent configuration changes agents might use randomized gossip protocol. This flexibility could be extremely useful in settings where local administrators control policies governing, for example, use of IP multicast, and hence where different groups may need to adhere to different rules. It also allows for local optimizations. Protocols used in different parts of the network could be adjusted so as to match the local network topology, node capacities, throughput or latency, the present of firewalls, security policies, etc. Indeed, we believe that the best approach to building high-performance systems on the Internet scale is not through a uniform approach that forces the use of the same protocol in every part of the network, but by the sorts of modularity our architecture enables, because it can leverage the creativity and specific domain expertise of a very large class of users,

who can tune their local protocols to match their very specific needs.

The flexibility enabled by our architecture also brings a new perspective on a node in a publish-subscribe system. A node subscribing to the same topics in different portions of the Internet, joining an already established infrastructure (existing recovery domains and agents implementing them running among existing subscribers) may be forced to follow a different protocol, potentially not known in advance. Indeed, if we exploit the full power of modern runtime platforms, a node might be asked to use a protocol that must first be downloaded and installed, in plug-and-play fashion. Because in our architecture, elements "installed" in nodes by the forwarding framework (filters and channels) and by the recovery framework (agents) are very simple, and communicate with the node via a small, standardized API (recall the abstract model of a peer on Figure 3.27), which can also be interpreted as the abstract model of an agent in the recovery framework), these elements can be viewed as downloadable software components.

Thus, a filter or a recovery agent can be a piece of code, written in any popular language, such as Java or one of the family of .NET languages, that exposes (to the node hosting it, or to agents above and below in the agent stack) and consumes a standardized interface, e.g., described in WSDL. Such components could be stored in online repositories, and downloaded as needed, much as a Windows XP user who tries to open a file in the new Vista XPS format will be prompted to download and install an XPS driver, or a visitor to a web page that uses some special ActiveX control will be given an opportunity to download and install that control. In this perspective, the subscribers and publishers that join a publish-subscribe infrastructure, rather than being applications compiled and linked with a specific library that implements a specific protocol, and thus very tightly coupled with the specific publish-subscribe engine, can now be thought of as "empty containers" that provide a standard set of hookups to host different sorts of agents. Nodes using a publish-subscribe system are thus runtime platforms,

programmable "devices", elements of a large, flexible, programmable, dynamically reconfigurable runtime environment, offering the sort of flexibility and expressive power unseen in prior architectures.

We believe that in light of the huge success of extensible, component-oriented programming environments, standards for distributed eventing must incorporate the analogous forms of flexibility. To do otherwise is to resist commercial off-the-shelf (COTS) trends, and history teaches that COTS solutions almost always dominate in the end. It is curious to realize that although web services standards were formulated by some of the same companies that are leaders in this componentized style of programming, they arrived at standards proposals that turn out to be both rigid and limited in this respect.

One part of our architecture, for which API standardization options may not be obvious, includes the upper and bottom interfaces. As the reader may have realized, the exact form of these interfaces would depend on the protocol. For example, while a simple protocol implementing the "last copy recall" semantics of the sort we used in some of our examples require agents to be able to exchange a simple ACK/NAK information, more complex protocols may need to determine if messages have been persisted to stable storage, to be able to temporarily suppress the delivery of messages to the application, control purging messages from cache, decide on whether to commit a message (or an operation represented by it) or abort it, etc. The "state" of recovery domain and the set of actions that can be "requested" from a recovery domain may vary significantly. As it turns out, however, defining the upper and bottom interfaces in a standard way is possible for a wide range of protocols through set of *properties* discussed in Chapter 5.

To conclude this section, we now turn to recovery in many topics at once. Throughout this section, the discussion focused on a single topic, or a single session, but as mentioned before, the recovery domains created by the reliability framework, and hence the sets of agents that are instantiated to "implement" those recovery domains, may be

requested to perform recovery in multiple sessions at once, for reasons of scalability. As mentioned earlier, after recovery domains are established, and agents instantiated, the root scope may issue requests to install or flush a session, passed along the hierarchy of domains, in a top-down fashion. These notifications are a part of the standard agent API. Agents respond to the notifications by introducing or eliminating information related to a particular session in the state they maintain or control messages they exchange.

For example, agents in a peer group could use a token ring protocol and tokens circulating around that ring could carry a separate recovery record for each session. After a new session would get installed, the token would start to include the recovery record for that session. When a flushing request would arrive for a session, the session would eventually quiesce, and the recovery record related to that session would be eliminated from the tokens, and from the state kept by the agents. The exact manner in which introducing a new session and flushing are expressed would depend on how the agent implements it. The available agent implementation might not support parallel recovery in many sessions at once; in this case, the SM could simply create a separate recovery domain for each topic, or even for each session, so that separate agents are used for each.

If an agent performs recovery for many sessions simultaneously, its "upper" and "bottom" interfaces would be essentially arrays of interfaces, one for each session. Likewise, agent stacks might no longer be vertical. The stacks for a scenario of Figure 3.25 are shown on (Figure 3.31). Here, agents on node *B* form a tree. Two parts of the upper interface of one agent that correspond to two different sessions that the agent is performing recovery for, are connected with two bottom interfaces of two independent higher-level agents. At this point, the structure depicted in this example may seem somewhat confusing; in Section 3.2.13, we come back to this scenario, and we explain how such elaborate structures can be automatically constructed.

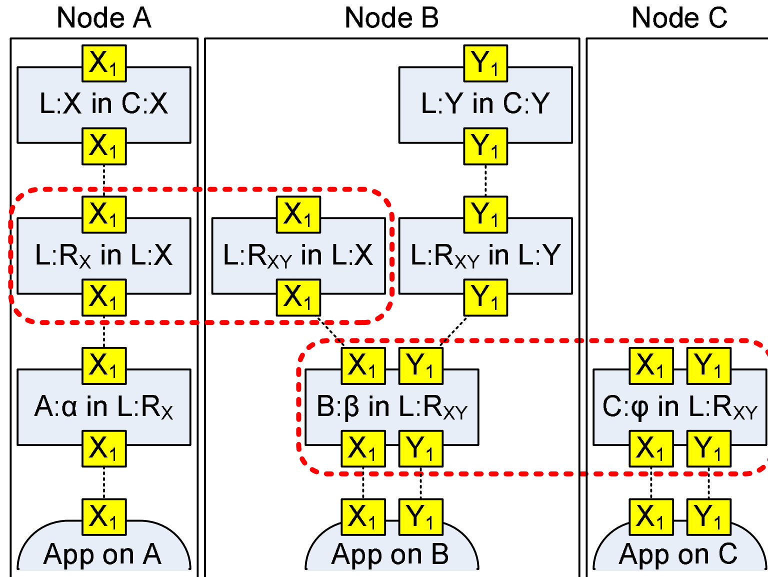

Figure 3.31: Agents stacks that are not vertical and that may be created in the scenario from Figure 3.25. Agents are shown as gray boxes, with the parts of their upper or bottom interfaces corresponding to particular sessions as small yellow boxes. On node *B*, a bottom-level agent connects each of the two parts of its upper interface to one of two different higher-level agents. The peer groups are circled with thick dotted red lines.

3.2.13 Implementing Recovery Domains with Agents

In Section 3.2.10 we have explained how a hierarchy of recovery domains is built, such that for each session, there is a tree of domains performing recovery for that session. In Section 3.2.11 we indicated that the recovery domains are "implemented" with agents, and in Section 3.2.12 we explained how recovery protocols can be expressed in a hierarchical manner, by a hierarchy of agents that represent recovery domains. We now explain how agents are created.

A distributed recovery domain D in our framework (i.e. a domain different than a node, not a leaf in the domain hierarchy) will correspond to a peer group. When D is created at some scope X, the latter selects a protocol to run in D, and then every subdomain Y_k:D_k of D is requested to create an agent that acts as a "peer Y_k:D_k within peer group X:D". We will refer to an agent defined in this manner as "Y_k:D_k in X:D". Note how the membership algorithm provides membership view at one level "above", i.e. the

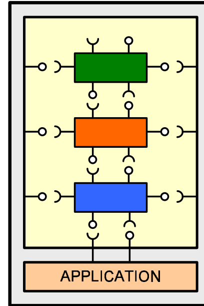

Figure 3.32: A node as a "container" for agents.

scope that owns a particular domain would learn about domains in all the sibling scopes. This is precisely what is required for each peer Y_k:D_k in a peer group X:D to learn the membership of its group. Hence, the scopes Y_k that own the different domains D_k will learn of the existence of domain X:D, each of them will realize that they need to create an agent "Y_k:D_k in X:D", and each of them will receive from X all the membership change events it needs to keep its agent with an up to date list of its peers.

For example, on Figure 3.25, domains B:β and C:ϕ are shown as members of L:R_{XY}, so according to our rules, agents "B:β in L:R_{XY}" and "C:ϕ in L:R_{XY}" should be created to implement L:R_{XY}. Indeed, the reader will find those agents on Figure 3.31, in the protocol stacks on nodes B and C.

When the manager of a scope Y discovers that an agent should be created for one of its recovery domains D_k that is a member of some X:D, two things may happen. If X manages a single node, the agent is created locally. Otherwise, Y delegates the task to one of its sub-scopes. As a result, the agents that serve as peers at the various levels of the hierarchy are eventually delegated to individual nodes, their definitions downloaded from an online repository if needed, placed on the agent stack and connected to other agents or to the applications. We thus arrive at a structure just like on Figure 3.29, Figure 3.30, Figure 3.32, (Figure 3.32). and Figure 3.31, where every node has a stack of agents, linked to one another, with each of them operating at a different level.

While agents are delegated, the records of it are kept by the scopes that recursively

delegated the agent, thus forming a *delegation chain*. This chain serves as a means of communication between the agent and the scope that originally requested it to be created. The scope and the agent can thus send messages to one another. This is the way membership changes or requests to install or flush sessions can be delivered to agents.

When the node hosting a delegated agent crashes, the node to which that agent is delegated changes. That is, some other node is assigned the role of running this agent, and will instantiate a new version of it to take over the failed agents responsibilities. Here, we again rely on the delegation chain. When the manager of the super-scope of the crashed node (e.g., a LAN scope manager) detects the crash, it can determine that an agent was delegated to the crashed node, and it can request the agent to be re-delegated elsewhere. Since our framework would transparently recreate channels between agents, it would look to other peers agents as if the agent lost its cached state (not permanently, for it can still query its bottom interface and talk to its peers). On the one part, this frees the agent developer from worrying about fault-tolerance. On the other part, this requires that agent protocols be defined in a way that allows peers to crash and "resume" with some of their state "erased". Based on our experience, for a wide class of protocols this is not hard to achieve.

3.2.14 Reconfiguration

Many large systems struggle with costs triggered when nodes join and leave. As a configuration scales up, the frequency of join and leave events increases, resulting in a phenomenon researchers refer to as "churn". A goal in our architecture was to support protocols that handle such events completely close to where they occur, but without precluding global reactions to a failure or join if the semantics of the protocol demand it. Accordingly, the architecture is designed so that management decisions and the responsibility for handling events can be isolated in the scope where they occurred. For

example, in Section 3.2.10 we saw a case in which membership changes resulting from failures or nodes joining or leaving were isolated in this manner. The broad principle is to enable solutions where the global infrastructure is able to ignore these kinds of events, leaving the local infrastructure to handle them, without precluding protocols in which certain events do trigger a global reconfiguration.

An example will illustrate some of the tradeoffs that arise. Consider a group of agents implementing some recovery domain D that has determined that a certain message m is locally cached, and reported it as such to a higher-level protocol. But now suppose that a node crashed and that it happens to have been the (only) one on which m was cached. To some extent, we can hide the consequences of the crash: D can reconfigure its peer group to drop the dead node and reconstruct associated data structures. Yet m is no longer cached in D and this may have consequences outside of D: so long as D cached m, higher level scopes could assume that m would eventually be delivered reliably in D; clearly, this is no longer the case.

This is not the setting for an extended discussion of the ways that protocols handle failures. Instead, we limit ourselves to the observations already made: a typical protocol will want to conceal some aspects of failure handling and reconfiguration, by handling them locally. Other aspects (here, the fact that m is no longer available in scope D) may have global consequences and hence some failure events need to be visible in some ways outside the scope. Our architecture offers the developer precisely this flexibility: events that he wishes to hide are hidden, and aspects of events that he wishes to propagate to higher-level scopes can do so.

Joining presents a different set of challenges. In some protocols, there is little notion of state and a node can join without much fuss. But there are many protocols in which a joining node must be brought up to date and the associated synchronization is a traditional source of complexity. In our own experience, protocols implementing

reconfiguration (especially joins) can be greatly facilitated if members of the recovery domains can be assigned certain "roles". In particular, we found it useful to distinguish between "regular" members, which are already "up to date" and are part of an ongoing run of a protocol, and "light" members, which have just been added, but are still being brought up to date.

When a new member joins a domain, its status is initially "light" (unless this is the first membership view ever created for this domain). The job of the "light" members, and their corresponding agents, is to get up-to-date with the rest of their peer group. At some point, presumably when the light members are more or less synchronized with the active ones, the "regular" agents may agree to briefly suspend the protocol and request some of these "light" peers to be promoted to the "regular" status.

The ability to mark members as "light" or "regular" is a fairly powerful tool. It provides agents with the ability to implement certain forms of agreement, or consensus protocols that would otherwise be hard to support. In particular, this feature turns out to be sufficient to allow our architecture to support virtually synchronous, consensus-like, or transactional semantics.

3.2.15 Reliability and Consistency

In this last section, we briefly comment on the expressive power of the architecture we proposed, and the way we can support strong semantics. The main issue here stems from the fact that notions such as consistency or reliability have typically been expressed in terms of global membership views. Thus, for example, one could talk about a message being eventually delivered to all non-faulty nodes within a certain view, or delivery being suppressed until all non-faulty nodes within a view receive and acknowledge it; the notion of being non-faulty is also typically expressed in terms of membership, for example a node could be considered faulty in a view if it is not a member of one of the

subsequent membership views. In the architecture we have proposed, we eliminated the notion of a global view as fundamentally non-scalable and violating design principles such as local autonomy, isolation, and encapsulation. The membership is decentralized, represented as a hierarchy of membership views maintained by different scope managers. The complete hierarchy is never delivered to the clients, and it never materializes in any part of the system. Hence, expressing semantics in terms of global views is impossible in this new model we propose, and defining reliability and consistency in a rigorous way becomes significantly more difficult.

An in-depth discussion of the decentralized membership model and the way strong semantics can be supported in it is a work in progress, and is beyond the scope of this dissertation. The problem is closely related to, and in a sense it is a foundation for our work on the Properties Framework of Chapter 5. We comment on the issue and outline our future plans in Section 6.2. Here, we limit ourselves to a few brief comments.

The key to understanding the architecture is to realize that just as the network of scope managers forms a hierarchy, so does a protocol running by a network of interconnected agents, and that agents running at different levels, in a sense, perform partial computations within the protocol. Although the global membership of the system never arises anywhere, the results of such partial computations can, in fact, correspond to well defined sets of nodes. We explain this by example.

Consider a part of a reliable multicast protocol that is concerned with cleaning up messages. The protocol needs to determine when a message is stable on all nodes that might need a copy, and makes a global cleanup decision as soon as this property holds. To this purpose, the protocol needs to calculate a predicate stating that “message k is stable on all nodes”. In our hierarchical architecture, such computation would performed in a hierarchical manner. Thus, for example, a group of interconnected agents at the lowest level in the hierarchy of a given topic would calculate that message k is stable

in their peer group. The results of calculations made by several peer groups would then be aggregated by a peer group running at a higher level. Cascading aggregations of this sort would eventually result in the top-level agent learning that k is stable everywhere.

Now, of course, the membership of the system could change, and so would the notion of “everywhere”. The key realization here is first, that the notion of “everywhere” can have a well defined meaning, and secondly, that distributed membership changes could be orchestrated in such a way that once a global predicate such as “a message k is stable everywhere” is calculated, this predicate remains true despite the membership changes that might have occurred in the network.

To understand this, suppose that whenever a predicate is calculated, a hidden “certificate” is attached to it that documents the way the computation took place. The certificate would be a tree of membership views corresponding to all the recovery domains that took part in the aggregation, including the specific views, times or rounds within those views etc. If x is a node in the membership tree, then the child nodes of x would be membership views of entities that are in the view x. At the lowest level, individual nodes would produce hidden certificates with their names and incarnation numbers, and as the value of the predicate is aggregated in a bottom up fashion, hidden certificates would also be combined into higher-level certificates by the peer groups performing the aggregation. Eventually, when a predicate such as “message k is stable on all nodes” surfaces at the root agent, it would be accompanied by a hidden certificate representing a tree of membership views descending from the recovery domain of the root agent, down to a well defined set of the individual nodes that the predicate was calculated over. Every such certificate would thus determine a global membership. Of course, this hidden certificate would not be a part of the actual running protocol, but rather a virtual concept one could use to define and prove the protocol’s properties.

Now, just as in traditional systems one defines strong properties in terms of values of

certain predicates in global views, in our architecture one would define such properties in terms of such hidden certificates, and express the distributed behavior of the membership infrastructure in terms of view hierarchies reflected in such certificates.

Thus, for example, one could state that once the protocol computes a certificate such as “message k is stable everywhere” at any point, every subsequent computation made by the protocol would be performed using a more up-to-date information; this property is called “monotonicity”, and is a fundamental concept in the Properties Framework of Chapter 5. Monotonicity could be expressed in terms of certificates by stating that in every new computation, the hidden certificate must be such that whenever a view of the same recovery domain is present in both certificates, the view in the “newer” certificate is also “newer”, i.e. it has a higher view number, a higher round, time, or incarnation number etc. This, in turn, appears to be possible to achieve in a scalable manner, by carefully defining the mini-protocol run by each peer group, and without the need for any centralized infrastructure. Although a formal proof is a work in progress, experiments reported in [238], with tens of thousands of nodes and extremely high churn rates, suggest that our intuition is correct.

Likewise, one could state that if nodes nodes join the protocol, they can only take part in a new computation provided that they have sufficiently “caught up” with the rest of the group. This can again be expressed in terms of hidden certificates, by saying that if a new value is to be aggregated such that in its hidden certificate a new recovery domain is present, the value of the calculated predicate must not be affecting the state of the aggregation. This can be made more formal, but we won’t go into details at this point. Intuitively, this would ensure, for example, that if a new node is added to a group that considered a certain message as stable, the new node must also have a copy of the message, so that the result of the earlier computation is respected. Until this holds, the node could participate in peer-to-peer recovery, and could request state

transfers, but it would not take part in voting on whether messages can be cleaned up; in the terminology of the preceding section, it would serve as a "light" member. This sort of "guarded join" semantics again appears to be expressible in terms of certificates, and possible to implement in a scalable manner. Again, although a rigorous formal proof is still needed, experiments reported in [238] suggest that the approach is valid.

Combined with monotonicity, guarded join could be used to argue that whenever the protocol calculates a global predicate "message k is stable everywhere", the result of such computation will remain the same under certain very weak assumptions (such as that the entire hierarchical infrastructure does not collapse). This, in turn, allows to talk about "irreversible" decisions, and appears to be sufficient to prove facts such as that messages are not "lost" by the protocol, i.e. if a message has once been transmitted, a newly joining node will always be able to eventually obtain either a copy of the message, or a copy of replicated state affected by this message (again under weak assumptions). This form of a guarantee appears to fit well into the live objects model of Section 2.

Chapter 4
Performance

In the preceding chapters, we have proposed a new programming model and an object-oriented multicast architecture supporting it. In this chapter, we report on our experiences building QuickSilver Scalable Multicast (QSM), a simple reliable multicast platform based on a subset of the architecture presented in Chapter 3, and on the architectural insights we have gained in the process.

4.1 Introduction

4.1.1 Application Scenario

In our prototype implementation, we focus on enterprise computing environments, which may comprise of 10,000s of commodity PCs. Although our vision in general, and the architecture presented in Chapter 3 in particular, are targeted for Internet-scale scenarios, it is clear that the path to broad adoption of the paradigm leads through successful use in corporate settings and on university campus networks, in context of applications such as collaborative editing, video conferencing, online courses, virtual classrooms, in-house television, interactive gaming, or network management tasks such as image and software deployment. Also, such environments permit much higher data rates than what is available in WAN settings, thus raising the bar for a high-performance platform. Our target environment has the following characteristics:

1. The system is used as a part of the live objects runtime, within a managed environment such as .NET or Java/J2EE. Consequently, it is essential to interoperate well with managed applications, and within a runtime environment that maintains a degree of control over aspects such as scheduling or the handling of memory.

2. Reliability is important, and one does need at least a simple form of loss recovery, such as that as long as the data source or a sufficiently large number of copies is present, eventually all nodes that do not leave the protocol receive a copy of each transmitted message. We assume that stronger forms of reliability are used very rarely: multicast is used primarily to access visual content, and properties such as atomicity in the presence of failures, or other strong forms of agreement, are not needed. Where necessary, such functionality can be implemented at higher levels, by leveraging functional compositionality of live objects. Higher-level protocol logic can be implemented using the Properties Language discussed in Chapter 5.

3. Most applications can avoid relying on total ordering, for example by using locking schemes to obtain exclusive right to modify a portion of a shared document, or by assigning ownership of an object to one client at a time, as is the case with most video and audio transmissions, and the sporadic use of this property does not justify the high cost it would incur if it were an integral part of the base protocol. By default, ordering is FIFO, and where necessary, total ordering can be implemented externally, by leveraging the live objects framework and the Properties Language. Likewise, synchronized state transfer can be implemented at the application level: the logic for "catching up" with the group may vary depending on the application.

4. There are 1000s to 10,000s of client nodes, all residing in a single administrative domain with system-wide availability of IP multicast. Our platform must be able to leverage IP multicast and efficiently use the hardware resources.

5. Node crashes and sudden losses of connectivity are uncommon: most clients leave the protocol in a controlled manner, by contacting a membership service (or it is possible to arrange for this to be the case, e.g., by hosting the multicast engine in a system service that cannot be "terminated" simply by closing its window, and

that explicitly leaves all groups when the system is rebooting or shutting down). It is feasible to run a centralized service that tracks all membership requests, such as joining and leaving multicast groups, has full knowledge of the network, and a direct connectivity to all clients.

6. Nodes access a very large number of multicast groups underlying replicated visual elements, such as shared documents, folders, files, live or offline multimedia streams, etc., but most of the traffic is cumulated within a small number of high-bandwidth data streams originating at a relatively small number of sources, carrying data such as television channels, or system images to larger numbers of recipients. Multicast in very interactive groups, such as those underlying collaboratively edited documents, is relatively low-bandwidth, and from a performance standpoint, it can be treated simply as background "noise" that consumes some portion of network bandwidth and CPU resources. Furthermore, the patterns of interest in different objects exhibit certain types of structural regularities, as discussed in Section 4.1.2.

7. Network loss is very uncommon, at the level of 0.1% or less. Packet loss is caused mostly by the overloaded end hosts dropping packets from receive buffers in the operating system, when the multicast engine is denied CPU for a longer period. Consequently, we adopt a reactive approach and optimize for a "stable" scenario. Also, when it does occur, loss has a bursty nature: packets are dropped in sequences, typically about 100 packets long or in windows of time at the order of 10ms, and in an uncorrelated manner. Accordingly, our loss recovery scheme should be optimized for parallel recovery from bursty, uncorrelated losses, and does not need to perform extremely well on networks with random loss or flaky connectivity.

8. Clients are relatively busy because they consume a significant amount of resources

rendering mashups, decompressing multimedia streams, or running other applications, etc., and relatively underprovisioned: most of them are cheap office PCs. At the same time, perturbations on the end hosts are very common, and it is essential for the protocol to operate well in such environment. Client nodes are used for other activities or to run computationally intensive tasks that may trigger bursts of heavy CPU load, cause the multicast platform to be occasionally preempted, or otherwise introduce random, unpredictable processing delays.

4.1.2 Exploiting Overlap

In the preceding section, we stated that although in our target application environment, there may be large numbers of multicast groups underlying replicated objects, a large portion of the bandwidth would be concentrated in a relatively small number of groups, and the "interest" in different objects would exhibit a degree of "regularity". These two assumptions, that bandwidth is concentrated and that there exists a "regularity", reflect our approach to scalability in the number of protocol instances, as explained below.

Recall from our discussion in Section 1.4.3 and the scenario depicted on Figure 1.4, that the main reasons preventing the existing reliable multicast systems from scaling in the number of protocol instances, are related to two factors:

1. Existing protocols use far too many IP multicast groups for dissemination, which leads to state explosion in the networking hardware and triggers software filtering.
2. Existing protocols create recovery structures for different groups independently. When the structures overlap, nodes end up interacting with large numbers of peers.

Accordingly, in designing our system we adopted the following key objectives:

1. We aggregate all multicast traffic within a smaller number of IP multicast groups.
2. We construct recovery structures in a manner coordinated across multiple groups.

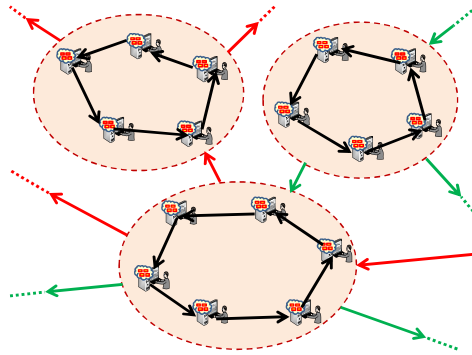

Figure 4.1: A fragment of the network partitioned into three regions. Within each region, nodes run a single local protocol for all groups at the same time to amortize overhead.

Both of the above essentially boil down to amortizing overhead and sharing work across groups. We want to “aggregate” traffic to multiple groups within a single IP multicast channel, or perform recovery in such a way that recovery structures for different groups would create peering relationships between the same pairs of nodes, and so that the same control packets could carry recovery state pertaining to multiple groups at a time.

Our approach is based on the concept of a “region”, a set of nodes within the system that closely cooperate on dissemination and loss recovery. The concept is related to the notion of a “scope” introduced in Chapter 3, but here we go one step further: all nodes in the region coordinate on dissemination and recovery not for just one group, but for all groups simultaneously, and they run a single local protocol instance.

The concept is perhaps best explained by comparing Figure 4.1, showing nodes in a segment of the network that have been partitioned into three regions, with Figure 1.4, where recovery structures for two protocols have been constructed in a completely uncoordinated manner. Note that in the structure shown on Figure 4.1, each node in a region is a part of only a single local protocol (the black lines). We can think of the regions as “meta-nodes”, and the individual system-wide multicast protocols (the green and red lines) as running between such meta-nodes, rather than running between nodes. This is consistent with the object-oriented perspective proposed in Chapter 3.

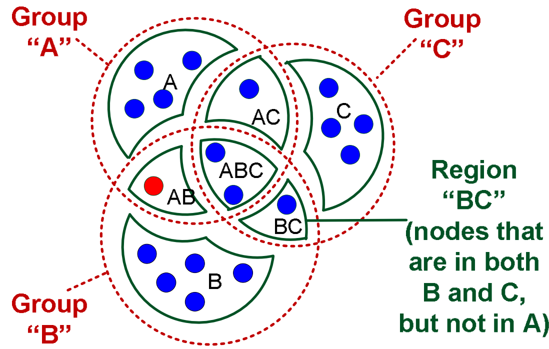

Figure 4.2: Groups overlap to form regions. Nodes belong to the same region if they are members of the same multicast groups. Formally, each node belongs to a single region $R_{G(x)}$, where for any node x, $G(x)$ is a set of groups that node x belongs to, and for any set of groups S, R_S is a set of nodes x such that $G(x) = S$.

Of course, clustering nodes into regions can only bring real benefits from a performance standpoint provided that the following three essential conditions are met:

1. *Nodes in the region have work to share*. If each node in the region is subscribed to a different set of groups, clustering is counterproductive: nodes have no reason to interact with each other, and the local protocol represents overhead.

2. *Regions span at least a few nodes*. In the extreme case, when each node constitutes a single region, our scheme brings no benefit: no useful work sharing takes place.

3. *Nodes in a region belong to more than one group*. If nodes in a region all belong to a single group, no work is shared across multiple protocols.

We can satisfy the first condition by defining a region as a set of nodes with the same membership. Formally, if for a node x, the set $G(x)$ is the set of all multicast groups that node x is a member of, then for any set of groups S, "region R_S" is the set of all nodes x such that $G(x) = S$. We partition the set of all nodes in the system into subsets R_S corresponding to different sets of groups (Figure 4.2). In this scheme, each node x is a part of exactly one region, $R_{G(x)}$. Later in this section, we will relax this assumption.

The fact that all nodes within a region defined this way are members of the same groups makes work sharing trivial: each of these nodes expects to receive exactly the

same messages, hence the recovery protocol can just compare messages between pairs of nodes, irrespective of what groups those messages were transmitted in, and whenever one node is found to have a message that another node is missing, arrange for the two nodes to forward the message to one another. Since all nodes receive the same messages, it does not matter how nodes are organized: they can form a token ring or a tree, and in any case, any pair of neighboring nodes can engage in a useful interaction.

Although the number of all possible regions is exponential with respect to the number of groups, the number of nonempty regions is bounded by the number of nodes, and if certain conditions discussed further in this section are met, this number can be even smaller. In particular, it will be much smaller than the number of groups. This fact will have important implications: because IP multicast addresses in our protocol will be associated with regions rather than with groups, we can avoid the state explosion problem mentioned earlier that plagues systems based on IP multicast.

Now, focusing back on our "three essential conditions" required for clustering to be efficient, we define the system to be *regular* if the remaining two conditions are met. In this dissertation, we will not need a formal notion of regularity. We'll think of "regularity" as a mental shortcut for a class of functions that have the following general properties:

1. A system is more regular if regions are larger.
2. A system is more regular if regions R_S representing overlap of larger sets of groups S are more common.

One example of a possible formal definition of a "regularity" metric ρ satisfying these general properties is the inverse of the number of nonempty regions, formally defined below. Here, the system is represented as a set of regions $R : \mathcal{P}(G) \rightarrow \mathcal{P}(N)$, where for any X, $\mathcal{P}(X)$ is a power set of X, N is the set of nodes, G is the set of groups, and $\forall_{S,S' \in G,\ S \neq S'}\ R(S) \cap R(S') = \emptyset$. In this metric, regularity is the highest, $\rho(R) = 1$,

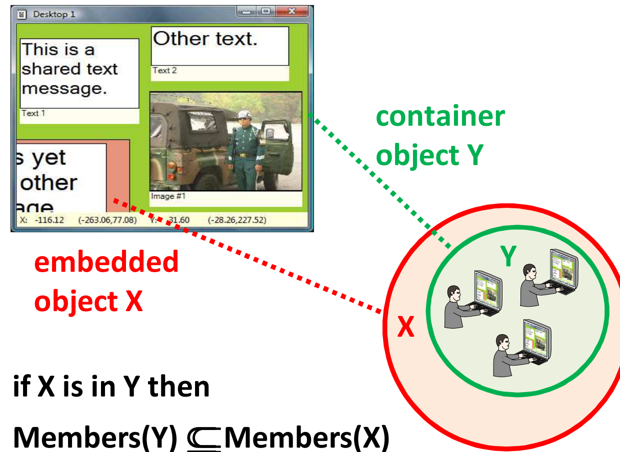

Figure 4.3: Inclusion relation between multicast groups in a mashup. When a live object X is embedded within a container object Y, nodes that access Y also usually access X.

if all groups overlap perfectly on the same set of members, and there exists a single systemwide region, whereas if every node forms a region on its own, the regularity is $\rho(R) = \frac{1}{|N|}$, a value close to 0.

$$\rho(R) = \frac{1}{|\{(g, n) \in R : |n| > 0\}|} \tag{4.1}$$

In the following section, we will describe in detail a protocol that can take advantage of regularity, whereas in the remainder of this section, we will argue that regular systems are not uncommon, and that to a degree, regularity can be arranged for. We will focus here in particular on one class of regularity that arises when groups of nodes running different protocols are included in one another.

First, consider the scenario depicted on Figure 4.3, a client running a “mashup” of the sort common in the live objects platform. Note that when a visual object X, such as a text note or is a video stream, is embedded within a container Y, e.g., a desktop or a shared document, every node that opens the container Y also automatically accesses X. Thus, a group of nodes running the protocol underlying Y is a subset of the group of nodes running the protocol underlying X. Composition of this sort is extremely common in our platform, hence such inclusion relations between different groups are also

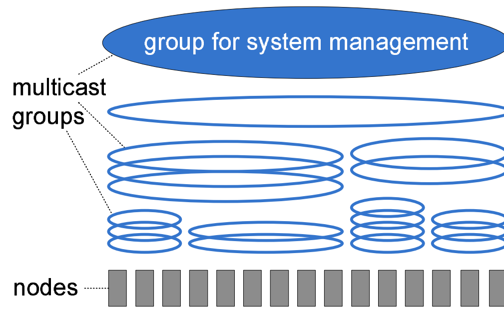

Figure 4.4: In data centers, where replicated components are shared, multicast groups often overlap to form regular hierarchies. The hierarchy shown in this picture is *perfect*, i.e. every pair of groups is either ordered by inclusion or disjoint. In practice, hierarchies are usually not perfect, but one can usually partition the set of all groups in the system into a small number of subsets such that groups in each subset form a perfect hierarchy.

common, leading to hierarchies of groups partially ordered by inclusion. Furthermore, since elements related to the same topic will often be collected on the same mashups, and elements containing unrelated content will rarely be placed together, many groups will have the same or almost the same sets of members.

This type of regularity is also common in data centers, where applications consisting of multiple components are replicated and then deployed within a cluster. If each component needs a group to disseminate updates, the groups overlap because the components are replicated on the same nodes. A hierarchy arises if larger groups are used for control and monitoring purposes, or if larger components are built of smaller ones, much as in the mashup example (Figure 4.4).

It is not hard to see why systems with hierarchical inclusion relationships are more "regular" in the sense proposed earlier. Consider a pair of groups $\{g_1, g_2\}$. In general, a system containing of just those two groups and their nodes could contain up to three regions: $R_{\{g_1\}}$, $R_{\{g_1,g_2\}}$, and $R_{\{g_2\}}$. However, if members of these groups are contained in one another, one of the regions $R_{\{g_1\}}$, $R_{\{g_2\}}$ is empty, and if they are disjoint, $R_{\{g_1,g_2\}}$ is empty. In any case, the number of regions is smaller, hence an average region is bigger. Following this line of reasoning, if in a set of k groups $g_1, \ldots, g_k$, no inclusion relation

is present, the groups could partition the system into up to $2^k - 1$ nonempty regions. One the other hand, if groups are hierarchically ordered by inclusion, the number of such regions is much smaller. In particular, in a *perfect* hierarchy of the sort depicted on Figure 4.4, where every pair of groups is either ordered by inclusion or disjoint, the number of nonempty regions is bounded by $2k - 1$ (compared to $2^k - 1$ in regular case).

Although perfect hierarchies may not be very common, it turns out that in practice, one can very often partition the set G of all groups in the system into a small number of disjoint subsets of groups, $G_1, G_2, \ldots, G_n$, such that when each subset G_k is considered in isolation, groups in this subset form a perfect hierarchy. In another work [300], we propose a simple greedy algorithm that turns out to work extremely well for this purpose:

1. Start with a set of groups G, and an empty set of perfect hierarchies $S = \emptyset$.
2. For as long as $\bigcup S \subset G$, perform the following repetitive task.
   (a) Pick the largest group g in $G \setminus \bigcup S$.
   (b) If there exists $H \in S$, such that $H \cup \{g\}$ would be a perfect hierarchy, then add g to H, otherwise start a new hierarchy $H = \{g\}$ and add it to S.

In addition to the above basic scheme, one can add constraints such as the lower bound on the size of any of the region in a perfect hierarchy: we do not add groups to an existing hierarchy if doing so would cause any of the regions in the resulting hierarchy to become too small. This helps to ensure that the hierarchies we build are not only perfect, but also that the system consisting of those groups is "regular", in the sense defined earlier.

Having partitioned the set of all groups into k perfect hierarchies, we can now simply apply the approach to scalability described earlier k times, for each hierarchy independently. Each hierarchy, considered in isolation, would now define its own set of regions, and each node would now belong to at most k regions, up to one region in each hierarchy.

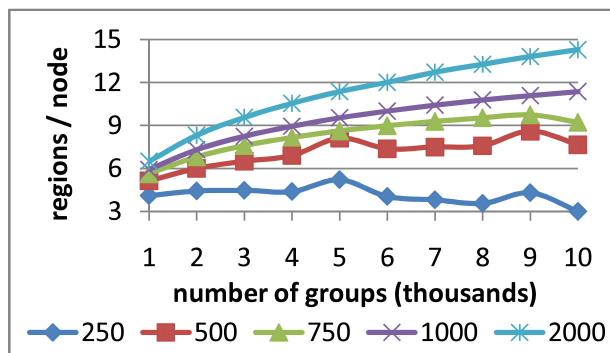

Figure 4.5: A decomposition of an irregular system into perfect hierarchies. Each hierarchy, considered in isolation, defines its own regions. With k hierarchies, a single node can be a member of up to k regions, at most one in each hierarchy. Source: Ken Birman.

We could think of the resulting system as running k independent instances of itself, one instance for each hierarchy managing its own groups, calculating its own regions, and managing traffic in those regions. If the number of groups is large, and the number of hierarchies created this way is kept small, this still represents improvement. Simulation results reported in [300] suggest that in practice, this will very often be the case.

An example of this is shown on Figure 4.5. Here nodes, between 250 and 2000, each join 10% of some set of groups, between 1000 and 10000, using a Zipf popularity distribution with parameter with $a = 1.5$. According to many recent studies, the assumption of Zipf popularity is realistic [119, 193, 263]; even if one views financial trading and RSS feeds as extreme cases, it is likely that live documents would exhibit similar behavior. After running the above algorithm, the average node belongs to between 4 and 14 regions. Additionally, if traffic in the individual groups is also Zipf-distributed, independently of the group popularity, then from the perspective of any particular node, 95% of the traffic it sees is concentrated in just a few of the regions, typically 1-2 (Figure 4.6).

Of course, different nodes will see different "most-loaded" regions, but the implication is that if we have a multicast system that works well for a single regular system, and the network itself is not a bottleneck, as we have assumed earlier, we can just run the platform multiple times, once for each regular hierarchy the system was decomposed

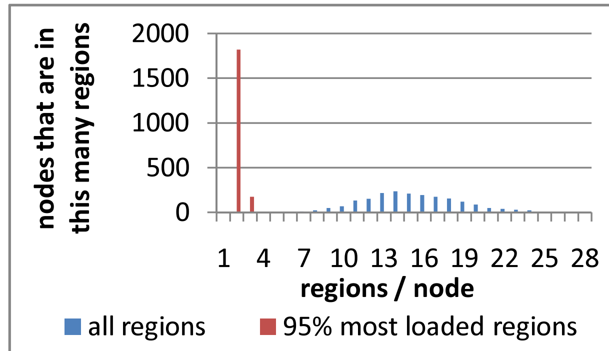

Figure 4.6: In an irregular system that is been decomposed into perfect hierarchies, most traffic seen by a node is concentrated in just 2 of the regions to which it belongs. Here, 2000 processes each joined some 10%, of a set of 10,000 groups. Source: Ken Birman.

into. Resource contention will not be an issue because there will not be many instances of the platform running on the same node, and most of them will be nearly idle. Moreover, since a typical node will find itself in just one or two high traffic regions, we can focus our evaluation and optimization on the behavior of the system in a single heavily loaded, but regular group overlap scenario. If the system does well in this case, it will also do well in systems with irregularly overlapping groups, and our performance and architectural insights will carry over.

Accordingly, in our performance evaluation, we'll focus on micro-benchmarks in a highly regular system, in scenarios with perfect overlap.

4.2 Implementation

4.2.1 Protocol

The protocol used in QSM is essentially a subset of the architecture described in Chapter 3, combined with the approach to scalability described in Section 4.1.2. The protocol is targeting the application scenario discussed in Section 4.1.1. The semantics of the protocol is defined as follows.

- Messages are FIFO-ordered on a per-sender basis, but they are not totally ordered.
- Each message is committed to a group membership view at the time of transmission. The system attempts to deliver the message to all nodes in the view that have not crashed. As long as the sender does not crash, the message is guaranteed to be eventually delivered to the view. If the sender crashes, the message is guaranteed to be delivered provided that it has been cached on at least one recipient within the region that does not crash. No last copy recall guarantees are currently provided across regions if the sender crashes.
- Every message is guaranteed to be eventually cleaned up. The protocol does not block, and if no new messages are sent, it is guaranteed to eventually quiescence.

In the remainder of this section, we describe in detail the membership infrastructure, and the design of dissemination and reliability layers.

A. Membership. The system is managed by a centralized Global Membership Service (GMS), an equivalent of a scope manager in Chapter 3. Nodes contact GMS to join or leave groups, and GMS monitors their health through low-volume periodic heartbeat messages, so that at all times, the GMS has a complete and up to date view of which nodes are running, and what groups they wish to be members of. Based on this information, for each group $g \in G$ the GMS maintains a sequence of *group views*, i.e. a sequence of the form $V_g : \mathbb{N} \rightarrow P(N)$ that represents the subsequent versions of the set of nodes in the group. The GMS batches join and leave requests, and failure reports, and if it detects changes, it periodically applies batches of requests to the current membership views, thus producing new views for those of the groups that gained or lost members.

The GMS also partitions the system into regions, such as those depicted in Figure 4.2, and for each region r, maintains a sequence of *region views*, defined similarly as in the case of groups views, except that nodes do not join or leave region views directly, but rather they are moved between the views by the GMS automatically, as they join or

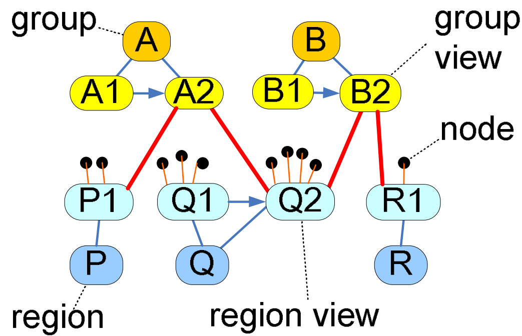

Figure 4.7: A two-level membership service used in QSM, which maintains a sequence of views for each group and each region in the system, and a mapping from group views to region views, the relevant parts of which are delivered to the interested parties.

leave groups. The GMS maintains a mapping from group views to region views. Each group view V in this mapping is mapped to a list L of disjoint region views such that together they contain the full list of members of the group view, i.e. $\bigcup L = V$ (Figure 4.7). As nodes join and leave, the entire structure consisting of group and region views, and the mapping between the two, is periodically updated by applying changes in batches, and relevant parts of it are distributed to the interested clients via incremental updates. Clients use this information to build distributed structures in a consistent manner.

This two-level hierarchy of group and region views is an instantiation of the hierarchical structure described in Section 3.2.10, and in particular, the reader will recognize here the structure used in the example on Figure 3.25 in that section. In the nomenclature of Chapter 3, each region represents a dynamically maintained scope, and the GMS acts as a scope manager for the system as a whole as well as for each individual region.

B. Dissemination. Dissemination is implemented hierarchically, along the lines of what has been described in Section 3.2.3. For each region, the GMS assigns a per-region IP multicast address, reserved only for this particular region, that is used for dissemination within the region. Multicast to a group is then performed by multicasting to each of the regions that the group spans over, using multiple IP multicast transmissions for each message (Figure 4.8). This decomposition of a single logical request to multicast into a

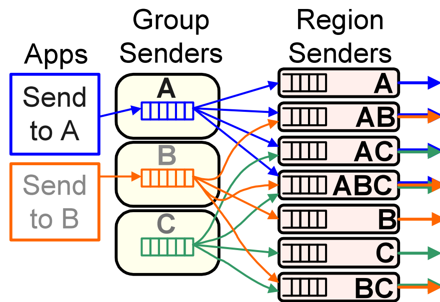

Figure 4.8: One level of indirection for multicast in QSM. To multicast to a group, QSM sends a copy to each of the regions spanned by the multicast group.

group into multiple "sub-requests" to multicast into each of the regions that the group spans across is a special case of the scheme described in Section 3.2.4, and depicted on Figures 3.10, 3.11, and 3.12. The per-region dissemination policies are simply to use IP multicast protocol, and dissemination in each region is done independently of other regions. The per-group dissemination policy is to create a logical "channel" to each of the regions directly. When composed together, these policies yield the scheme we just described. Naturally, the approach could be extended: QSM could use an ALM protocol on a per-region basis to remove dependency on IP multicast, and it could construct an overlay across regions to support scenarios where a single group spans across multiple regions. In this work, however, our ultimate goal was to understand the key performance limitations of our prototype, so we limited ourselves to a scheme based on IP multicast.

At first, replacing a single per-group IP multicast, as in most prior systems, with a series of per-region IP multicasts, could be considered inefficient. However, recall that the existing systems suffer from the state explosion problem mentioned earlier. While their techniques might in theory sound efficient, in practice they fail to recognize the resource limitations of the routing infrastructure. In contrast, in QSM, the number of regions, and hence the number of IP multicast addresses used, can be much smaller than the number of groups; in simulations reported in [300], we found that in large deployments, one could achieve an address "compression" by a factor of 100 or more.

Furthermore, these simulation studies also show that even in irregular systems, after decomposition into perfect hierarchies most high-bandwidth groups would be mapped to a small number of regions, and the number of IP multicast transmissions could be at the order of 3-5, which is competitive with the application-level multicast techniques that construct multicast trees over point-to-point TCP connections.

Nevertheless, to support groups that overlap irregularly with others, QSM provides a form of “backwards compatibility”, a *hybrid mode* of multicast. In the hybrid mode, the GMS may optionally allocate an IP multicast address for a selected group, and disseminate messages in this group to a per-group address. However, the overall protocol remains unchanged, and this optimization is applied only to the initial IP multicast dissemination. What happens here is that, when a message is about to be transmitted into a “hybrid” group, the logical per-group multicast request is still decomposed into per-region requests. Per-region protocol stack components then process the request independently, up to the point when they with to initiate transmission. At this point, rather than multicasting the message to per-region IP multicast addresses, the per-region headers are combined, and a message is transmitted, with a complete set of per-region headers, using a per-group IP multicast address. Upon the receipt of the message, nodes throw away headers for regions they are not part of, and process the remaining message and per-region header in the exact same manner as in the “regular” case. Recovery is also subsequently performed on a per-region basis, and any retransmissions, if necessary, are done using the per-region IP multicast addresses. Overall, this allows a limited number of groups that span across large numbers of regions to be handled more efficiently, but it comes with performance penalty: the added complexity results in a severely degraded performance. In practice, this mechanism would be eventually subsumed by decomposing an irregular system into regular hierarchies, as described in Section 4.1.2.

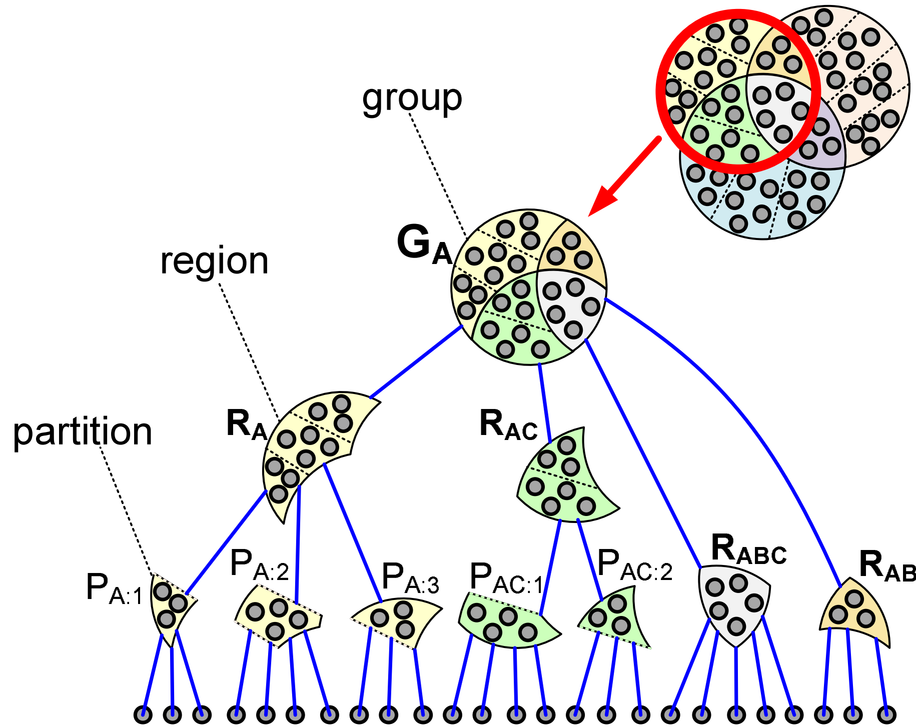

Figure 4.9: An example hierarchy of recovery domains for a group in QSM. Each group, as well as each region and a partition within a region can be thought of as a recovery domain, in the same sense as defined in Section 3.2.9. Recovery domains corresponding to regions can be members of multiple group domains, and their local recovery protocols work for multiple groups simultaneously, just like it was illustrated on Figure 3.25.

C. Recovery. Recovery is also hierarchical, and the implementation follows the general structure described in Sections 3.2.9, 3.2.10, 3.2.11, 3.2.12, and 3.2.13. In the nomenclature of Section 3.2.9, each group can be thought of as a recovery domain. The regions that a group spans over can also be thought of as recovery domains, and are modeled as members of the group's domain, much in the way it was illustrated on Figure 3.25. Note that, just like on Figure 3.25, a region, as a recovery domain, can be a "member" of multiple groups: the recovery protocol running in the region thus "works" for multiple groups simultaneously. For scalability, regions are further subdivided into partitions of a constant size k (typically we set $k = 5$), which can also be thought of as recovery domains on their own. In the end, recovery domains for a given group form a three-level hierarchy of the sort depicted on Figure 4.9.

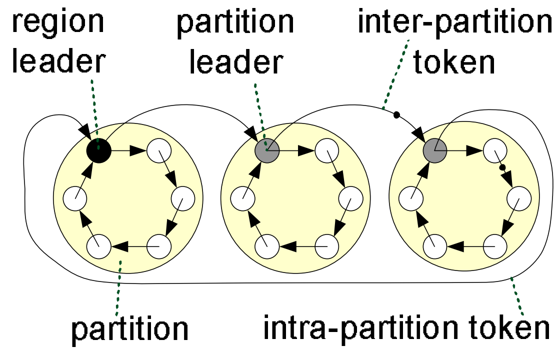

Figure 4.10: Recovery domains are implemented by agents hosted at "leader" nodes, as explained in Section 3.2.12 and Section 3.2.13. Partition and region agents are hosted by partition and region leaders, respectively. Agents in partition and region peer groups aggregate state using a token ring protocol. Agents representing regions do not interact with one another, but instead interact with an agent hosted at the sender. In the resulting hierarchy, the bottom two levels of agents form a two-level hierarchy of token rings, whereas agents at the highest level form a star topology rooted at the sender.

Recovery domains, just as it has been explained in Section 3.2.12 and illustrated on Figure 3.29 and Figure 3.30, are "implemented" by agents, pieces of code deployed on the nodes involved in the procotol that collect information about the domain they "represent", and interact with other agents in their peer group as well as with agents one level up and down the hierarchy. Agents representing partitions are hosted by *partition leaders*, distinguished nodes within the partition. Agents representing regions are hosted by *region leaders*, distinguished nodes within the set of partition leaders in the region. The agent representing the entire group is run by the sender (Figure 4.10).

Partition leaders and nodes within a partition form token rings, which implement local recovery protocols within the partitions and regions, respectively, thus resulting in a two-level token rin hierarchy depicted on Figure 4.10. Region leaders do not interact with one another; instead, they interact with an agent hosted on the sender (Figure 4.11). As an optimization, partition leaders can also interact with the agent hosted on the sender to a limited extent: they can send NAKs directly to it (Figure 4.11). This slightly extends the model proposed in Chapter 3, in the sense that agents can interact not only with their

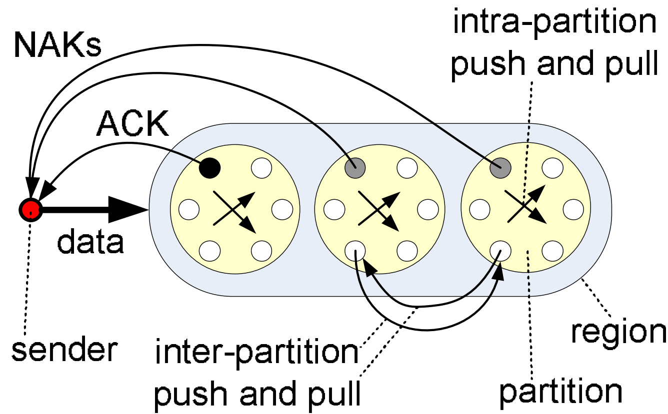

Figure 4.11: Token rings in partitions enable recovery from nearest neighbors; regional rings enable recovery across partitions.

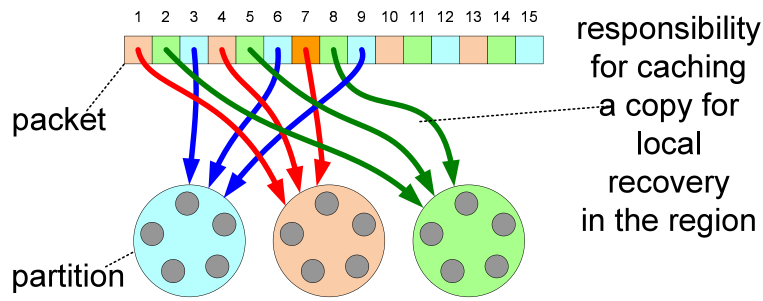

Figure 4.12: Responsibility for caching is subdivided among all partitions in the region.

parents, but also with their "grandparents" in the hierarchy. We found this optimization to be useful in reducing the latency of state aggregation, which, as we shall discuss further in this chapter, turns out to be the key factor affecting scalability in our system.

Another optimization of this sort that slightly deviates from the model is to limit caching packets for recovery purposes to only a single partition within the region (Figure 4.12), similar to a scheme used in [43]. We have introduced this optimization following our findings regarding memory overheads, discussed in Section 4.3.2, and as shown on Figure 4.30, doing this turned out to have a dramatic effect on performance. The way this cooperating caching scheme extends the architecture of Chapter 3 is, by distributing aggregate information about partitions collected by partition leaders across all nodes in the region, and allowing nodes to forward missing packets directly to and from other

partitions, by what we refer to as "inter-partition" push/pull (Figure 4.11). When a long burst of messages is lost by a single node, every partition is involved in recovery, and since specific nodes to recover from are picked at random among those in a partition, as a result, often the entire region is helping to repair the loss at a single node. The efficiency of this technique is especially high in very large regions. Indeed, somewhat contrary to intuition, we found that 50-node regions can achieve higher performance than 5-node regions because in such a large region, caching overhead is spread across a larger number of nodes, and response to a sporadic bursty loss can be much more rapid, and far less disruptive: no single node in the region is burdened by a request to forward a large number of missing packets. This makes isolated losses much less "contagious".

Having explained the high-level dynamics, we now turn to details. We have stated earlier that our goal is to amortize recovery in each region across all groups overlapping on it. This is done thorugh a combination of two features: indexing messages on a per-region basis, and "suspending" protocols that are not active. The former feature ensures that the overhead of the recovery protocol does not depend on the number of groups, but only on the number of different senders. The latter feature extends this further, and ensures that the overhead of recovery in a region is proportional only to the number of senders that are actively multicasting. This way, even if thousands of groups overlap on a region, and thousands of nodes occasionally send streams of messages, but at any given time, only a few of those nodes are transmitting simultaneously, recovery overhead is only a few times larger than it would have been for a single sender and a single group.

Indexing on a per-region basis is done by nodes that are actually transmitting data. Recall from Figure 4.8 that when a message is transmitted, a per-group multicast request is split into multiple per-region requests processed independently. The per-region elements of the protocol stack on each node maintain sequence numbers, and order outgoing packets across groups. During the recovery phase, and when interacting with each

other, nodes in each region use only those sender-assigned per-region message sequence numbers, and ignore per-group sequence numbers entirely. Only when a given message is about to be delivered to the application, the per-group sequence numbers are extracted from messages and passed along with the received data. Because all network communication and data structures use only per-region sequence numbers, recovery overhead within a region R_S is independent of the number $|S|$ of groups overlapping on it.

This overhead depends on the number of senders, and it is manifested mostly by the size of the circulating token, whereas the number of network packets circulating in a region is constant: the system is configured to circulate tokens at a fixed rate, by having the region leader release a new token every Δ_R seconds. Each token carries multiple *recovery records*, one recovery record for each multicasting sender. Additionally, the protocol does not include in the token recovery records for senders that are not currently multicasting. If during a token round, no recovery activity has been performed for a given sender, all packets from the sender have been acknowledged and cleaned up, and no new packets have been received, the recovery record is marked by the region leader as quiescent. During the following token round, all nodes can see that the record is marked as quiescent, and reflect this in their local data structures. If during this second round, still no new messages have been received from the sender, the record is no longer included in the token in the subsequence phases. At the same time, when any of the nodes receives a message, and finds that the recovery record has been previously marked as quiescent, it spontaneously reinserts the recovery record during the next token round. Thanks to the above scheme, token size is proportional only to the number of actively multicasting senders. Depending on this number, as well as factors such as the intensity of traffic or the level of network loss, the token usually occupies between a few hundred bytes and a few kilobytes in size. This overhead is extremely low; in the experiments reported here, we typically use $\Delta_R = 1$. This means that each node in the re-

gion receives, updates, and forwards a token containing a few hundreds of bytes of data about once per second. Cumulative ACK and NAK reports transmitted to the sender are generated by the region leaders at the same rate, and are even smaller, for they only contain an ACK. Our protocol thus operates with an extremely low amount of feedback, and is very nondisruptive to the sender. Reducing the feedback was a deliberate design decision, aimed at improving performance. To a degree, this decision was a successful one, although as we'll demonstrate in Section 4.3, it also has negative consequences: reducing the intensity of feedback resulted in a higher latency that impacts performance indirectly, by slowing down cleanup and elevating memory-related overheads.

4.2.2 Protocol Stack

As mentioned previously, the ultimate purpose of QSM is, to serve as one of the building blocks that provide communication facilities within our live objects platform. The power of the model presented in Chapter 2 comes largely from its dynamic typing and reflection mechanisms. These are only practically feasible within managed runtime environments, such as Java or .NET. Thus, we needed a high-performance multicast substrate that forms an integral part of a managed runtime. Consequently, we have implemented QSM as a .NET library; almost entirely in C#, except only a small portion of about 2.5% of code handling event scheduling and asynchronous I/O using completion ports, which was implemented in managed C++.

In the course of our implementation, we have found prominent features of managed environments, such as garbage collection and multithreading, to have surprisingly strong performance implications. We report on our experiences and explain the significance of our design choices in Section 4.3. Here, we just outline the architecture we settled upon.

QSM is single-threaded and purely event-driven. A dedicated *core thread* communicates with its environment using three event queues (Figure 4.13); these include:

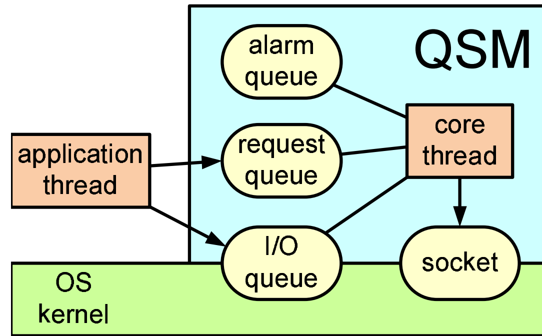

Figure 4.13: QSM *core thread* controls three event queues: an *I/O queue*, implemented as a Windows completion port, that stores system I/O events, such as reports of completed or failed transmissions, an *alarm queue*, implemented as a splay tree, that stores timer events used internally by elements of our protocol stack, and a nonblocking *request queue*, implemented with CAS-style operations, to interact with application threads.

1. *I/O queue*, implemented using a Windows I/O completion port, that collects asynchronous I/O completion notifications from the operating system for all network sockets and files used by QSM, including notifications of received UDP packets, and notifications of completed or failed send operations.

2. *Alarm queue*, based on a splay tree, used internally to store timer events scheduled by elements of the QSM protocol stack.

3. *Request queue*, implemented with nonblocking CAS-style operations, used by the core thread to communicate with application threads without running the risk of being blocked on a synchronization object, which could severely degrade performance. In a system where a single thread is used to handle all kinds of events, it is essential that all event handlers terminate quickly.

The core thread polls its three queues in a round-robin fashion, and processes events of the same type in batches (Figure 4.14), up to the limit determined by its quantum: 50 ms for events from the I/O queue, 5 ms for events from the alarm queue; and no limit for events from the request queue. Additionally, if an incoming packet is found on any socket, the socket is drained of all I/O to reduce the risk of packet loss. Also, for all I/O

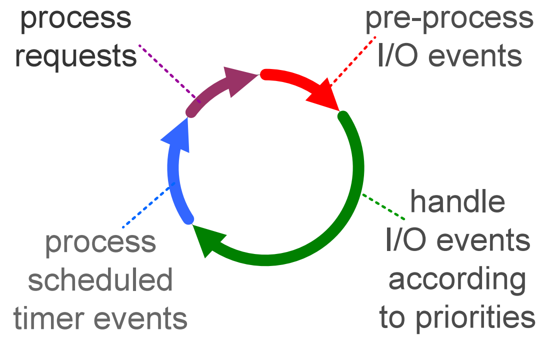

Figure 4.14: QSM uses time-sharing, with fixed quanta per event type. I/O events are handled in 2 stages, much like interrupts in an operating system: first, events are read from the operating system structures into QSM's internal priority queues, and then, in a subsequent phase, all events in these queues are processed in the priority order.

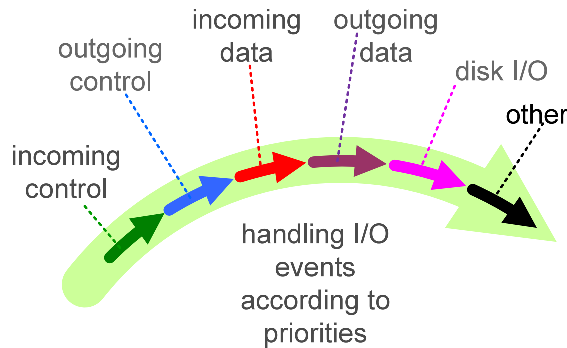

Figure 4.15: QSM assigns priorities to types of I/O events: control packets or inbound network I/O are handled more urgently.

events, QSM prioritizes their processing in a manner reminiscent of interrupt handling: first, all events are read from the completion port, to move them from the operating system structures into a set of 6 internal priority queues. Then, events in those queues are processed in the order of decreasing priorities (Figure 4.15). Priorities are assigned based on the following criteria:

1. Network I/O is prioritized over disk I/O
2. Inbound I/O is prioritized over outbound I/O to reduce packet loss and avoid contention.
3. Control and recovery packets are prioritized over regular multicast, to reduce delays in reacting to packet loss.

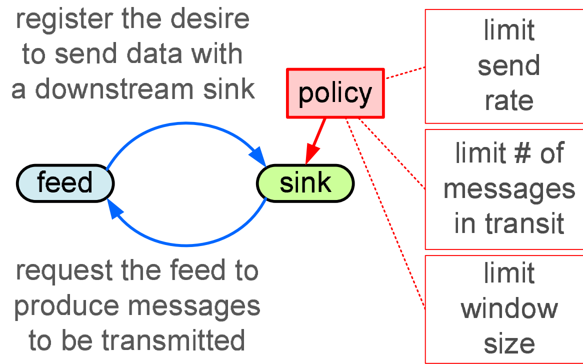

Figure 4.16: In our pull architecture, data to send is produced in a lazy fashion. Rather than creating a message and pushin it down the protocol stack, we simply register the intent to send a message, and delay its creation for as long as possible. Only when the downstream protocol stack element is ready to accept a new message is the message created, using the most up to date information possible. At the time of transmission, it also often turns out that sending the message is no longer needed; thus, delaying the transmission often helps to prevent wasting resources.

The pros and cons of using threads in event-oriented systems are hotly debated. In our case, multithreading was not only a source of overhead due to context switches, but more importantly, a cause of instabilities, oscillatory behaviors, and priority inversions due to the random processing order. Eliminating threads and introducing custom scheduling let us take control of this order, and greatly improve the stability of the system. In Section 4.3 we demonstrate that the latency of control traffic, which is affected by the event processing order, is key to minimizing memory overheads, and as a result, it has a serious impact on the overall system performance.

Control latencies were also the key motivation behind another design feature: a pull protocol stack architecture depicted on Figure 4.16. QSM avoids buffering data, control, or recovery messages, and delays their creation until the moment they are about to be transmitted. Our protocol stack is organized into a set of trees rooted at individual sockets (Figure 4.17), and consisting of *feeds* that can produce messages, and *sinks* that can accept them. Feeds register their intent to send with sinks, but do not push the data downstream, and do not create their messages at the time of registering such intent.

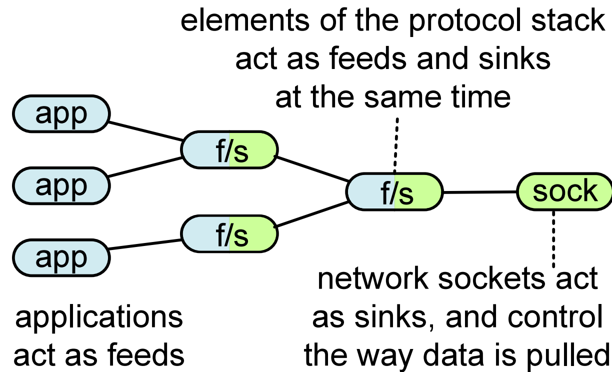

Figure 4.17: Elements of the protocol stack in QSM form trees rooted at sockets. Each socket "pulls" data from the attached tree.

Sinks pull data from registered feeds asynchronously, according to their local rate, concurrency, windows size, and other control policies. Messages to send are created just in time for transmission by the feeds that registered the intent to transmit.

Using this scheme yields two advantages. First, bulky data does not linger in memory and stress a garbage collector. Second, information created just in time for transmission is fresher. ACKs and NAKs become stale rather quickly: if sent after a delay, they often trigger unnecessary recovery or fail to report that data was received. Likewise, recovery packets created upon the receipt of a NAK and stored in buffers are often redundant after a short period: meanwhile, the data may be recovered via other channels. Postponing their creation prevents QSM from doing useless work.

4.3 Evaluation

Evaluation of QSM could pursue many directions, but the primary purpose of building this prototype was to understand the factors affecting performance of such systems, and their architectural implications. Accordingly, we focused on experiments in which QSM is configured to operate at the highest possible data rates, and with very large numbers of nodes or very large numbers of groups, and in the presence of various destabilizing

factors, such as bursty losses or node freeze-ups. The scenarios we evaluate may not necessarily be realistic, since their purpose is to stress the system in a certain dimension, to observe the way it responds and understand the major forces at play, particularly in the context of interactions between our protocol and a managed runtime environment. And, we have succeeded in this regard: by controlling factors such as event scheduling and memory consumption, our system achieves excellent scalability. It can saturate our communication network with very modest CPU loads, it tolerates perturbations well and automatically stabilizes itself even after the most disruptive events, and we see only minor degradation as a function of the number of nodes (Figure 4.18) or the number of groups (Figure 4.46). In what follows, we will explain the origins of these slowdowns, but before we get into the details, it may be helpful to summarize our findings.

The experiments we report reveal a pattern: in each scenario, the performance of QSM is ultimately limited by overheads associated with memory management in the .NET runtime environment. Basically, the more memory in use, the higher the overheads of the memory management subsystem and the more CPU time it consumes, leaving less time for QSM to run. These are not just garbage collection costs: every aspect of memory management gets expensive, and the costs grow linearly in the amount of memory in use. When QSM runs flat-out, CPU cycles are a precious commodity. Thus, in addition to optimizing our code to minimize its direct CPU consumption, minimizing the memory footprint and hence the indirect CPU costs were the key to high performance.

These findings are not specific to .NET and its CLR. While managed environments such as the .NET CLR do have overheads, we believe the phenomena we are observing are universal. An application with large amounts of buffered data may incur high context switching and paging delays, and even minor tasks become costly as data structures get large. We will see that memory-related overheads can be amplified in distributed

protocols, manifesting as high latency when nodes interact. Since traditional protocol suites buffer messages aggressively, existing multicast systems certainly exhibit such problems, no matter what language they are coded in or what platform hosts them. The mechanisms QSM uses to reduce memory consumption, such as event prioritization, pull protocol stacks, and cooperative caching, should therefore be broadly useful.

The structure of this section is as follows.

- In Section 4.3.1, we show that performance degradation in a scenario with large numbers of nodes is indirectly linked to the growing latency of multicast state aggregation through memory overhead on the sender.
- In Section 4.3.2, we show that similar phenomena occur at the receivers, and that latency itself can be affected by the overheads it causes.
- In Section 4.3.3, we show that in scenarios with perturbations, the mechanisms we identified in Section 4.3.1 and Section 4.3.2 are still dominant factors affecting performance.
- In Section 4.3.4, we confirm that this is true even if the system is not saturated.
- In Section 4.3.5 we show that not just the system size, but the number of groups can lead to these sorts of overheads.
- In Section 4.3.6, we construct a simple analytical model of the system and extrapolate some of our results to large numbers of nodes.
- In Section 4.3.7, we discuss our findings and provide further insight into the behavior of the system.
- In Section 4.3.8, we generalize our experience into a set of recommendations for a system designer.

All results reported here come from experiments on a 200-node cluster of Pentium III 1.3GHz blades with 512MB memory, connected into a multicast domain with a switched

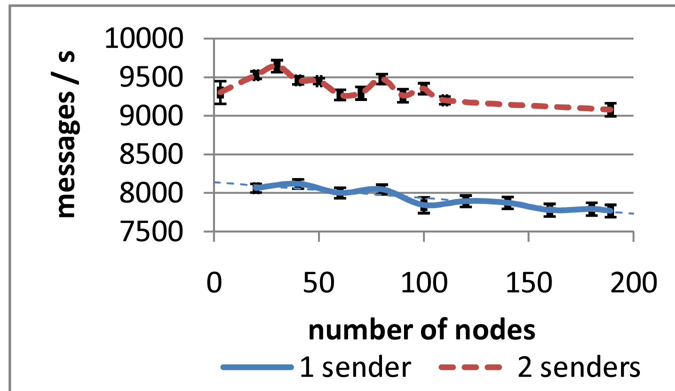

Figure 4.18: Max. sustainable throughput in messages/s as a function of the number of receivers with 1 group and 1KB messages.

100Mbps network. Nodes run Windows Server 2003 with the .NET Framework, v2.0. Our benchmark is an ordinary application, linked to QSM on the same node. Unless otherwise specified, we send 1000-byte arrays, without preallocating them, at the maximum possible rate, and without batching. Nearly all of the figures include 95% confidence intervals, but these intervals are sometimes so small that they may not always be visible.

4.3.1 Memory Overheads on the Sender

We begin by showing that memory overhead at the sender is central to throughput. Figure 4.18 shows throughput in messages/s in two experiments with either 1 or 2 senders multicasting to a varying number of receivers, all of which belong to a single group. With a single sender, no rate limit was used: the sender has more work to do than the receivers and on our clusters, it is not fast enough to saturate the network (Figure 4.19). With two senders, we report the highest combined send rate that the system could sustain without developing backlogs at the senders.

Why does performance decrease with the number of receivers? Let us focus on a 1-sender scenario. Figure 4.19 shows that whereas receivers are not CPU-bound, and loss rates in this experiment (not shown here) are very small, the sender is saturated, and

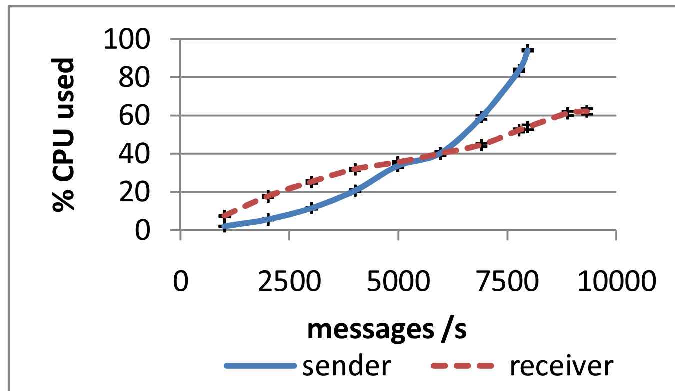

Figure 4.19: CPU utilization as a function of multicast rate, in a group of 100 receivers.

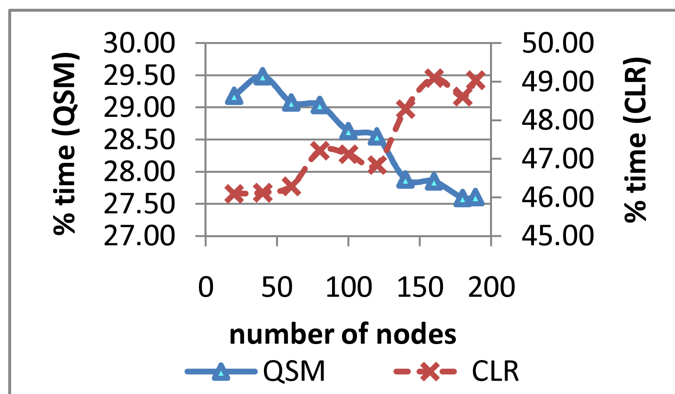

Figure 4.20: The percentages of the profiler samples taken from QSM and CLR DLLs.

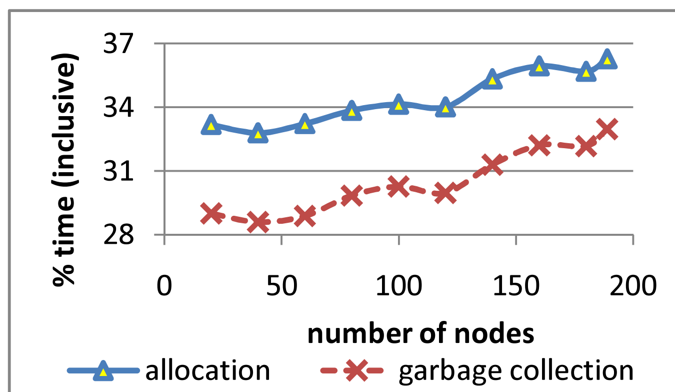

Figure 4.21: Memory overheads on the sender: allocation and garbage collection.

hence is the bottleneck. Running this test again in a profiler reveals that the percentage of time spent in QSM code is decreasing, whereas more and more time is spent in mscorwks.dll, the CLR (Figure 4.20). More detailed analysis (Figure 4.21). makes it clear that

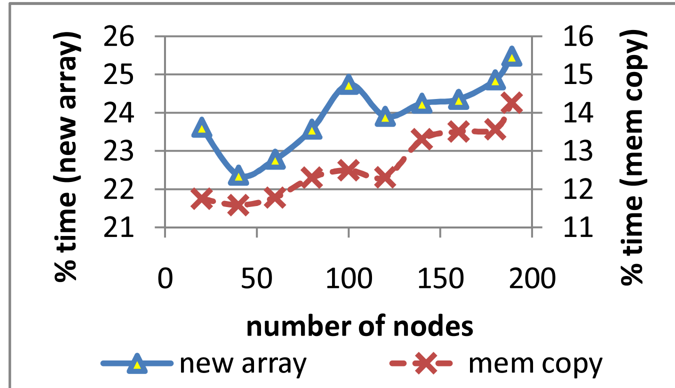

Figure 4.22: Time spent allocating byte arrays in the application, and copying.

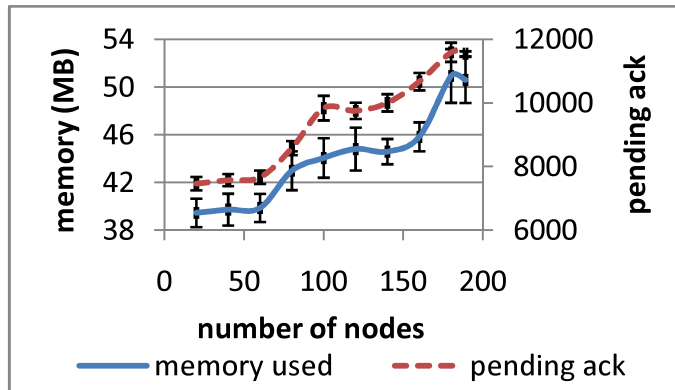

Figure 4.23: Memory used on sender and the # of multicast requests in progress.

the the increasing overhead is a consequence of increasingly costly memory allocation (GCHeap::Alloc) and garbage collection (gc_heap_garbage_collect). The former grows by 10% and the latter by 15%, as compared to 5% decrease of throughput. The bulk of the overhead is the allocation of byte arrays to send in the application ("JIT_NewArr1", (Figure 4.22). Roughly 12-14% of time is spent exclusively on copying memory internally in the CLR ("memcopy"), even though we use scatter-gather I/O.

The increase in the memory allocation overhead and the activity of the garbage collector are caused by the increasing memory usage. This, in turn, reflects an increase of the average number of multicasts pending ACK (Figure 4.23). For each, a copy is kept by the sender for possible loss recovery. Notice that memory consumption grows nearly 3 times faster than the number of messages pending ACK. If we freeze the sender node

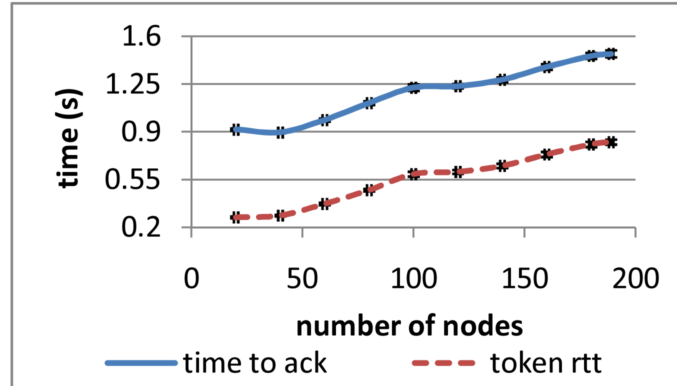

Figure 4.24: Token roundtrip time and an average time to acknowledge a message.

and inspect the contents of the managed heap, we find the number of objects in memory to be more than twice the number of multicasts pending ACK. Although some of these have already been acknowledged, they have not yet been garbage collected.

Thus, acknowledgement latency accounts for the decreasing sender performance. Now, let us shift our focus to the latency: what causes it? The growing amount of unacknowledged data is caused by the increase of the average time to acknowledge a message (Figure 4.24). This grows because of the increasing time to circulate a token around the region for purposes of state aggregation ("roundtrip time"). The time to acknowledge is only slightly higher than the expected 0.5s to wait until the next token round, plus the roundtrip time; as we scale up, however, roundtrip time becomes dominant. These experiments show that the performance-limiting factor is the time needed for to aggregate state over regions. Moreover, they shed light on a mechanism that links latency to throughput, via increased memory consumption and the resulting increase in allocation and garbage collection overheads.

A 500ms increase in latency, resulting in just 10MB more memory, inflates overheads by 10-15%, and degrades throughput by 5%. One way to alleviate the problem we have identified could be to reduce the latency of state aggregation, by using a deeper hierarchy of rings, letting tokens in each of these rings circulate independently. This

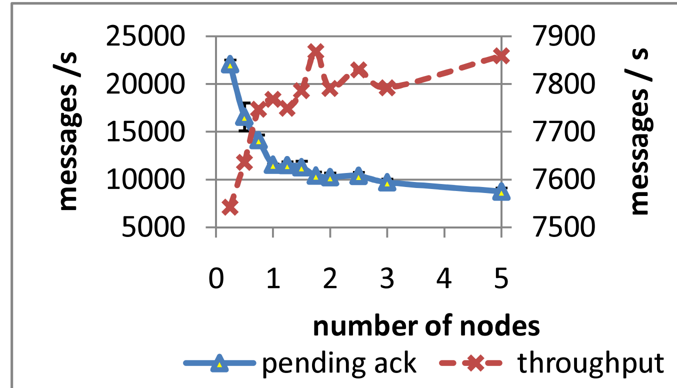

Figure 4.25: Varying token circulation rate.

would create a more complex structure, but aggregation latency would grow logarithmically rather than linearly. But is reducing state aggregation latency the only option? Of two alternative approaches we evaluated, neither could substitute for lowering the latency of the state aggregation.

Our first approach varies the rate of aggregation by increasing the rate at which tokens are released (Figure 4.25). This helps only up to a point. Beyond 1.25 tokens/s, more than one aggregation is underway at a time, and successive tokens perform redundant work. Worse, processing all these tokens is CPU-costly. Changing the default 1 token/s to 5 tokens/s decreases the amount of unacknowledged data by 30%, but increases throughput by less than 1%.

Our second approach increased the amount of feedback to the sender. In our base implementation, each aggregate ACK contains a single value *MaxContiguous*, representing the maximum number such that messages with this and all lower numbers are stable in the region. To increase the amount of feedback, we permit ACK to contain up to k numeric ranges, $(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)$. The system can now clean-up message sequences that have k gaps.

In the experiment shown in Figure 4.26 and Figure 4.27, we set the number of ranges proportional to the number of nodes. Unfortunately, while the amount of ac-

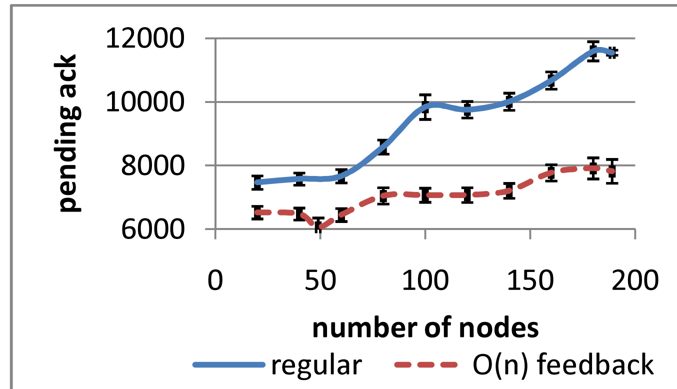

Figure 4.26: More aggressive cleanup with O(n) feedback in the token and in ACKs.

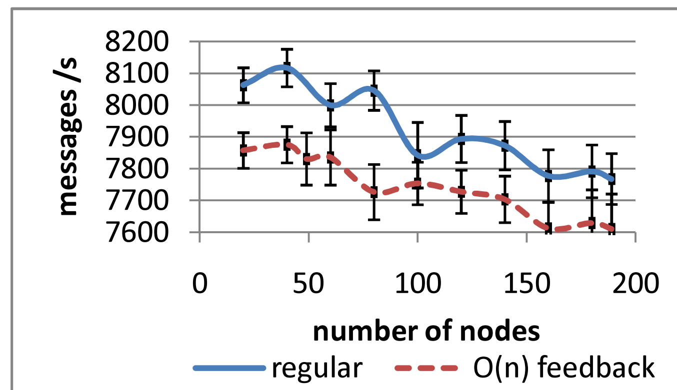

Figure 4.27: More work with O(n) feedback and lower rate despite saving on memory.

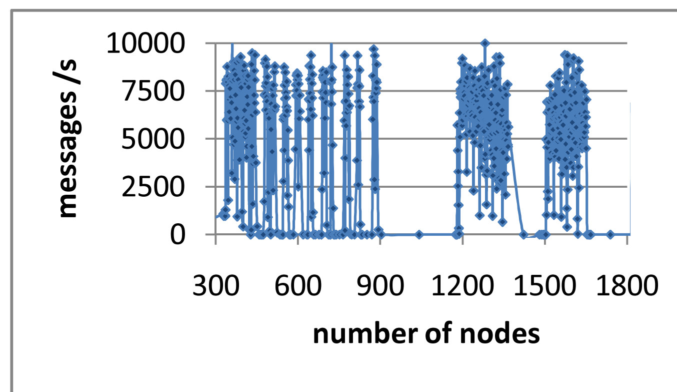

Figure 4.28: Instability with O(n) feedback.

knowledged data is reduced by 30%, it still grows, and the overall throughput is actually lower because token processing becomes more costly. Furthermore, the system becomes unstable (notice the large variances in (Figure 4.28). because our flow control scheme,

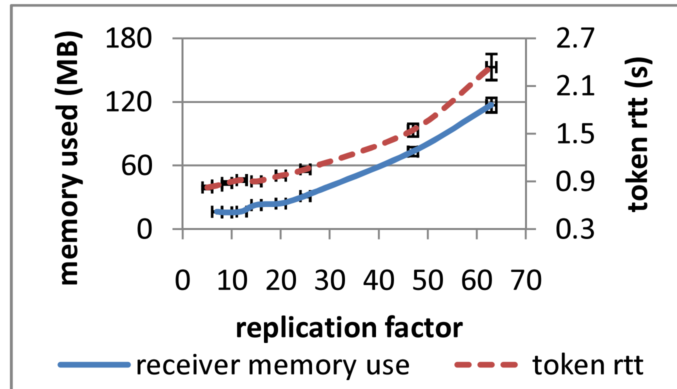

Figure 4.29: Varying the number of caching replicas per message in a 192-node region.

based on limiting the amount of unacknowledged data, breaks down. While the *sender* can now cleanup any portion of the message sequence, *receivers* have to deliver in FIFO order. The amount of data they cache is larger, and this reduces their ability to accept incoming traffic.

4.3.2 Memory Overheads on the Receiver

The growth in cached data at the receivers repeats the pattern of performance linked to memory. The pattern is similar to what we saw earlier: stress that causes the amount of the buffered data to grow, on any node, is enough to slow everything down.

The reader may doubt that memory overhead on receivers is the real issue, considering that their CPUs are half-idle (Figure 4.29). Can increasing memory consumption affect a half-idle node? To find out, we performed an experiment with 1 sender multicasting to 192 receivers, in which we vary the number of receivers that cache a copy of each message (“replication factor” in Figure 4.29). Increasing this value results in a linear increase of memory usage on receivers. If memory overheads were not a significant issue on half-idle CPUs, we would expect performance to remain unchanged. Instead, we see a dramatic, super-linear increase of the token roundtrip time, a slow increase of the number of messages pending ACK on the sender, and a sharp decrease in throughput

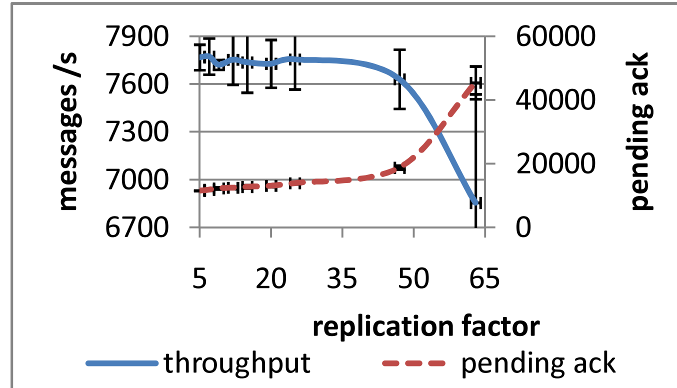

Figure 4.30: As the # of caching replicas increases, the throughput decreases.

(Figure 4.30).

The underlying mechanism is as follows. The increased activity of the garbage collector and allocation overheads slow the system down and processing of the incoming packets and tokens takes more time. Although the effect is not significant when considering a single node in isolation, a token must visit all nodes in a region to aggregate the recovery state, and delays are cumulative. Normally, QSM is configured so that five nodes in each region cache each packet. If half the nodes in a 192-node region cache each packet, token roundtrip time increases 3-fold. This delays state aggregation, increases pending messages and reduces throughput (Figure 4.30). As the replication factor increases, the sender's flow control policy kicks in, and the system goes into a form of the oscillating state we encountered in Figure 4.28: the amount of memory in use at the sender ceases to be a good predictor of the amount of memory in use at receivers, violating what turns out to be an implicit requirement of the flow-control policy.

4.3.3 Overheads in a Perturbed System

Another question to ask is whether our results would be different if the system experienced high loss rates or was otherwise perturbed. To find out, we performed two experiments. In the "sleep" scenario, one of the receivers experiences a periodic, pro-

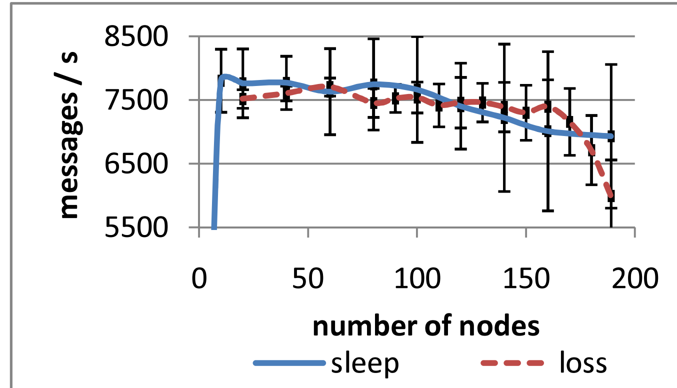

Figure 4.31: Throughput in the experiments with a perturbed node (1 sender, 1 group).

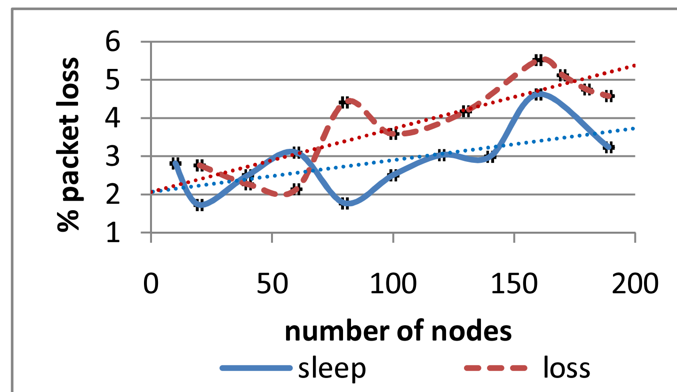

Figure 4.32: Average packet loss observed at the perturbed node.

grammed perturbation: every 5s, QSM instance on the receiver suspends all activity for 0.5s. This simulates the effect of an OS overloaded by disruptive applications. In the "loss" scenario, every 1s the node drops all incoming packets for 10ms, thus simulating 1% bursty packet loss. In practice, the resulting loss rate is even higher, up to 2-5%, because recovery traffic interferes with regular multicast, causing further losses.

In both scenarios, CPU utilization at the receivers is in the 50-60% range and does not grow with system size, but throughput decreases (Figure 4.31). In the sleep scenario, the decrease starts at about 80 nodes and proceeds steadily thereafter. It does not appear to be correlated to the amount of loss, which oscillates at the level of 2-3% (Figure 4.32). In the controlled loss scenario, throughput remains fairly constant, until it falls sharply beyond 160 nodes. Here again, performance does not appear to be directly correlated to

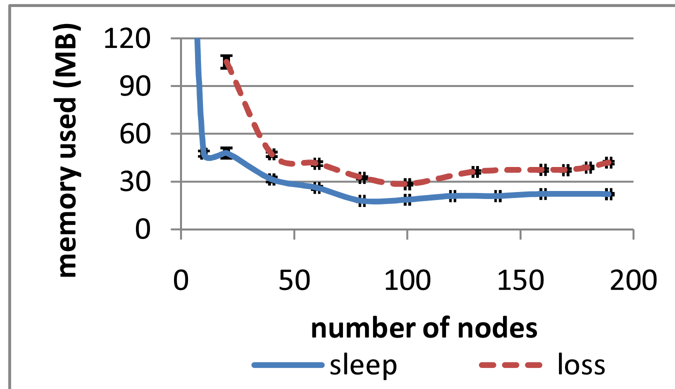

Figure 4.33: Memory usage at a perturbed node (at unperturbed nodes it is similar).

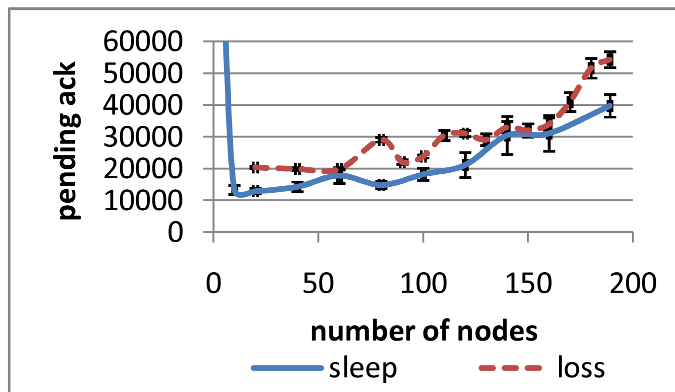

Figure 4.34: Number of messages awaiting ACK in experiments with perturbations.

the observed packet loss. Finally, throughput is uncorrelated with memory use both on the perturbed receiver (Figure 4.33). or other receivers (not shown). Indeed, at scales of up to 80 nodes, memory usage actually decreases, a consequence of the cooperative caching policy described in Section 4.2.2. The shape of the performance curve does, however, correlate closely with the number of unacknowledged requests (Figure 4.34).

We conclude that the drop in performance in these scenarios cannot be explained by correlation with CPU activity, memory, or loss rates at the receivers, but that it does appear correlated to slower cleanup and the resulting memory-related overheads at the sender. The effect is much stronger than in the undisturbed experiments; the number of pending messages starts at a higher level, and grows 6-8 times faster. Token roundtrip time increases 2-fold, and if a failure occurs, it requires 2 token rounds before repair

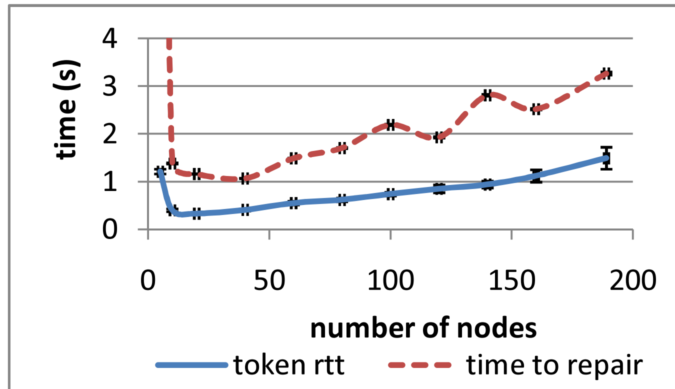

Figure 4.35: Token roundtrip time and the time to recover in the ”sleep” scenario.

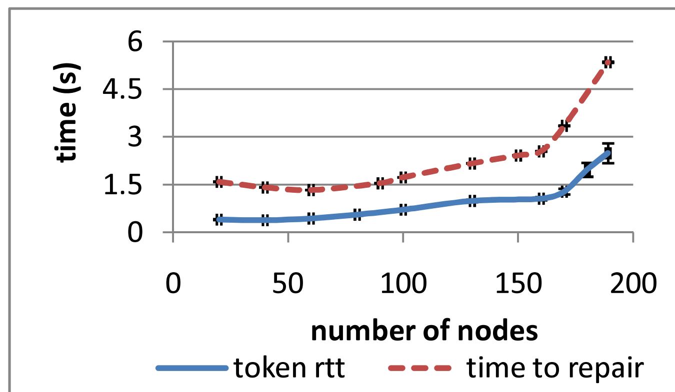

Figure 4.36: Token roundtrip time and the time to recover in the ”loss” scenario.

occurs, and then another round before cleanup takes place (Figure 4.35, Figure 4.36). Combined, these account for the rapid increase in acknowledgement latency.

It is worth noting that the doubled token roundtrip time, as compared to unperturbed experiments, cannot be accounted for by the increase in memory overhead or CPU activity on the receivers, as was the case in experiments where we varied the replication factor. The problem can be traced to a priority inversion. Because of repeated losses, the system maintains a high volume of forwarding traffic. The forwarded messages tend to get ahead of the tokens, both on the sending, and on the receiving path. As a result, tokens are processed with higher latency.

Although it would be hard to precisely measure these delays, measuring alarm (timer event) delays sheds light on the magnitude of the problem. Recall that our time-sharing

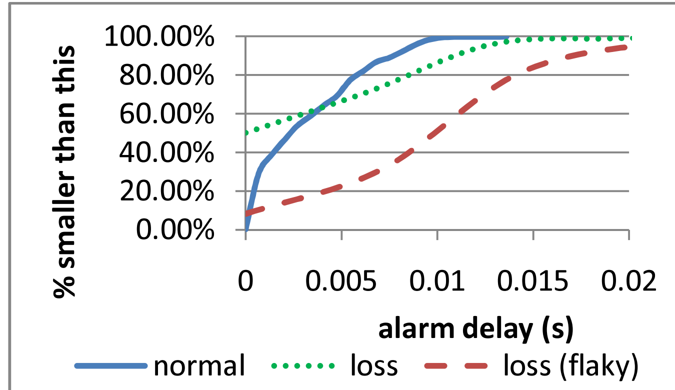

Figure 4.37: Histogram of maximum alarm delays in 1s intervals, on the receivers.

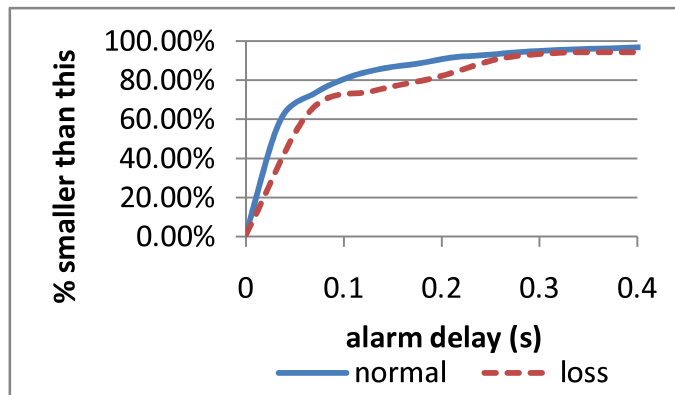

Figure 4.38: Histogram of maximum alarm delays in 1s intervals, on the sender.

policy assigns quanta to different types of events. High volumes of I/O, such as caused by the increased forwarding traffic, will cause QSM to use a larger fraction of its I/O quantum to process I/O events, with the consequence that timers will fire late. This effect is magnified each time QSM is preempted by other processes on the same node or by the garbage collector; such delays are typically shorter than the I/O quantum, yet longer than the alarm quantum, thus causing the alarm, but not the I/O quanta, to expire.

The maximum alarm firing delays taken from samples in 1s intervals are indeed much larger in the perturbed experiments, both on the sender and on the receiver side (Figure 4.37 and Figure 4.38). Large delays are also more frequent (not shown). The maximum delay measured on receivers in the perturbed runs is 130-140ms, as com-

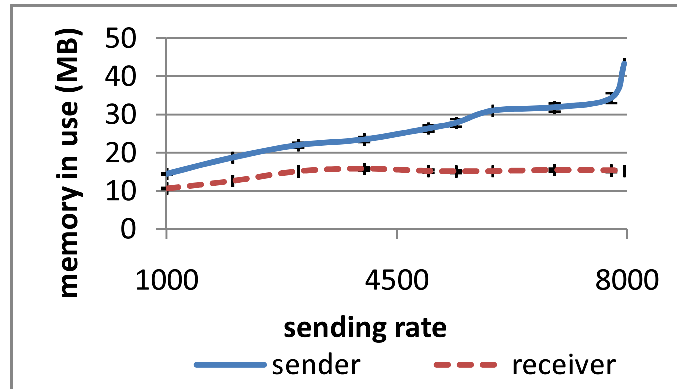

Figure 4.39: Linearly growing memory use on sender and the nearly flat usage on the receiver as a function of the sending rate.

pared in 12-14ms in the unperturbed experiments. On the sender, the value grows from 700ms to 1.3s. In all scenarios, the problem could be alleviated by making our priority scheduling more fine-grained, e.g., varying priorities for control packets, or by assigning priorities to feeds in the sending stack.

4.3.4 Overheads in a Lightly-Loaded System

So far we have focused on scenarios where the system was heavily loaded, with unbounded multicast rates and occasional perturbations. In each case, we traced degraded performance or scheduling delays to memory-related overheads. But how does the system behave when lightly loaded? Do similar phenomena occur? We'll see that load has a super-linear impact on performance. In a nutshell, the growth in memory consumption causes slowdowns that amplify the increased latencies associated with the growth in traffic.

To show this we designed experiments that vary the multicast rate. Figure 4.19 showed that the load on receivers grows roughly linearly, as expected given the linearly increasing load, negligible loss rates and the nearly flat curve of memory consumption (Figure 4.39), the latter reflecting our cooperative caching policy. Load on the

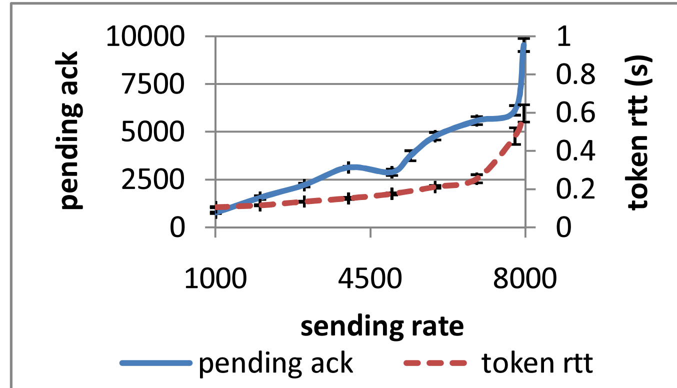

Figure 4.40: Number of unacknowledged messages and average token roundtrip time as a function of the sending rate.

sender, however, grows super-linearly, because the linear growth of traffic, combined with our fixed rate of state aggregation, increases the amount of unacknowledged data (Figure 4.40), increasing memory usage. This triggers higher overheads: for example, the time spent in the garbage collector grows from 50% to 60% (not shown here). Combined with a linear growth of CPU usage due to the increasing volume of traffic, these overheads cause the super-linear growth of CPU overhead shown on Figure 4.19.

The increasing number of unacknowledged requests and the resulting overheads rise sharply at the highest rates because of the increasing token roundtrip time. The issue here is that the amount of I/O to be processed increases, much as in some of the earlier scenarios. This delays tokens as a function of the growing volume of multicast traffic. We confirm the hypothesis by looking at the end-to-end latency (Figure 4.41). Generally, we would expect latency to decrease as the sending rate increases because the system operates more smoothly, avoiding context switching overheads and the extra latencies caused by the small amount of buffering in our protocol stack.

With larger packets once the rate exceeds 6000 packets/s, the latency starts increasing again, due to the longer pipeline at the receive side and other phenomena just mentioned. This is not the case for small packets (also in Figure 4.41); here the load on the system is much smaller. Finally, the above observations are consistent with the sharp

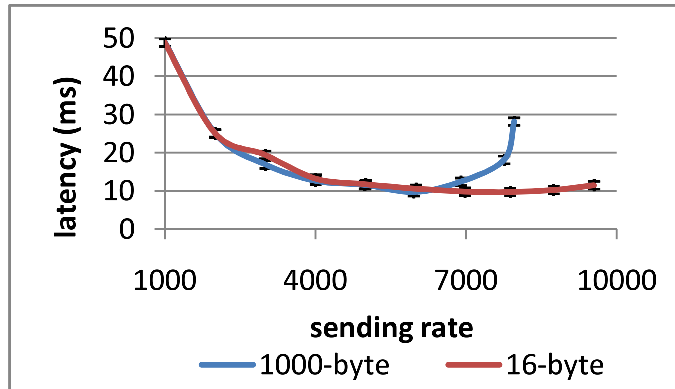

Figure 4.41: The send-to-receive latency for varying rate, with various message sizes.

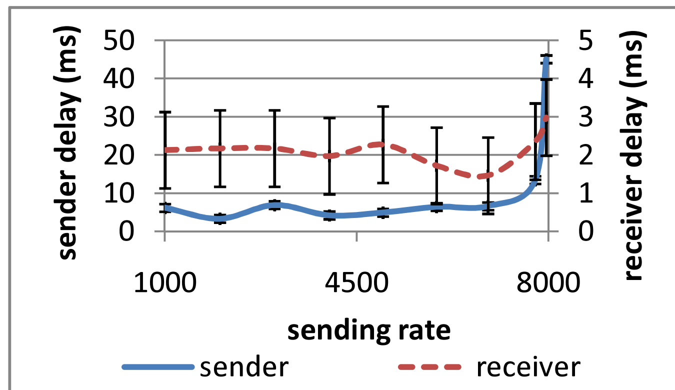

Figure 4.42: Alarm firing delays on sender and receiver as a function of sending rate.

rise of the average delay for timer events (Figure 4.42). As the rate changes from 7000 to 8000, timer delays at the receiver increase from 1.5ms to 3ms, and on the sender, from 7ms to 45ms.

4.3.5 Per-Group Memory Consumption

In our next set of experiments, we explored scalability in the number of groups. A single sender multicasts to a varying number of groups in a round-robin fashion. All receivers join all groups, and since the groups are perfectly overlapped, the system contains a single region. QSM's regional recovery protocol is oblivious to the groups, hence the receivers behave identically no matter how many groups we use. On the other hand,

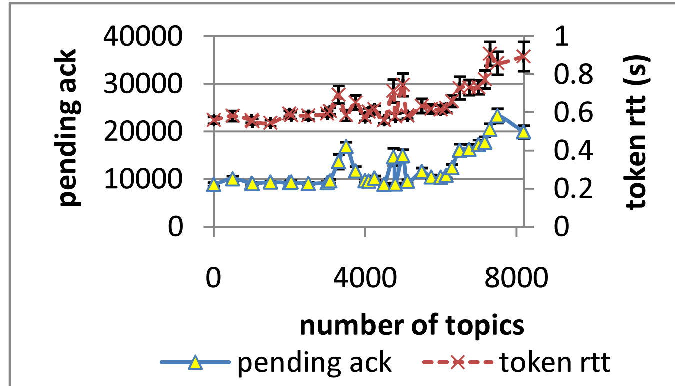

Figure 4.43: Number of messages pending ACK and token roundtrip time as a function of the number of groups.

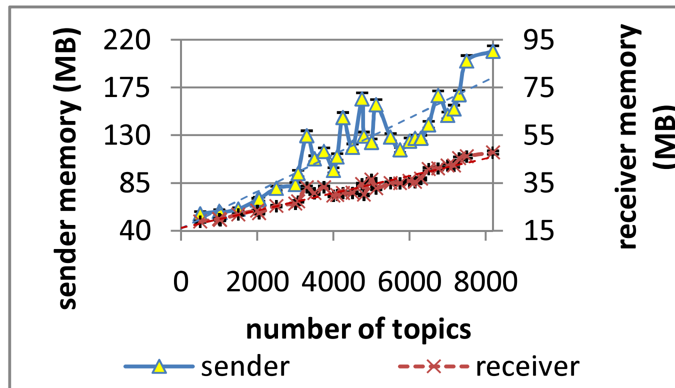

Figure 4.44: Memory usage grows with the # of groups. Beyond a certain threshold, the system becomes increasingly unstable.

the sender maintains a number of per-group data structures. This affects the sender's memory footprint, so we expect the changes to throughput or protocol behavior to be linked to memory usage.

We would not expect the token roundtrip time or the amount of messages pending acknowledgement to vary with the number of groups, and until about 3500 groups this is the case (Figure 4.43). However, in this range memory consumption on the sender grows (Figure 4.44), and so does the time spent in the CLR (Figure 4.45), hurting throughput (Figure 4.46). (Figure 4.46). Inspection of the managed heap in a debugger shows that the growth in memory used is caused not by messages, but by the per-group elements

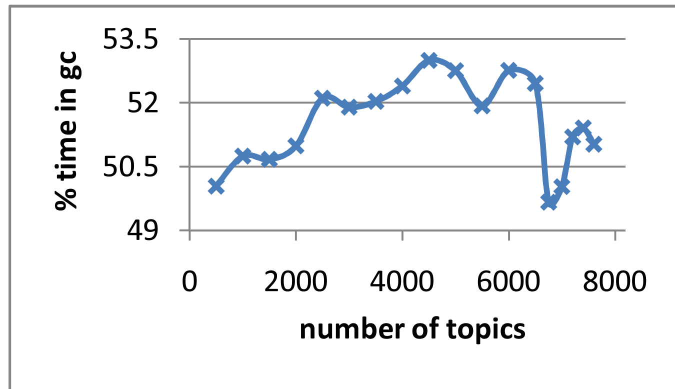

Figure 4.45: Time spent in the CLR code.

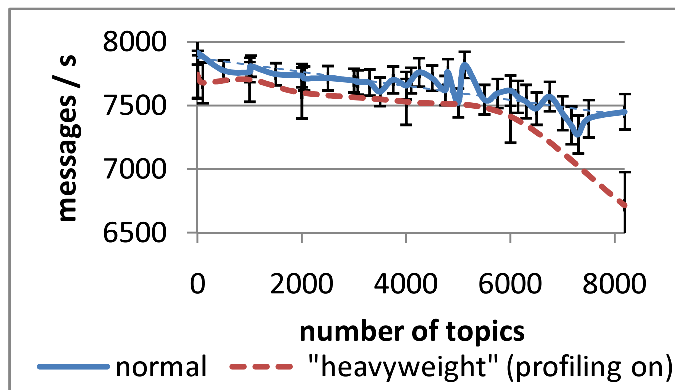

Figure 4.46: Throughput decreases with the number of groups (1 sender, 110 receivers, all groups have the same subscribers).

of the protocol stack. Each maintains a queue, dictionaries, strings, small structures for profiling, etc. With thousands of groups, these add up to tens of megabytes.

We can confirm the hypothesis by turning on additional tracing in the per-group components. This tracing is very lightweight and has no effect on CPU consumption, but it increases the memory footprint by adding additional data structures that are updated once per second, which burdens the GC. As expected, throughput decreases (Figure 4.46, the "heavyweight" scenario as compared to the "normal" one).

It is worth noting that the memory usage reported here are averages. Throughout the experiment, memory usage oscillates, and the peak values are typically 50-100% higher. The nodes on our cluster only have 512MB memory, hence a 100MB average (200MB

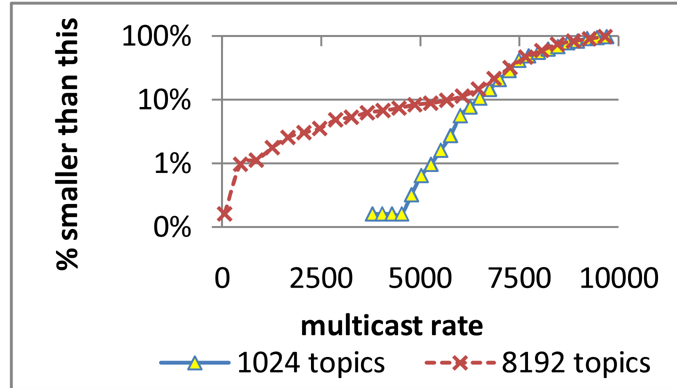

Figure 4.47: Cumulative distribution of the multicast rates for 1K and 8K groups.

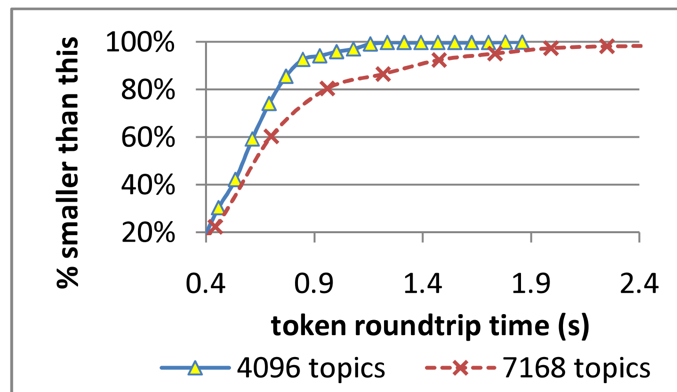

Figure 4.48: Token roundtrip times for 4K and 7K groups (cumulative distribution).

peak) memory footprint is significant. With 8192 groups, peak footprint approaches 360MB, and the system is close to swapping.

Even 3500-4000 groups are enough to trigger signs of instability. Token roundtrip times start to grow, thus delaying message cleanup (Figure 4.47). and increasing memory overhead (Figure 4.48). Although the process is fairly unpredictable (we see spikes and anomalies), we can easily recognize a super-linear trend starting at around 6000 groups. At around this point, we also start to see occasional bursts of packet losses (not shown), often roughly correlated across receivers. Such events trigger bursty recovery overloads, exacerbating the problem.

Stepping back, the key insight is that all these effects originate at the sender node, which is more loaded and less responsive. In fact, detailed analysis of the captured

network traffic shows that the multicast stream in all cases looks basically identical, and hence we cannot attribute token latency or losses to the increased volume of traffic, throughput spikes or longer bursts of data. With more groups, the sender spends more time transmitting at lower rates, but does not produce any faster data bursts than those we observe with smaller numbers of groups (Figure 4.47). Receiver performance indicators such as delays in firing timer event or CPU utilization do not show any noticeable trend. Thus, all roads lead back to the sender, and the main thing affecting the sender is the growing memory footprint.

We have also looked at token round-trip times. The distribution of token roundtrip times for different numbers of groups shows an increase of the token roundtrip time, caused almost entirely by 50% of the tokens that are delayed the most (Figure 4.48), which points to disruptive events as the culprit, rather than a uniform increase of the token processing overhead. And, not surprisingly, we found that these tokens were delayed mostly on the sender.

With many thousands of groups, the average time to travel by one hop from sender to receiver or receiver to sender can grow to nearly 50-90ms, as compared to an average 2ms per hop from receiver to receiver (not shown). Also, the overloaded sender occasionally releases the tokens with a delay, thus introducing irregularity. For 10% of the most-delayed tokens, the value of the delay grows with the number of groups (Figure 4.49). Our old culprit is back: memory-related costs at the sender! To summarize, increasing the number of groups slows the sender, and this cascades to create all sorts of downstream problems that can destabilize the system as a whole.

4.3.6 Extrapolating the Results

In this section, we describe a simple analytical model that captures major relationships between the key parameters and performance metrics in our platform, and we use it to

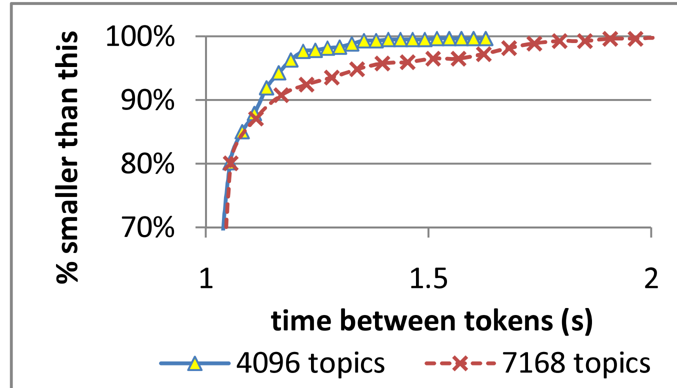

Figure 4.49: Intervals between subsequent tokens (cumulative distribution).

make predictions about the behavior of the system with thousands of nodes.

We will focus on an experiment with a single sender, multicasting messages of a fixed size to a large group of receivers in an unperturbed scenario without losses. We have argued that throughput in this scenario is mostly affected by the amount of memory and the resulting garbage collection and allocation overheads on the sender. We shall now derive the precise form of this relationship.

We start by deriving the set of equations that relate the memory consumption to throughput and latencies. In the first part of our discussion, we shall use the following parameters. Unless otherwise specified, the values of these parameters are averages.

m_s The total amount of memory in use on the sender, in megabytes.

m_0 The amount of memory on the sender, in megabytes, that is occupied by the base system and libraries, and that does not depend on the volume of traffic.

p_s The number of application messages pending cleanup (cached) on the sender.

d_s The size of a message.

c_s The proportion between the number of messages in memory, and the number of messages pending cleanup. This value is typically larger than 1 because messages are not garbage collected immediately.

μ The maximum sustainable throughput in messages/s, with constant-sized, 1000-byte application messages, each message transmitted in a separate packet.

n The number of nodes in the system.

Δ_A Time to acknowledge (and cleanup) a message.

Δ_T The interval between releasing subsequent tokens by the region leader.

Δ_R Token roundtrip time.

Δ_L End-to-end latency, i.e. the time from sending the message to receiving it.

Δ_U A sum of all unaccounted for (but constant) factors that contribute to the acknowledgement latency, but are not captured within the scope of this analytical model.

Δ_1^S Token delay introduced by the sender.

Δ_1^R Token delay introduced by a single receiver.

Now, we'll derive the equations that capture the relationships between these parameters.

The amount of memory m_s in use on the sender is expressed by the following equation. Note that the equation needs to take into account the delay in garbage collection. Each of the p_s pending messages remains in memory approximately c_s times longer than it takes for the protocol to clean it up.

$$m_s = m_0 + p_s \cdot c_s \cdot d_s \tag{4.2}$$

Based on the results presented Section 4.3.1, in particular a linear regression on the data presented in Figure 4.23, we can estimate $m_0 \approx 20.311$ megabytes in our scenario.

The value of parameter c_s can be estimated without polluting the experiment through instrumentation by suspending the sender process in the middle of the transmission, and counting objects on the managed heap using the SOS Debugging Extension [210]. While the number of samples obtained this way is small, the value of c_s typically fits

well within the range between 2 and 3, and is fairly well approximated in our scenario by $c_s \approx 2.5$, throughout the range of configurations.

The number of pending messages can be predicted by the Little's Law [192], where the throughput μ plays the role of the "arrival rate" and time to acknowledge Δ_A plays the role of the "processing time".

$$p_s = \mu \cdot \Delta_A \tag{4.3}$$

The time to acknowledge can be estimated by the following formula. Intuitively, the network latency Δ_L is the time after a message is observed by the receivers. On average, the message will wait $\frac{1}{2} \cdot \Delta_T$ until a new token is released that will collect information about it. The token will take Δ_R to go around the ring. Cleanup is further delayed for additional Δ_U due to various unaccounted for factors that we will not model here.

$$\Delta_A = \Delta_L + \frac{1}{2} \cdot \Delta_T + \Delta_R + \Delta_U \tag{4.4}$$

Based on the data presented in Figure 4.41, the latency can be estimated at $\Delta_L \approx 0.0115$ seconds. The internal between tokens is inversely proportional to the token rate, which in most of our experiments is fixed to 1 token/s, hence $\Delta_T \approx 1$. The sum of factors unaccounted for remains constant, $\Delta_U \approx 0.122$ seconds. The only variable in this equation that depends on the scale of the system is the token roundtrip time Δ_T. This time is the sum of hop by hop delays introduced by the sender and each of the receivers, and can be estimated as follows.

$$\Delta_T = \Delta_1^S + (n-1) \cdot \Delta_1^R \tag{4.5}$$

Based on our measurements, assuming no perturbations, in this experiment we can estimate $\Delta_1^S \approx 0.1914$ seconds and $\Delta_1^R \approx 0.0034$ seconds. The values do not depend to a significant degree on the system size and can be assumed constant, because the sender's CPU is fully saturated, whereas the receiver CPUs are mostly underutilized. Note that the delay introduced by the sender is almost two orders of magnitude larger.

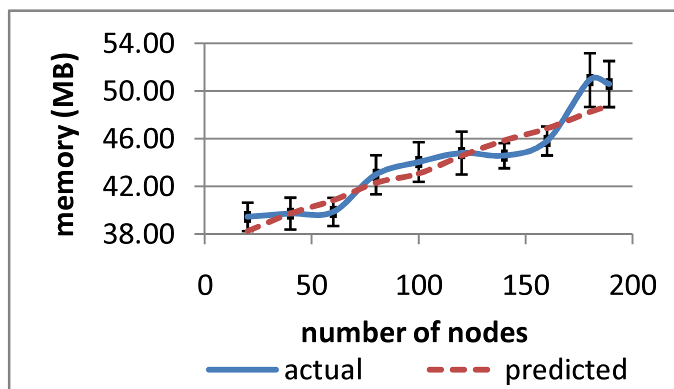

Figure 4.50: Verifying the model: predicted vs. actual memory usage. An estimate of m_s is generated from equations 4.2, 4.3, 4.4, and 4.5, estimates for m_0, c_s, Δ_L, Δ_U, Δ_1^S, and Δ_1^R, and the measured values of μ from the experiment shown on Figure 4.18.

Equations 4.2, 4.3, 4.4, and 4.5, together with the estimates we made for parameters m_0, c_s, Δ_L, Δ_U, Δ_1^S, and Δ_1^R, which can be assumed constant in this scenario, are not yet enough to make useful predictions, but we can use them to verify that the equations correctly reproduce a few of the basic metrics. Figure 4.50 shows the predicated and the actual memory usage on the sender as a function of the number of nodes, calculated from our equations. Because the model is not complete, and lacks the feedback loop to derive μ as a function of the remaining variables, we use the measured values of μ. The predictions closely match the experiment, indicating that for this part of the model, the simple linear equations we proposed and the parameter estimates are sufficient.

In order to close the feedback loop, we need a model for the relationship between the memory usage, processor utilization, and throughput. We shall start with equations based on the data shown in Figure 4.19. First, we introduce a few new parameters.

- θ_s The fraction of the CPU utilized on the sender irrespective of the volume of traffic.
- θ_s The fraction of the CPU utilized on the sender by the multicasting process, including the time consumed by the managed runtime.
- ω_s A cumulative CPU time consumed by all stages of processing for a single message on the sender, including per-message memory overheads such as allocation.

σ_s A cumulative CPU time consumed by all stages of processing for a single megabyte of memory used on the sender, such as garbage collection.

We will assume that the multicasting application and its garbage collector are the only major sources of overhead on the system, and that the total amount of work θ_s performed by the CPU on the sender is a linear function of the rate at which the sender is multicasting and the amount of memory maintained in the process. Paramater ω_s represents the amount of work "contributed" by each message.

$$\theta_s = \theta_0^s + \omega_s \cdot \mu + \sigma_s \cdot m_s \tag{4.6}$$

Of course in practice, this relationship is much more complex, and would depend to a large degree on how the messages are generated and how application is using them, but as we shall demonstrate shortly, the approximation is fairly accurate.

We will also assume that the per-message load on the sender ω_s is a linear function of the amount of the memory in use. Again, this estimate is simplistic, and as we shall demonstrate, it does not accurately describe the behavior of the system in all scenarios, but it will allow us to better uderstand the importance of different parameters. We introduce two new parameters:

α_s The linear scaling factor that relates the amount of CPU time consumed by all stages of processing a single message on the sender, contributed by operations affected by the amount of memory in use, expressed per megabyte of memory.

β_s The linear scaling factor that relates the cumulative CPU time consumed by all stages of processing a single message on the sender, consumed by operations unaffected by the amount of memory in use.

The above definitions are somewhat obscure; the precise meaning of those parameters is perhaps best explained through the following equation:

$$\omega_s = \alpha_s \cdot m_s + \beta_s \tag{4.7}$$

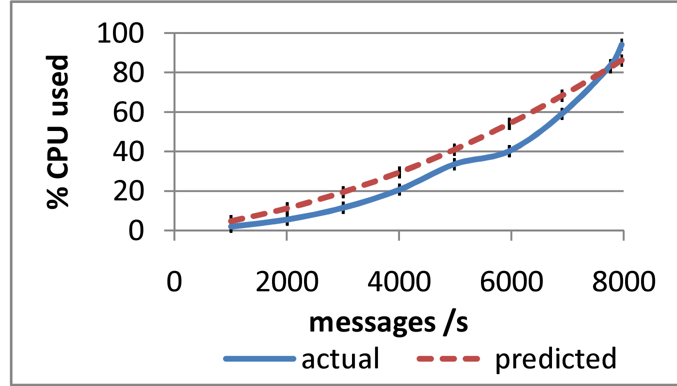

Figure 4.51: Verifying the model: predicted vs. actual CPU utilization on the sender as a function of the sending rate. Equation 4.8 does not fits the data because the underlying equations do not take into account the growth of processing delays of the sort illustrated on Figure 4.40 and Figure 4.42, which are negligible at the lowest rates.

Combined with equations 4.2, 4.3, and 4.6, equation 4.7 yields a quadratic dependency of processor utilization on the sending rate, $\theta_s = \Theta(\mu^2)$, just as we have predicted in Section 4.3.4.

$$\theta_s = \theta_s^0 + (\beta_s + \sigma_s \cdot c_s \cdot d_s \cdot \Delta_A) \cdot \mu + (\alpha_s \cdot c_s \cdot d_s \cdot \Delta_A) \cdot \mu^2 \tag{4.8}$$

By analyzing the data and playing with the model, we found that θ_0^s is negligible and can be ignored, hence we will set $\theta_0^s = 0$. Likewise, the value $\sigma_s \approx 0.003$, corresponding to about 0.3% of processor time per megabyte of memory, consumed by background processing such as garbage collection, appears to fit well with both the model and the data. With 40-50MB of memory in use, this adds up to 12-15%.

As it turns out, Equation 4.8 alone does not explain the shape of the CPU utilization curve. For example, by setting $\alpha_s = 3\mu s$ per megabyte per message, and $\beta_s = 30\mu s$ per message, values we obtained by fitting the model to the data, we obtain predictions that for lower data rates are too high (Figure 4.51). To mimic the shape of the CPU utilization curve, we would need to set much higher values of α, but as we shall argue later, this is unrealistic. The estimate is off because so far, in our calculations we assumed that values Δ_1^S and Δ_1^R, were constant: this was fair because we the highest rates, the sender

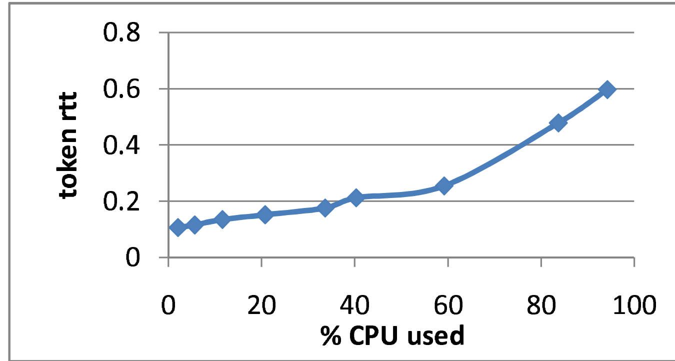

Figure 4.52: Dependency of the token roundtrip time Δ_R on the sender's CPU usage θ_s.

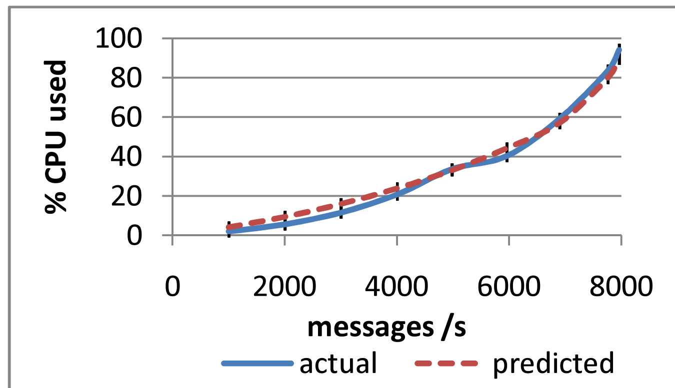

Figure 4.53: Verifying the model: an improved predicted vs. actual CPU utilization on the sender. The shape of the CPU utilization curve is more accurately modeled when processing delays at highest rates are taken into account.

was saturated. However, when the CPU utilization itself varies with sending rate, this assumption no longer holds. As the CPU gets close to full utilization, processing delays of the sort illustrated on Figure 4.40 and Figure 4.42 start to kick in and become the dominant factor, strongly affecting Δ_1^S. The values of Δ_1^S and Δ_1^R, estimated in an experiment under full load, are overestimated for lower throughputs, hence the actual CPU utilization for lower send rates is lower than that predicted by Equation 4.8. At the same time, CPU utilization for the highest loads is overly conservative.

If instead of using the predicted value of Δ_R, we use the measured values of this parameter (Figure 4.52), the CPU utilization curve predicted by Equation 4.8 becomes more realistic (Figure 4.53). Indeed, the quadratic growth we observe here is better

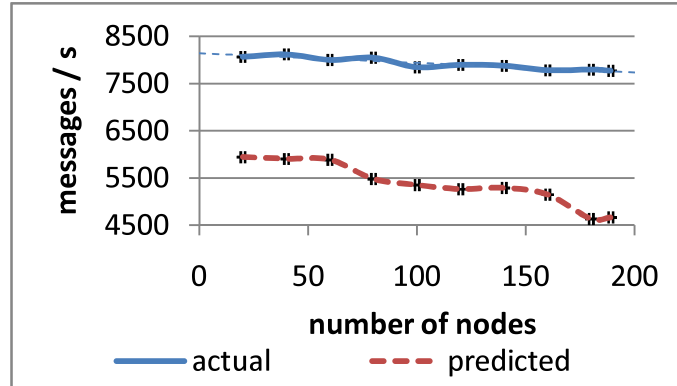

Figure 4.54: A failed throughput prediction: extrapolating the model constructed based on measurements of a system that is not saturated leads to overly pessimistic predictions.

explained by processing delays as the underlying cause than by the dependency of ω_s on the amount of memory in use. Although the model so far can predict the behavior of the system as we vary the send rate, attributing ω_s to memory consumption does not explain the relationship between throughput and system size.

To derive the formula for throughput, we solve equation 4.8 for $\theta_s = 1$, to get the throughput that would saturate the CPU on the sender node. This yields the following.

$$\mu = \frac{1 - \theta_0^s - \sigma_s \cdot m_s}{\alpha_s \cdot m_s + \beta_s} \tag{4.9}$$

As we can see on Figure 4.54, where the values of m_s in the formula have been taken from real measurements, this prediction fails when the system is saturated. Just as we argued before, the model constructed for an unsaturated system is not a good predictor at the highest data rates. It turns out that at the highest data rates, the value of ω_s ceases to depend to a significant degree on the amount of memory in use, the reason being perhaps that as the rate grows, the efficiency of buffer management increases.

If instead, we assume that $\alpha_s = 0$, and fix $\omega_s = \beta_s = 110\mu s$, the model fits the data accurately (Figure 4.55), confirming our theory regarding the role of processing delays as the main reason behind the quadratic dependency of CPU utilization on the send rate.

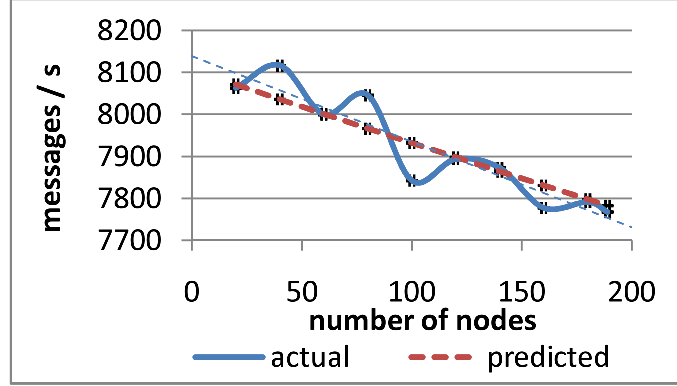

Figure 4.55: A correct throughput prediction: the parameters of the model have been modified to reflect the fact that at the highest data rates, the per-message processing overhead does not depend on the amount of memory used.

In the remainder of the section, we use the model we settled upon to extrapolate results to larger numbers of nodes. We compare predictions made with two parameter settings. The first prediction is the one that fits the throughput measurements, and that we settled upon, with $\omega = 110\mu s$. The second prediction is the one we obtained by extrapolating the model constructed for an unsaturated system, with $\omega_s = 30\mu s + m_s \cdot 3\mu s$.

Both predictions are obtained by solving equation 4.9 for μ. The first prediction is based on the following solution.

$$\mu = \frac{1 - \theta_0^s - \sigma_s \cdot m_0}{\beta_s + c_s \cdot d_s \cdot \sigma_s \cdot \Delta_A} \tag{4.10}$$

The value of Δ_A in this formula is taken from Equation 4.4. The second prediction is based on the following solution:

$$\mu = \frac{-(\alpha_s m_0 + \sigma_s c_s d_s \Delta_A) + \sqrt{(\alpha_s m_0 + \sigma_s c_s d_s \Delta_A)^2 - 4\alpha_s c_s d_s \Delta_A (\sigma_s m_0 - 1)}}{2\alpha_s c_s d_s \Delta_A} \tag{4.11}$$

The two predictions are shown on Figure 4.56. What is striking about this result is that, without taking into account effects such as packet loss, which we would expect to grow linearly with the system size, performance drops significantly due to memory overheads alone. The trend is clearly visible in both of our predictions, and it seems likely that, even if our model does not accurately capture the dynamics of the system, the overall

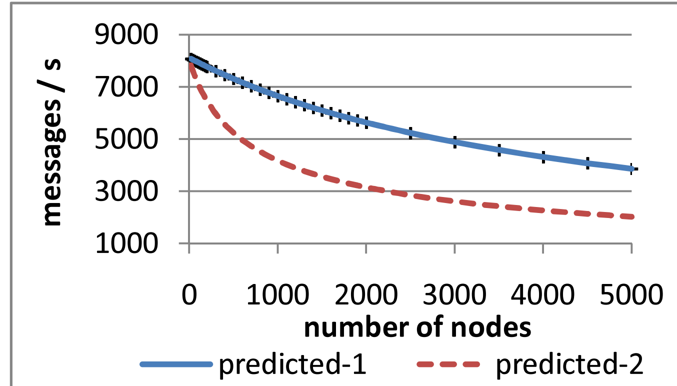

Figure 4.56: Throughput predictions based on Equations 4.10 and 4.11.

shape of the performance curve will remain the same. Indeed, as it turns out, in larger systems, performance in QSM is additionally limited by another factor: the maximum number of pending requests on the sender. Let us introduce yet another parameter.

m_{max} The maximum amount of memory that the system is allowed to consume.

Now, from Equations 4.2 and 4.3, we get the following practical limitation.

$$\mu \leq \frac{m_{max} - m_0}{c_s d_s \Delta_A} \tag{4.12}$$

If we set m_{max} = 100 megabytes and add this equation to our model, the original predictions from Figure 4.56 become even more conservative. This is shown on Figure 4.57.

We could modify our model to account for losses and other factors, but it remain clear that, contrary to popular belief, neither our experiment, nor analytical predictions based on extrapolating the data point to losses as the main reason behind the degradation of performance with scale. With the level of losses we have observed in our experiment, the performance degradation they seem to cause would not become significant before the effect of the memory cap (Equation 4.12) dominates all other factors.

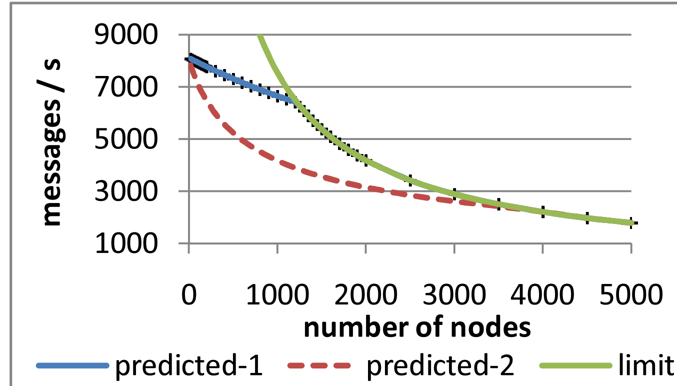

Figure 4.57: Throughput predictions from Figure 4.56 modified to account for memory limit now have become even more conservative. In larger systems, the memory limit is the dominant factor limiting throughput.

4.3.7 Discussion

In the preceding sections we identified some of the factors that affect performance and scalability of QSM. There are a number of forces at play, some of them mutually reinforcing, thus leading to a feedback loop that has a potential to inflate disruptive events caused by losses or busy applications to the level where they hurt performance (Figure 4.58). All these phenomena are ultimately tied to delays and latencies, which act as the common link through which the vicious cycle can sustains itself. The biggest sources of latency, at least in large configurations, are the design of the protocol (which might require multiple hops, rounds, or roundtrips to aggregate state), and what we refer to as "scheduling delays" (which might cause each individual hop or round to take longer). The latter represent the overall, cumulative time "penalty" imposed on important tasks such as processing a token. These delays may come from a variety of sources, some of which have been discussed: for example, a high volume of I/O, or disruptive scheduling. As demonstrated in our experiments, even small delays may be effectively "inflated" by the protocol, thus resulting in high latencies for critical tasks. As our experiments show, even seemingly low-priority tasks, such as collecting acknowledgements, may turn out to be critical and require low latency because of the high memory-related overheads they

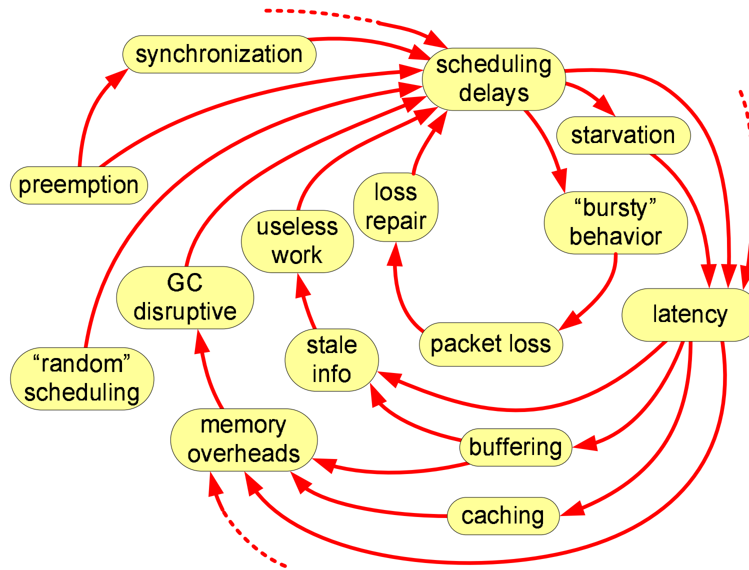

Figure 4.58: A variety of "forces" controlling the behavior of the system form a self-reinforcing "vicious cycle" that has the potential to inflate any temporary perturbations to the level where they hurt performance. All these "forces" are ultimately tied to memory overheads and various sorts of delays and latencies.

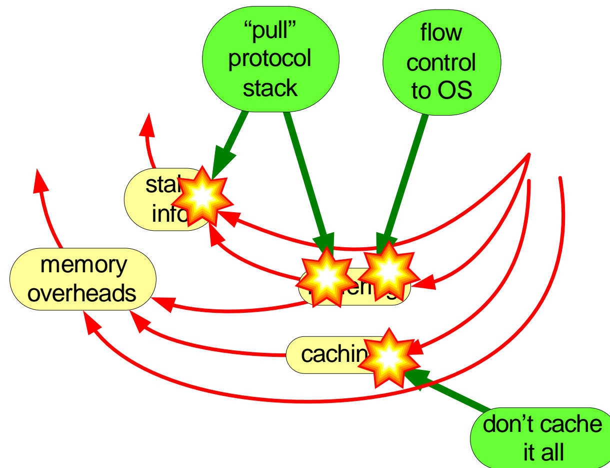

Figure 4.59: Breaking the vicious cycle by controlling memory use. Reducing the buffering and caching overheads and postponing message creation until the time of transmission weakens the strength of memory-related overhead factors.

cause, and the unexpectedly strong impact it has on overall system performance.

Our architectural decisions have been motivated by the need to break the vicious cycle we have just described: memory-related techniques were aimed at decreasing the overheads caused by garbage collection and allocation (Figure 4.59), whereas

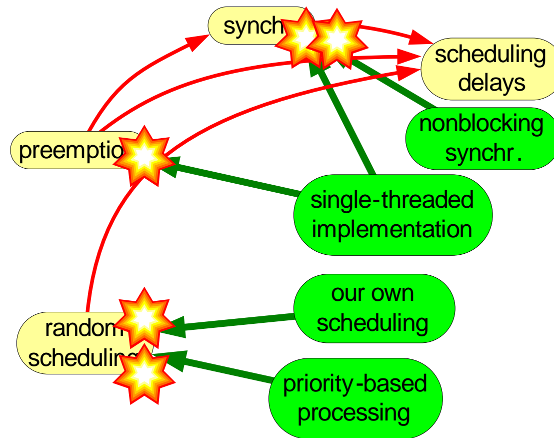

Figure 4.60: Breaking the vicious cycle by controlling scheduling. Eliminating preemption and taking control over event processing order decreases the processing delays inflated by the protocol.

scheduling-related techniques were aimed at decreasing the latency of important phases of the protocol involved in application-level cleanup (Figure 4.60).

Another factor that may have been less pronounced in the results we have presented so far, but that we found equally important for achieving high performance, is the ability of the system to stabilize itself in the presence of minor perturbations, which are common even in isolated experiments on a cluster over which we have complete physical control. It is not unusual for our platform to experience periods lasting tens, and sometimes even hundreds of milliseconds during which our process is preempted, which occasionally causes packet loss. When this happens, as it inevitably does, it is essential for the system to engage in a measured response.

We like to think of QMS as a crowded highway: the faster it runs, and the shorter the inter-message spacing, the higher the chances of an accident and the more severe the consequences. If a system is too conservative in its handling of loss, failures and the flow control, it fails to achieve the highest speeds, but if it is too aggressive, loads can flap from very low to extremely high, causing the kinds of “broadcast storms” that can escalate and eventually shut down an entire data center. Our work suggests that

oscillating throughput has many causes:

1. *Uncontrolled reaction to failures*, for example when a packet is lost by several receivers, stresses the system. The resulting load surge can cause more loss, creating a feedback cycle capable of overwhelming the network (a "broadcast storm"). To avoid such problems QSM does rate-limited recovery triggered (only) by circulating tokens.

2. *Recovery that requires action by a single node*, such as a sender, can trigger a kind of convoy in which many nodes must pause until that one node acts and convoys are contagious because once that node finally acts, other nodes can be overloaded. QSM prevents this via cooperative caching. A burst of losses will often trigger parallel recovery actions by tens of peers.

3. *Jumping the gun* by instantly requesting recovery data on the basis of potentially stale state data can trigger redundant work that reinforces the positive feedback loop mentioned earlier. Our "pull" architecture eliminated this issue entirely: we always act upon fresh information.

4. *Priority inversions* can leave long lists of messages stacked up waiting for a recovery or control packet. Prioritized event handling is needed to prevent this. Control packets are like emergency vehicles: By letting them move faster than regular traffic, QSM can also heal faster.

5. *Reconfiguration* after node joins or failures can destabilize a large system because changes reach different nodes at different times, and structures such as trees or rings can take seconds to form. QSM suspends multicast and recovery on reconfiguration, and briefly buffers "unexpected" messages, in case a join is underway.

By addressing the problems just mentioned, QSM can stabilize itself in the presence of long bursts of loss or when it experiences artificial "outages" (such as on Figure 4.31).

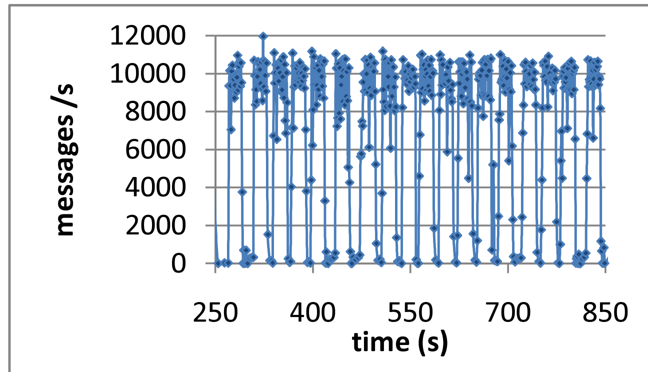

Figure 4.61: Combined send rate oscillates in 30-sec periods in a 110-node group. The maximum load exceeds receiver capacity.

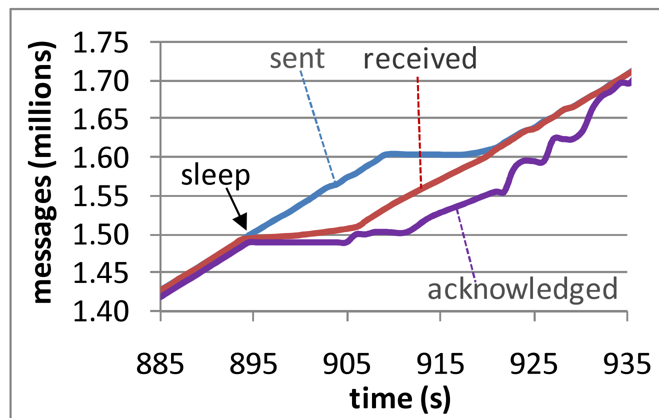

Figure 4.62: An experiment with a “freezing” receiver. The receiver “sleeps” for 10s undetected, causing massive recovery. QSM responds by suppressing multicast.

and tolerates random loss or flaky hardware, responding with reduced throughput. With strong enough perturbation, QSM can still be forced into mild oscillatory behavior. This can be provoked, e.g., by enforcing multicast at a rate exceeding the capacity of the network or of the receivers (Figure 4.61). Similar behavior can be observed with flaky hardware or very disruptive applications that consume CPU.

To explore a massive perturbation, we created a “sleep” scenario (recall Figure 4.31) lasting 10s, causing an 80MB backlog. QSM takes longer time to recover, running recovery at a steady pace, rate controlled, and suppressing multicast until the nodes start to “catch up” (Figure 4.62). Yet even in such extreme cases QSM can stabilize.

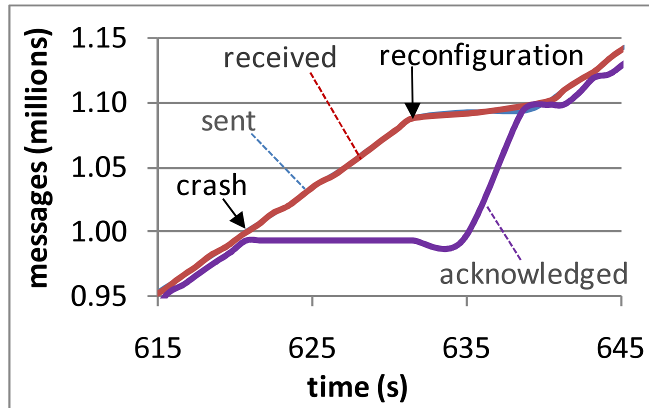

Figure 4.63: On reconfiguration following a crash, changes can take time to propagate. In this case QSM temporarily slows down.

Similarly, a reconfiguration following a crash (Figure 4.63). or join results in slowdown, but the system soon recovers.

4.3.8 Conclusions

Below, we briefly summarize our design insights:

1. *Exploit Structural Regularity*. A key enabler to our approach was the recognition that even irregular group overlap can be reduced to a small number of regularly overlapping groups (cover sets), with most of the traffic concentrated in just a few cover sets. This justified a focus on optimizing the regular case and on the performance of QSM in a single, heavily loaded, cover set.

2. *Minimize the memory footprint*. We expected garbage collection to be costly, but were surprised to realize that when a system has a large memory footprint, the effects are pervasive and subtle. The insight led us to focus on the use of memory throughout our protocols:

   (a) *Pull data*. Most multicast systems accept messages whenever the application layer or the protocols produces them. QSM uses an upcall-driven pull

architecture. We can delay generating a message until the last minute, and avoid situations in which data piles up in the sender's buffers.

(b) *Limit buffering and caching*. Most existing protocols buffer data at many layers and cache data rather casually for recovery purposes. The overall memory footprint becomes huge. QSM avoids buffering and uses distributed, cooperative caching. This limits message replication and spreads the burden evenly, yet allows parallel recovery.

(c) *Clear messages out of the system quickly*. Data paths should have rapid data movement as a key goal, to limit the amount of time packets spend in the send or receive buffers.

(d) *Message flow is not the whole story*. Most multicast protocols are optimized for steady low-latency message flow. To minimize memory usage, QSM sometimes accepts increased end-to-end latency for data, so as to allow a faster flow of control traffic, for faster cleanup and loss recovery.

3. *Minimize delays*. We have already mentioned that the data paths should clear messages quickly, but there are other important forms of delay, too. Most situations in which QSM developed convoy-like behavior or oscillatory throughput can be traced to design decisions that caused scheduling jitter or allowed some form of priority inversion to occur, delaying a crucial message behind a less important one. Implications included the following:

(a) *Event handlers should be short, predictable and terminating*. In building QSM, we struggled to make the behavior of the system as predictable as possible not a trivial task in configurations where hundreds of processes might be multicasting in thousands of overlapping groups. By keeping event handlers short and eliminating the need for locking or preemption, we ob-

tained a more predictable system and were able to eliminate multithreading, with the associated context switching and locking overheads.

(b) *Drain input queues*. We encountered a tension here: from a memory footprint perspective, one might prefer not to pull in a message until QSM can process it. But in a datacenter or cluster, most message loss occurs in the operating system, not on the network, hence loss rates soar if we leave messages in the system buffers for too long.

(c) *Control the event processing order*. In QSM, this involved single-threading, batched asynchronous I/O, and the imposition of an internal event processing prioritization. Small delays add up in large systems: tight control over event processing largely eliminated convoy effects and oscillatory throughput problems.

(d) *Act on fresh state*. Many inefficiencies can be traced to situations in which one node takes action on the basis of stale state information from some other node, triggering redundant retransmissions or other overheads. The pull architecture has the secondary benefit of letting us delay the preparation of status packets until they are about to be transmitted, to minimize the risk of such redundant actions.

4. *Handle disruptions gracefully*. Broadcast storms are triggered when the attempt to recover lost data is itself disruptive, causing convoy effects or triggering bursts of even more packet loss. In addition to the above, QSM employs the following techniques to maintain balance:

   (a) *Limit resources used for recovery*. QSM controls recovery traffic rate and delays the creation of recovery packets to prevent them from overwhelming the system.

(b) *Act proactively on reconfiguration.* Reconfiguration takes time. Slowing down and tolerating overheads, buffering packets from "unknown" sources, and delaying recovery to avoid redundant work is a cost worth paying.

(c) *Balance recovery overhead.* In some protocols, bursty loss triggers a form of thrashing. QSM delays recovery until a message is stable on its caching replicas, then coordinates a parallel recovery in which separate point-to-point retransmissions can be sent concurrently by 10s of nodes.

Chapter 5
Future Work: Properties Framework

The live objects platform described in Chapter 2 makes it very easy to compose different types of protocol objects into applications, but in order for such platform to make a truly big impact, one also needs a library of components to compose from. Most functionality relies on different flavors of replication, such as reliable multicast, replicated state, or agreement, and synchronization, such as locking or ordering. One way we could make developers' lives easier is simply by implementing a large library of live objects to choose from. However, recall from the discussion in Section 1.3 that applications have diverse needs, and run in diverse environments. Offering the developers a small number of tools that do not perfectly match their needs would be to betray the very spirit of this work. Instead, we propose a tool that makes it easier to construct distributed protocols.

In this section, we briefly describe the Properties Framework (PF), a new programming language for building distributed protocols that fits cleanly into the live objects model described in Chapter 2. The Properties Framework transforms concise programs in Properties Language (PL), a new language we have designed, into hierarchical protocols targeting the architecture described in Chapter 3, many elements of which are implemented much in the same way as the system evaluated in Chapter 4. Programs in the Properties Language can express semantics such as consensus, virtual synchrony, or transactions, alongside a variety of weaker models, using sets of 10-30 rules that fit on a single page and have an easy to understand structure. It frees the user from worrying about scalability or explicit handling of failures; both are achieved by construction and through a very careful selection of programming language abstractions.

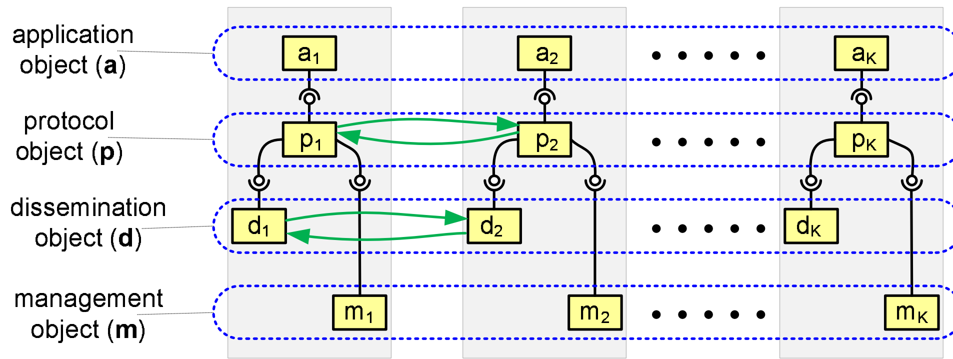

Figure 5.1: Using a protocol generated by PF within the live objects platform. A program in PL is translated into a live "protocol" object p with three types of endpoints. One endpoint connects to an application object a, and allows p to send and receive application-level events. Another endpoint connects it to one or more underlying dissemination objects d, which represent various kinds of event streams. Finally, the third type of endpoints connects p to a management infrastructure m, which provides p with configuration events and runtime context, such as membership views. The "business logic" of p, encoded in a set of rules in PL, is controlling the way p uses these three types of resources available to it, and how it coordinates different categories of events.

5.1 Protocols

Protocols created by the Properties Framework are wrapped as live objects that run in a context similar to that depicted on Figure 5.1. A protocol object p has three kinds of endpoints that connect it to an application object a it is going to be provide a service for, as well as one or more underlying event streams d, and one or more management objects m that might provide useful contextual information and services, such as membership or failure detection. Running within this abstract context, p implements what we may think of as a logic for "coordinating" the actions taken by the application with events in the underlying event streams. For example, given an underlying event stream d that lacks strong reliability properties, p might implement higher-level "coordination" logic such as peer-to-peer recovery, ordering, atomic delivery, or atomic commit. The functionality provided by p might not be related to any particular events stream; p might just as well implement tasks such as aggregating some value, managing locks or keys, or it might allow replicas of a to enter subsequent phases of processing in a synchronous manner.

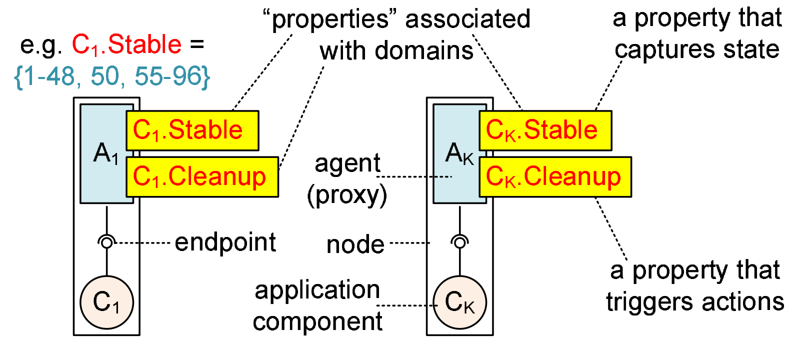

Figure 5.2: On the end hosts, properties are linked to application state and actions. For example, a property *Stable* defined on a node might be updated every time a message becomes stable, and a property *Cleanup* might cause the node to purge a message copy from its local cache cleanup whenever a new message is added to the set.

5.2 Properties

The logic implemented by protocol objects is expressed in terms of *properties*, variables that are replicated to each of the object's proxies. Some of these variables would represent the local state of the proxy, and would be updated in response to the incoming events. For example, a property *Received* might be a set of identifiers of messages that have been received by the proxy, and *Phase* might be the number of the current phase of processing the application object is in. Other properties might cause actions: whenever their values are updated, the proxy might send events to the application, or issue a request to the underlying management object. For example, a property *Clean* might be a set of identifiers of messages that can be cleaned up. Whenever a new number is added to the set, the proxy would issue a cleanup request (Figure 5.2). The goal is to represent the complete state required by the protocol to operate, and any actions it might need to trigger, through a set of properties replicated across its proxies. The protocol logic is then expressed as a distributed transformation that updates the values of properties at different locations (Figure 5.3). For example, a protocol that implements a "cleanup" logic would use a pair of properties *Stable* and *Cleanup*, the former representing messages that a node has received, and the latter representing the messages that the node

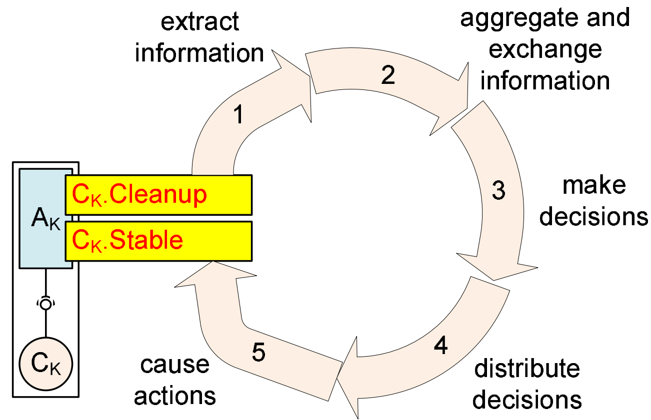

Figure 5.3: Protocols are modeled as distributed transformations on sets of properties.

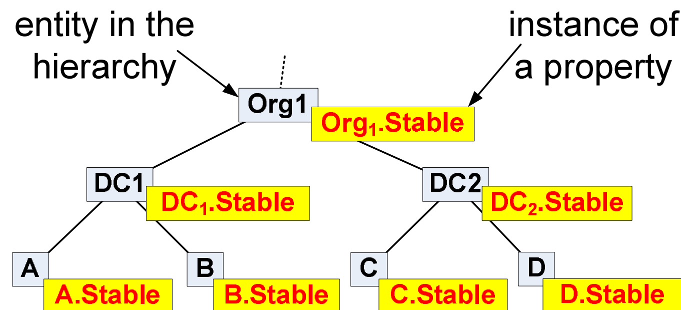

Figure 5.4: An instance of a *property* is associated with each domain that the system is subdivided into, from the global domain representing the entire system, down to the level of individual nodes. The values of different instances of the same property can be connected through aggregation and dissemination operators.

will purge from cache. The logic of such a protocol could be modeled as a transformation that inserts number m into the value of property *Cleanup* at each of the proxies as soon as, but not sooner than, the value of *Stable* at each of the proxies contains m.

5.3 Rules

Properties Framework implements a distributed protocol logic by leveraging the architecture described in Chapter 3. When a protocol is created, PF instantiates a hierarchy of recovery domains, each of which maintains a complete set of properties (Figure 5.4). Physically, these properties are maintained by agents implementing the respective do-

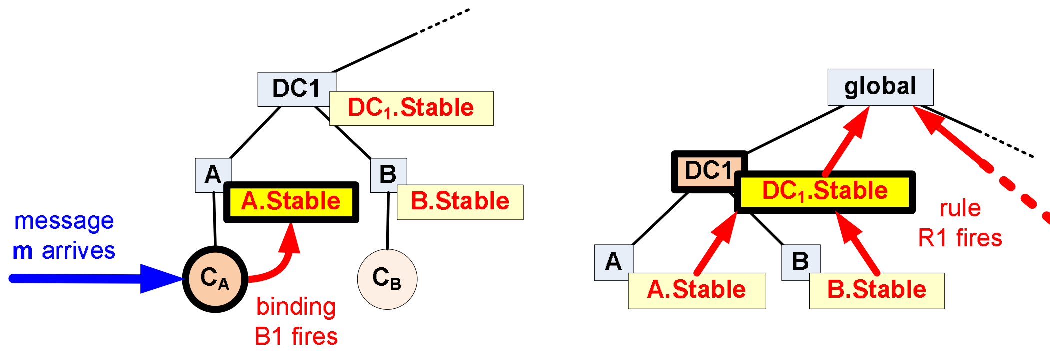

Figure 5.5: Collecting and aggregating information in the cleanup example. Whenever a message arrives, binding $B1$ updates the local value of $Stable$ (left). Rule $R1$ causes $Stable$ to be periodically aggregated among members of each domain (right). When the message stabilizes, information about it propagates bottom-up in a cascading manner.

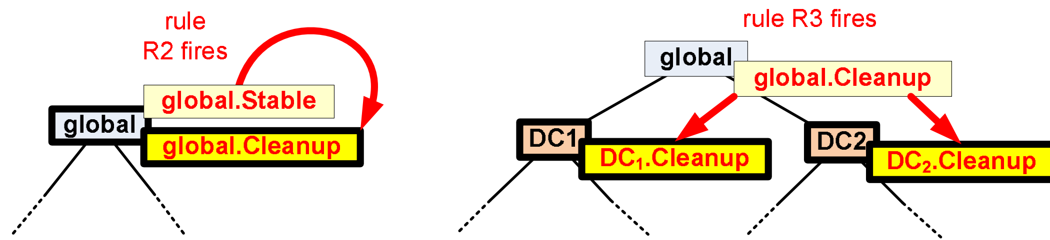

Figure 5.6: Making and disseminating decisions in the cleanup example. When information about a newly stabilized message surfaces at the root, rule $R2$ updates $Cleanup$ (left). Rule $R3$ periodically updates the values of this property at member domains from the values of their parents, causing global decisions to propagate top-down.

mains and deployed on the individual nodes. Agents form token rings, much as in the system described in Chapter 4, and can exchange, aggregate and update the values of their properties, or interact with agents "above" and "below" them in the hierarchy. The actions of the agents, and the exact way they update properties, is governed by a set of rules and conditions. Rules determine how property values are updated. Conditions control the process of joining new domains into the existing structure. For example, a program shown on Code 5.1 uses two bindings $B1, B2$ to relate property values to events, three rules $R1, R2, R3$ to update values, and a single condition $C1$ to control access to the protocol. The intuitive meaning of these is illustrated on Figure 5.5 and Figure 5.6.

```
01  protocol Cleanup
02  {
03    interface
04    {
05      callback Receive(int m);
06      DoCleanup(int m);
07    }
08
09    properties
10    {
11      intset Stable;
12      intset Cleanup;
13    }
14
15    bindings
16    {
17      on Receive(m) : Stable += m; (B1)
18      on update Cleanup(added A) : foreach (m in A) DoCleanup(m); (B2)
19    }
20
21    rules
22    {
23      Stable := [mono ,all, strict] children(∩).Stable; (R1)
24      global.Cleanup := Stable; (R2)
25      Cleanup ∪= parent.Cleanup; (R3)
26    }
27
28    conditions
29    {
30      parent.Stable ⊆ Stable; (C1)
31    }
32  }
```

Code 5.1: An example program in our Properties Language: "Cleanup". The program generates event *DoCleanup*(m) on all nodes as soon as, but not earlier than, an event *Received*(m) occurs on all nodes. The interaction between the protocol and the environment is captured within a pair of properties: $Stable$ and $Cleanup$, and bindings $B1$ and $B2$ that tie the values of those properties to events. The distributed computation on properties is implemented by rules $R1$, $R2$, and $R3$. Rule $R1$ aggregates information about the messages that are stable up the hierarchy. Rule $R2$ implements a "global" decision that a message stable everywhere can be cleaned up, and Rule $R3$ disseminates this decision down the hierarchy, to the individual nodes.

```
01  protocol CoordinatedPhases
02  {
03    interface
04    {
05      Phase(int k);
06    }
07
08    properties
09    {
10      int Last = 0;
11      int Next;
12    }
13
14    bindings
15    {
16      on update Next(assign k) : Phase(k); (B1)
17    }
18
19    rules
20    {
21      Last := [mono ,all, strict] children(min).Last; (R1)
22      global.Next := Last + 1; (R2)
23      Next [mono] := parent.Next; (R3)
24      local.Last := Next; (R4)
25    }
26
27    conditions
28    {
29      parent.Last ≤ Last; (C1)
30    }
31  }
```

Code 5.2: Another example program in our Properties Language: “CoordinatedPhases”.

5.4 Performance

We have implemented an early prototype of the Properties Framework and evaluated simple, manually translated Properties Language programs in a simulator. Our preliminary results demonstrate that the approach is feasible and useful, and provide additional validation of the architecture presented in Chapter 3.

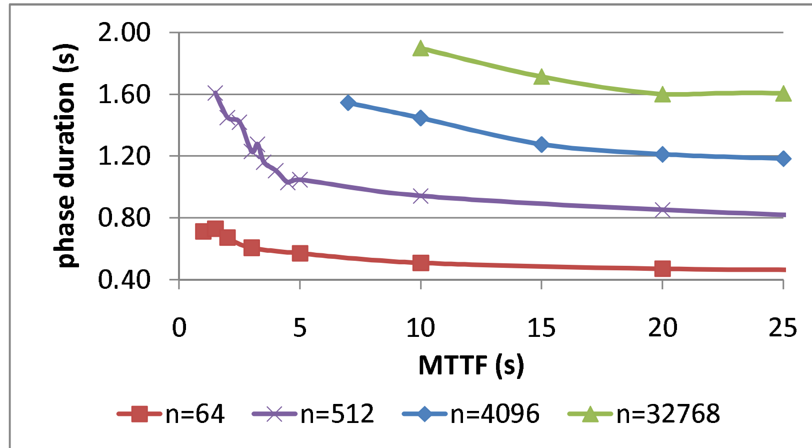

Figure 5.7: The effect of churn on the performance of the protocol from Code 5.2. The duration of a single phase is expressed as a function of the average time it takes for a node to crash, in systems of sizes ranging from 64 to 32768 nodes. Even with extreme rates of churn, where crash every 10 seconds, the protocol generated by the Properties Framework is able to make steady progress with very small performance penalty.

Code 5.2 shows a simple programs in the Properties Language that arranges for all protocol members to enter subsequence phases of computation in a coordinated manner. Figure 5.7 shows the performance of this simple program, in a simple simulated experiment, where a large number of nodes constantly crash, losing all state, and reboot. Even through nodes are crashing, and the rate of churn is extreme, the system as a whole makes steady progress. More preliminary results and additional details about the system can be found in a technical report [238].

Chapter 6
Conclusions

In this dissertation, we present a vision for a component integration technology that allows developers to work with instances of distributed protocols as entities similar to objects in an object-oriented programming language. In the preceding chapters, we have described a new programming model and a multicast architecture that support this vision. We have also described two prototype implementations that represent the first step towards making this vision a reality: the Live Objects (LO) component integration platform, and the QuickSilver Scalable Multicast (QSM) scalable reliable multicast engine. Our early work suggests that the vision may ultimately be feasible. Our prototypes already allow us to experiment with the model to understand its power and limitations, and to explore the performance and scalability implications of our design decisions. However, to fully realize our vision's potential, much work remains to be done.

In this chapter, we summarize our technical contributions and work in progress; we list unsolved problems and the limitations of our existing platform, and we point to various directions that might be explored by other researchers or by engineers.

The chapter is divided into two sections, corresponding to the two working prototypes we have implemented. In Section 6.1, we discuss topics most closely related to the Live Objects (LO) platform discussed in Chapter 2; we focus on component integration, the live objects abstraction, live object type system and language embeddings. In Section 6.2, we discuss topics most closely related to the QuickSilver Scalable Multicast (QSM) engine discussed in Chapter 4; we focus on topics revolving around multicast, including the architecture of Chapter 3 and the Properties Framework of Chapter 5. The reader should not attach too much importance to the structure; many of the topics discussed here cross the boundary and could be discussed in several places.

Each of the two sections begins with a summary of what has been built as a part of

the respective prototype, what has been demonstrated through it, and what is missing. This is followed by a list of subsections, each of which corresponds to a discrete piece of ongoing or future work related to the prototype. The discussion is intended to lay the road map for future extensions, but also as an opportunity to better illustrate the specific weaknesses and limitations of the existing prototypes.

The list of problem areas is not complete; we focus on major limitations we are aware of and technical issues related to some of the specific applications we have considered.

This chapter is fairly long, and not all problem areas we discuss here are of equal weight and importance. Some of these, such as those related to the live objects type system, or support for objects with multiple degrees of connectivity, are open research directions that we hope to tackle ourselves or inspire other researchers to pursue. Others represent engineering challenges and practical, if not somewhat tedious technical issues that need to be addressed to transform our vision into a real product.

6.1 Component Integration

We proposed a set of programming abstractions, such as a live object, a typed endpoint, a proxy, a distributed type, or a self-describing live object reference, that resemble abstractions used in object-oriented programming languages, but that reflect a distributed, peer-to-peer, replication-centric perspective. We have described the roles of the new constructs, and gave the intuitive meaning of familiar terms, such as to "store", "run", or "garbage-collect" a live object, or "dereference" a reference to it, within this new model. We have developed a prototype that supports these abstractions, programmatically as well as through a drag and drop interface. Through this prototype, we demonstrated numerous powerful features of the new model:

1. *The model cleanly supports replication*. Our prototype provides several template objects such as shared text or image, shared desktop, or a shared view into a region

of space, each of which maintains replicated state, and objects such as a reliable, totally ordered, checkpointed multicast channel that support a simple form of state replication. Our system is extensible, and creating new replicated objects is easy. We had students in a systems course use our platform to develop shared document objects that allow concurrent editing and locking of text regions. Overall, they felt that this task was quite easy, and the high quality of their submissions confirms it.

2. *The model takes advantage of strong typing without being language-specific*. Even though our existing prototype does not yet include a concrete language for specifying behavioral constraints, it does already implement its own typing infrastructure and reflection mechanism that are independent of any concrete programming language; the live objects runtime builds a separate metadata to represent types of objects and endpoints, and does not rely on the underlying .NET type system for type checking, except for the types of events. Thus, when a user drags and object onto a desktop, our platform dynamically determines whether the object has a user interface of the right type and can accept specific events, and dynamically makes decisions based on such information. Similarly, our visual live object design tool already performs static type checking during composition, and assists the user in correcting typing errors, by detailing the reasons for type mismatch. While we have not yet ported our runtime to platforms such as Java, we are currently not aware of any serious obstacles that would prevent cross-platform compatibility. Certain technical challenges do exist; we will discuss them in more detail below.

3. *The model supports legacy applications*. Unlike techniques that rely on introducing new language mechanisms, our model, by virtue of being language-agnostic, can be interfaced to a variety of existing systems. Users working within our model have been able to wrap materialized database views and groups of Microsoft Excel spreadsheet cells as live objects that they have subsequently used in composition

within our runtime, as well as embed live objects with a user interface within Microsoft Word documents and Microsoft PowerPoint presentations.

4. *The model and its abstractions correspond to familiar OO concepts*. We have built a visual designer tool, in which live objects are represented through block diagrams that resemble UML class diagrams. Composition in this tool is easy and intuitive.

5. *The model is protocol-agnostic, and can be applied from front-end to back-end*. We have implemented both higher-level components, such as shared documents, as well as lower-level components, such as naming services, repositories, or multicast protocols as live objects, and are currently in the process of implementing a host of other distributed protocols. The live object abstraction has also been used in parts of the live object runtime, and we are currently redesigning our communication infrastructure to transform the system into a set of live objects, dynamically assembled within the live objects runtime. Elements of the architecture of Chapter 3, including scope managers, recovery domains, or peer groups, can be wrapped as live objects, and our in-progress work on implementing this architecture adopts precisely this perspective. In future, we hope to explore using the paradigm at the level of network services such as routing, DNS, or network-level multicast.

6. *The model can be incrementally deployed*. We proposed a way to extend the .NET type system, by treating ordinary types as aliases that represent distributed entities. When building our type system and metadata and performing type checking, the live objects runtime recognizes such aliases as distributed types. This technique allowed us to implement a working and usable prototype without the need to change the compiler or develop a custom programming language. At the same time, the use of alias types makes it natural to work with the paradigm: live object and distributed type declarations resemble "ordinary" type declarations, only

decorated with additional annotations.

7. *The model supports type-safe composition*. The existing prototype supports several modes of composition: connecting multiple objects via their endpoints, loading objects stored within other objects, declaring parameterized templates and instantiating them by passing types or objects as parameters. Composite entities of this sort can be created within a visual designer tool. For each of the composition modes listed above, the platform already performs type checking, using endpoint matching or object subtyping relations we described earlier. Obviously, until we incorporate a language for expressing behavioral constraints, the full power of the model cannot be exploited, and its limits cannot be fully tested. Designing a behavioral constraint language is a future work. Nevertheless, even the existing, limited form of type checking is proving useful.

8. *The model is extensible*. New types of events, endpoints, and objects can already be defined by users today, and our implementation of the type system leaves open the possibility for users to define custom types of constraints, in temporal logic or other formalisms. The latter leads to potential issues; we will discuss those below.

9. *The model supports drag and drop development*. With our prototype, we have been able to demonstrate a process of developing a distributed application without the need to write any code, by first composing shared document and other live object references within a drag and drop visual designer tool, and then mashing up those components into what resembles a compound document, a web page, and a simple game, by just clicking on these references and dragging objects into one another.

10. *The model supports deployment via email*. With the existing prototype, live object references can be stored in files, sent over email, and embedded within Microsoft Word documents, and activated in place, much like ActiveX components (indeed,

we used the same underlying technology to achieve this result). Our implementation is still limited, in that libraries with .NET code that live objects may depend on still have to be deployed manually, but this is a limitation of our prototype, not the model. Several technical challenges this limitation raises are discussed below.

11. *The model supports composition without relying on inheritance*. While most systems require connecting components to agree on a common interface that they'll import from a shared library, we adopted the opposite approach: each object defines any of its dependencies as endpoints, and our runtime attempts to connect even certain pairs of binary-incompatible endpoints, by dynamically generating gluing code that intercepts calls directed to one interface, and efficiently routes them to another. Our current implementation of this feature is still very limited, in that the system is unable to convert events from one type to another. The work on this feature is ongoing. Still, the initial results are promising, and the prospect of being able to connect pairs of binary-incompatible components that do not depend on any shared libraries is exciting. In the world of live objects and mashups, where users may share their components on a massive scale, it may turn out to be a necessity, and we believe that the shared library paradigm is inherently non-scalable in this regard and doomed to fail in such environment. Nevertheless, in order to fully take advantage of the potential of this new style of composition, much further work is needed. We discuss some of the technical challenges below.

12. *The model does not depend on centralized services, such as naming*. Unlike many techniques that depend on the existence of external infrastructure, such as naming services, our model can express protocols that are completely peer-to-peer in flavor, and that can be bootstrapped from a clean state. This is possible because live objects do not have globally unique names or identifiers that might need to be resolved, and live object references are self-describing, and can be self-contained. It

is possible to index objects by storing their references in repositories, but we make no assumption of a single global naming scheme. While in most systems, an object is identified by an address in memory, or an identifier in a global namespace, in our model an object is uniquely identified simply by its description expressed in XML, which may, but does not have to, refer to components stored within global namespaces. In the current implementation, the use of this capability is very limited because currently the only way to express protocols is to refer to predefined .NET components that are further composed. Our ongoing work on the Properties Language, where protocols can be described with a small set of rules that are automatically translated into real implementations, should make it possible to encode even complex protocols within self-contained XML descriptions. This potential capability is intriguing; it has strengths, but also potential weaknesses. Some of the anticipated problems and technical challenges are discussed below.

While our experiences with the existing prototype have been very positive, this work is just an initial step. Many technical issues remain unsolved, and many capabilities have not been realized. In the remainder of this section, we discuss the problem areas.

6.1.1 Types

As mentioned earlier, although our model is ultimately designed to deal with behavioral types, the current implementation does not yet have a language for expressing behavioral constraints, and the typing and endpoint matching relations are currently limited to comparing the lists of endpoints and events, to determine whether entities "mechanically" fit together, but without comparing their (as yet unspecified) behaviors. Nevertheless, our implementation has been designed to support annotating endpoint and object types with custom user-defined behavioral constraints, and comparing them as a part of subtyping and endpoint matching implementations. Further work will involve the following steps.

Formalisms to Express Reliability and Security Properties

We are currently working on using examples of protocol specifications to develop simple constraint formalisms that can be used to specify reliability properties of multicast protocols. The most promising direction appears to be to leverage ideas used in temporal logic, I/O automata, and TLA specifications found in [14, 169, 39, 149, 171, 79]. We are also looking into using BAN logic [57] as a basic for a constraint formalism that could be used to describe security properties of protocols. All of this is a work in progress.

In the context of behavioral constraints, the biggest challenge appears to be addressing the inability to provide absolute liveness guarantees, stemming from the FLP impossibility result [111], a problem we have already signaled in Section 2.1.3. How to address this issue remains an open question. On one hand, a theoretician will argue that a protocol type that cannot be implemented in practice is useless, and a liveness property such as $\Diamond\mathcal{W}$ [72], which is provably impossible to implement in an asynchronous system [111], should never be used as a part of a type description. On the other hand, knowing that a multicast protocol would guarantee liveness if only it were given access to a failure detector of type $\Diamond\mathcal{W}$ through one of its endpoints does give us useful information about the protocol; if nothing else, it tells us that the protocol does provide certain mechanisms that attempt to achieve progress, a property that would distinguish it from a trivial "do nothing" protocol that trivially satisfies correctness guarantees by virtue of not doing anything at all. Hence, a practitioner would appreciate such a typing annotation; even though it could never be used to formally prove liveness guarantees in any real system, it could be useful in "filtering out" trivial implementations.

Ideally, we would like to be able to specify some form of weak liveness guarantees that cannot be satisfied by "do nothing" implementations, but are possible to implement in practice. At this point, we do not know if this is even possible; researchers have struggled to address this problem, and as of today, the issue remains unsolved. It might

turn out that to express such properties, one would need a complex formalism, and to determine subtyping in practice, one would need a theorem prover, which could be impractical if the live objects model were to be used pervasively, e.g., on low-powered mobile devices.

Ontology and Inheritance for Constraint Formalisms

While developing a few simple ad-hoc constraint formalisms following the approach we outlined in Chapter 2 would not be hard, and it may just be the best way to get started, we immediately run into a potential problem: we may easily end up with hundreds of formalisms that express the same kinds of behavioral properties (e.g., reliability), but that cannot be compared. As a result, components that otherwise logically match (e.g., a shared document that requires reliable multicast channel to work, and a reliable multicast channel that implements all of the required properties) may be incompatible simply because their constraints are expressed in different formalisms, and are thus incomparable.

Short of giving up on extensibility, and forcing users to use a single formalism that would subsume all others, we will need to develop a core set of abstractions and operators that all formalisms would use, and introduce a form of inheritance. At the minimum, all formalisms will need to be based on concepts such as an endpoint, event, a set of endpoints that belong to an object, connection, disconnection, endpoint matching and subtyping relations, but beyond that, it might be necessary to introduce concepts such as membership views or be able to talk about failure models, etc. In the end, one would like to be able to specify a formalism Ψ' that inherits abstractions of some other formalism Ψ, but may introduce additional ones, and where a limited form of comparison between formulas in Ψ and Ψ' might be possible. Thus, for example, while a formula in Ψ' might define subtle reliability properties of a protocol, it might still be possible to imply that

the protocol supports a simple form of reliability expressed by a formula in Ψ, and thus can be used with a range of application-level components annotated with formulas in Ψ.

A careful reader would notice that we have just proposed to reintroduce inheritance, which we have previously rejected. In the discussion of Section 1.4.2, we have argued that inheritance should not be confused with subtyping, and that it is not the right mechanism to implement composition. Here, however, we propose to introduce inheritance between constraint formalisms, not between live object types. Inheritance in this context would not serve as a basis for a subtyping relation, and would not be used to compose objects; it would simply allow one formalism to subsume and extend another.

Proof-Carrying Annotations for Provable Composition

Our live object types, as defined in Section 2.1.2, allow expressing and comparing behaviors, but not reasoning about composition. When a pair of objects is connected through their endpoints, and we try to reason about their behavior based on constraints mutually provided and required, we end up with a circular argument of the sort $P \Rightarrow Q \wedge Q \Rightarrow P$, which implies neither P, nor Q. In order to be able to prove either P or Q, one would need additional information on how the constraints provided depend on the constraints required, e.g., that some of the constraints are provided even if the required constraints are not satisfied, or that different constraints provided by the protocol depend on different subsets of the required constraints. Such additional annotation might be optionally appended to live object references, somewhat in the spirit of Proof-Carrying Code (PCC) [226]. A machine that may control sensitive resources would refuse to run objects that lack such annotations, whereas a typical user's desktop, where provable type safety is not critical, might accept composition based solely on comparing behaviors. In between these two extremes, one might imagine using proof-carrying annotations within a visual designer, but then limit the type check to just comparing

behaviors at runtime.

Designing a language for such proof-supporting annotations is a future work. In the model described here, we adopted a more pragmatic approach, based on the presumption that for a majority of developers, a superficial behavioral type check would suffice.

Incorporating Theorem Provers in a Scalable Manner

With formalisms such as temporal logic, comparing types would often require a theorem prover, even for very basic properties, such as those of a reliable multicast protocol or a replicated state machine. If the live objects model is used pervasively, such type checks would inevitably need to be performed by devices that lack the computational power to do so, where the overhead of such type checks would be too costly, and where latency is important. As pointed out in Section 2.1.2, in practice one might imagine there being a distributed infrastructure, in which facts computed by theorem provers, such as parts of the subtyping relation, are cached, and made available to client machines. Instead of running a theorem prover to determine whether subtyping holds, a client machine could fetch a pre-computed fact from such distributed "type cache", e.g., based on hashed type descriptions, and might also cache some of the commonly used results locally, to speed up its type checks. Designing such infrastructure is a future work. Our current prototype does cache the results of type checks, but cannot use external theorem provers. The problem is complex, for it involves security. One can easily imagine rogue theorem provers attempting to pollute the type cache with bogus facts, as well as valid theorem provers configured with buggy proving rules that might want to "revoke" facts they have pre-computed earlier. Hence, an architecture of this sort would potentially need to manage trust relationships and deal with revocation in a scalable manner.

6.1.2 Components

As mentioned earlier, one of the objectives of our work is for the model to be extensible. Our prototype does allow for creating custom components and types, but our implementation is limited in many respects. Further work will involve the following steps.

Automatically Finding and Downloading Libraries

Our current prototype is heavily dependent on primitive object templates and types defined in .NET libraries. Our prototype includes a system for managing libraries of custom components that users can define, and automatically loads such libraries as needed when references to components and types defined within them are encountered. However, the libraries themselves still have to be deployed manually, by placing the DLLs containing the .NET code on the client machines. Thus, even though the model allows deployment via email, simply by sending XML references to other users, the mechanism is limited, in that it assumes the recipients have locally installed type and component libraries that these XML references may depend on. To fully realize the potential of deployment via email, we need the ability to automatically find, download, and locally deploy a missing library, somewhat akin to the mechanism with which today's Web browsers can automatically download missing ActiveX controls. A naïve solution would be to embed URLs, from which library code can be downloaded, anywhere a reference to a library component occurs, but it is very easy to see that this would lead to inflating the size of object references and make the platform heavy-weight, tie the model to a client-server paradigm, and fail to address the versioning problems discussed in the following section. A comprehensive solution would require a system somewhat similar to DNS, essentially a variant of a scalable publish-subscribe, with a low rate of publishing (whenever a new version of a library is built), but massive numbers of receivers, extreme churn, and the need for efficient caching. Addressing this issue in a principled

manner, without creating a potential security hole, remains a major technical challenge.

Managing Multiple Versions of Objects and Types

The ability to reference components and types defined in .NET is convenient, but it introduces problems; as new versions of libraries are developed, different objects may refer to different versions of the same types and components, and the runtime has to be able to address the situation, in which two versions of the same library have to be simultaneously present in the system. Our current prototype has a versioning system, but does not solve the versioning issue in an elegant manner; once it loads a certain version of a .NET library into the process, it cannot upgrade it until the process is terminated, and if a reference is subsequently encountered to a component or type that requires a newer version, a runtime exception is thrown. A more elegant solution would permit unloading the old version of a library, fetching a new version, re-loading the library, all in a manner non-disruptive to already running components. Whether this is possible is not obvious; it appears to be difficult to achieve this in .NET. The problem remains a major challenge.

Conversions Between Binary-Incompatible Values

As mentioned earlier, our model rejects inheritance as a way of connecting components, instead relying on the live objects runtime to dynamically generate gluing code to relay events between binary-incompatible, but logically identical endpoints. This was a deliberate decision, motivated by the fact that permitting the use of shared libraries would tremendously aggravate the already present library versioning issues described above.

The current implementation of this mechanism in our prototype is very limited. Our system can generate gluing code in simple cases, but it is currently not capable of converting between pairs of binary-incompatible value types. Unfortunately, this is a major

problem that severely restricts usability. In practice, most components would need to define custom types of events to pass between endpoints. Thus, a component A, which represents a sensor, might define an event of type V that represents a sensor value, and a component B, which represents a user-interface widget, might define an event of type V' that also represents a sensor value, but is defined within a separate .NET library, and is not binary-compatible with V. We would like to be able to detect that V and V' have the same logical structure, or that there is a known way of converting between the two, and inject conversion code automatically when A is connected to B, so that any type of sensor could be used with any type of a user-interface widget, without having to break the model and rely on shared libraries (which, as we pointed out, would lead to problems). Currently, components that may need to communicate via events of custom user-defined types have to be defined within the same library.

The inability to convert between binary-incompatible value types is also the main obstacle to cross-platform interoperability. To achieve true language-independence, we'd need to be able to treat .NET and Java event types as isomorphic, perfectly equivalent.

A simple, but limited solution might require that custom value types are annotated in a manner similar to data types used within WSDL. However, one might imagine annotations that attempt to convey the purpose of the individual fields within data structures and include semantic descriptions, thus potentially permitting conversion between types that are not structurally identical. The benefit of doing so would be a greatly improved code reusability. Similar problems have been explored in the context of web services.

Synchronous vs. Asynchronous Object Interactions

In the model described in Section 2.1, we adopt a simplified view that events are passed via endpoints in a purely asynchronous, nonblocking fashion. In practice, using a purely asynchronous model is problematic, for two reasons. First, in many situations, such as

looking up a value in a dictionary, an asynchronous programming style is unnatural. Second, implementing a nonblocking interface generally incurs the overhead associated with memory allocation, since the event and its arguments have to be stored somewhere, and increases latency and the overall cost of event passing. For these reasons, in our prototype we have relaxed the model and implemented event passing as method calls. In some situations, the runtime simply forwards these calls to the other proxy connected via the endpoint; in others, where interfaces are binary-incompatible, it uses a dynamically-generated gluing logic that adjusts the lists of arguments and return values. We left open a possibility for the runtime to restrict certain kinds of endpoints to only allow asynchronous interfaces, and for the runtime to assist in making the calls asynchronous by automatically adding event queueing logic as a part of the connection between two endpoints. There exist many options, and neither is better than others; the choice should, in general, be made on a per-object and per-connection basis. For certain connections, e.g., between objects where latency is important or that might communicate in a fine-grained manner, direct method calls might be preferred, whereas in other situations, one might prefer to isolate the caller from the callee by making calls asynchronous. The decision might be correlated to whether proxies are placed in the same or different application domains. One might even imagine adjusting it dynamically, e.g., based on the frequency of invocations.

Objects and endpoints will need a system of annotations that could assist the runtime in making decisions such as whether to allow synchronous interactions or generate event queuing logic, e.g., by declaring whether an event handler is (or should be) guaranteed to terminate, may potentially block, or whether it is reentrant. With the ability to interact synchronously, the platform will also need a way to detect and remove deadlocks. In the current prototype, creating deadlocks is easy, and the developer must be careful when using blocking synchronization primitives. Addressing the problem may also re-

quire developing new language features for event passing that alleviate the potential to deadlock, or that combine the synchronous and asynchronous invocation models.

Looking at it from a different perspective, we'll need the ability to leverage asynchronous object interactions wherever they occur. One of the possible benefits is related to parallelism. Our model seems to naturally fit distributed and NUMA-style architectures, but it might also yield benefits in multi-core platforms, since the high degree of modularity and the strong preference towards asynchronous event-passing potentially translates to less context switching overhead, smaller lock contention, and more locality in data access. Exploiting these benefits is a major technical challenge with high payoff.

6.1.3 Security

The current prototype does not offer any form of security besides minimal type checking. Besides adding more expressive constraint formalisms that could express security and reliability properties, our vision for securing live objects involves the following steps.

Assigning Proxies to Isolated Application Domains

In our current prototype, all interconnected live object proxies run in the same application domain. This raises security concerns, and contributes to the library versioning problem mentioned earlier. An alternative approach would be to run each proxy within a separate application domain; this would increase overall security and address the versioning concerns (one could unload an old component library by terminating the application domain in which it was sandboxed), but it would be at a steep performance cost: method calls that cross application domains always involve costly marshaling. In practice, one would want to use a mixture of these two approaches. It might also be useful to isolate some proxies in separate processes; this could be used, for example, to enable mixing proxies coded in .NET with ones implemented in Java or unmanaged C/C++.

Besides implementing the capability to run interconnected live object proxies across different processes and application domains, there remains a question of what factors should affect the decision as to where a given proxy should run. Presumably, one would factor in the information about object type, e.g., proxies of an object annotated as tolerant of Byzantine failures or privacy-preserving might be sandboxed, whereas proxies of objects that deal with multimedia streams or have been annotated as low-latency might be run within the application domain of the live objects runtime or the underlying communication stack, to minimize the overheads. Designing such scheme is a topic for future study.

Scalable Components for Implementing Security

While the ability to express security contracts and run components within a secure runtime provides a good framework for security, implementing secure objects in practice will require a library of reusable building blocks, such as authentication or key management protocols, to choose from. Many protocols useful in this context have already been described in the past [256]; the main challenge will most likely revolve around the scalability of these solutions. We believe that many of these security protocols can leverage scalable multicast infrastructure, e.g., to disseminate keys, ACLs, or revocations, and that solving the scalability problem for multicast can go a long way towards enabling scalable security. We are also considering the use of the Properties Framework, as a mechanism potentially much more general than reliable multicast, as a tool for implementing security, and possibly expressing certain aspects of security through rules in this language, much like we can do so for simple properties relevant to reliability.

Integration with Existing Security Infrastructure

The ability to deploy our technology in real corporate networks will depend, to a large degree, on whether we can "tap" seamlessly into the established security infrastructure, such as Active Directory domains and X.509 certificates. For example, one might expect that access to files on our shared desktop is determined by Windows permissions, which might be defined in terms of domain users and roles, such as "Domain Administrators" or "Backup Service", or that elements of the desktop that lack valid X.509 certificates are indicated as insecure. Implementing a simple variant of such capability could be a matter of simply interfacing the existing APIs to access ACLs or impersonate security principals as needed when interfacing legacy applications, leveraging scalable multicast protocols to synchronize clients with a domain controller, etc. In a broader perspective, however, we'll need to consider a general question of how best to integrate the inherently decentralized architecture of our live objects with infrastructure that is mostly centralized and client-server in spirit, and the existing applications designed to fit this architecture, without reducing the overall solution back to a client-server model, and thus eliminating many of the potential benefits, such as better scalability and increased availability.

Leveraging Dynamic Composition and Reflection

One of the key features of the live objects platform is the ability to create and connect live object proxies programmatically at runtime. This is already being used extensively, e.g., in all our shared desktop examples, as well as in the underlying communication infrastructure, which dynamically injects simple event type checking and casting code to transform the underlying binary channel abstraction to one that carries events of a given type. This capability could also be useful in security, to dynamically assemble protocol stacks that match dynamically changing security attributes, and can be customized for

each user. For example, when sending a live object reference outside of a firewalled network, we could automatically embed tunneling logic. Encrypting and decrypting logic could be constructed dynamically based on the current privacy and encryption attributes of an object, and dynamically changed on a per-view basis with a suitable support from a membership object that could orchestrate such transitions. In existing systems, where distributed components are composed at design time and glued by code, this is difficult to achieve; we do not know of examples where this idea was pursued. A part of the challenge is to understand the usefulness of this mechanism in the context of security.

Managing Distributed Object-Object Connections

One potential vulnerability of our current prototype is that it does not attempt to control and synchronize in any way the connections between pairs of proxies on different machines; even though our model is fundamentally about working with sets of proxies and endpoints, the type checks are performed locally and independently on each node. This can potentially lead to unsafe deployments that type-check correctly, but fail despite the fact that each of the individual components is functioning correctly. We'll explain this by example.

Consider an application object A with some endpoint of type τ, and a communication channel C with a matching endpoint of type τ', i.e. such that $\tau \propto \tau'$. One of the guarantees that C specifies through τ' may be some weak, practically implementable form of a liveness guarantee that a message is eventually delivered to all instances of its endpoint. Given this guarantee, A might assume that all of its replicas contain the same data. But this is only true provided that proxies of A are connected to proxies of C. Suppose now that there exists a different channel C' of the same type τ', and that some of A's proxies are connected to proxies of C, whereas some others are connected to proxies

of C'. Even though type checking would still succeed, no guarantees can be made any more; proxies of A connected to C and C' will become partitioned, and free to diverge. Similar problems would occur when the same channel C is connected to proxies of two objects A and A' that started off in different initial states and issue conflicting updates.

The problem may not be critical in many application domains, but it is real; one can easily imagine communication channels being reused due to a human mistake, or shared document references being maliciously edited by an attacker, who might divert multicast traffic to his own network, or cause a split brain scenario just to corrupt the data.

One might imagine two approaches: static and dynamic. In the former approach, a live object's references might be signed with certificates that authorize connecting them to references of a specific other object, perhaps identified by a fingerprint. In the latter approach, one could use some infrastructure that dynamically verifies each connection. In between these extremes, one could imagine lease-based approaches, where certificates are issued for periods of time, and periodically re-requested. Other approaches might be possible. Understanding the practical implications of this problem and working out the details of a solution are a work in progress.

Just-in-Time Typing Support for Library Objects

Live object types are only contracts; for true security, one would want to verify that the actual implementations abide by those contracts. For composite objects, one could leverage proof-carrying annotations mentioned earlier. For primitive objects, implemented in libraries, the task would require verifying the code; a process that tends to be costly. One way to speed it up would be to leverage techniques such as Proof-Carrying Code. Another idea could be to use a language such as ML and a system such as NuPrl [163] to prove facts about programs in it. However, we believe that in practice, the most efficient

approach would be to develop a family of domain-specific languages, optimized for the purpose of implementing live object proxies, and with expressive power limited in such a way that reasoning about distributed types could be straightforward and fast enough to be performed dynamically, at the time the library is loaded, or when the object's proxy is to be first created for use in a secure context (hence the term "just-in-time typing").

We believe that the Properties Language we are designing has the potential to be such language; its rules seem to have a fairly clear and intuitive "distributed" semantics. We hope to eventually extend the Properties Framework with an automated typing system.

6.1.4 Resources

The dynamic nature of our live objects raises questions about resource consumption. There are two aspects to it: minimizing resource usage to reduce overhead, and managing the allocation of resources that are needed by objects for their proper operation. The following problem areas stand out as particularly important for real deployments.

Intelligent Activation and Deactivation of Proxies

Keeping a live object proxy in a runnable state can be expensive. For example, reliable multicast protocols may issue periodic acknowledgements, employ failure detectors that may involve heartbeats, or a gossip scheme that involves periodic interactions with proxies on other nodes; this overhead is often paid even if no data is actively exchanged by the protocol. Similarly, a proxy that has a user interface may be periodically requested to draw its contents; an operation that consumes resources even if drawing occurs outside of the visible area of the user's screen. When working with a typical shared compound document or desktop, the user may be interacting with only a subset of all objects placed on the desktop. In the current prototype, all desktop objects are simultaneously activated, even if they are not visible. This represents overhead, and with a sufficiently

large number of objects placed on a desktop, it may cause the system to be unusable on slower hardware. One possible remedy could be to create a proxy of an object when the user moves his or her "viewfinder" over a portion of the desktop containing the object, and deactivate it when the user leaves the area; this, however, could lead to extreme churn, and destabilize the underlying communication substrate. Another problem occurs if the user "zooms out" and ends up with a view over the desktop in which thousands of objects may simultaneously be visible; we discuss this issue below. Neither of these problems is trivial.

As another example, consider the repository object pattern, where a part of an object reference might be stored within another object, and the original reference may contain an embedded reference of the repository access protocol. In the runtime joins the repository access protocol every time it encounters a reference to it, and tears down its proxy immediately after retrieving the requested data, resources may also be wasted. Joining and unjoining tends to be costly, and causes churn, and indirect references that involve the same repository might occur frequently. It might be desirable to keep a repository access protocol active for a period of time after retrieving a reference, or even pre-fetch commonly requested references, to minimize the overhead.

In order for our system to grow and support realistic use cases, we'll need a scheme for intelligently deciding when to activate and deactivate proxies, how long to keep them alive after they have been disposed, etc. The current prototype lacks such functionality. In general, the solution will be application-specific, but it is likely that there exist common usage patterns that could benefit from dedicated infrastructure or language mechanisms.

Proxies with Different "Degrees" of Connectivity

In many applications, different users might want different levels of access to the same object, e.g., different levels of detail, different permissions, or QoS guarantees. To some degree, implementing this is possible even in the current prototype: a single object may define multiple endpoints corresponding to different functional roles, e.g., one endpoint for senders and another for receivers, or one for clients and another for service replicas. One could go further and create separate endpoints for heavy-duty vs. casual clients, or otherwise group clients into a finite number of categories, each with its designated endpoint.

Some applications, however, pose an even greater challenge. Consider a large-scale version of a three-dimensional spatial desktop object shown on Figure 2.6. If the desktop were to visualize a real battlefield, there might be thousands of moving objects on it at different distances from the user's camera; viewing them all at once might be infeasible, excessively costly, or too disruptive. Intelligent proxy activation and deactivation, discussed earlier, might help to control resource usage by displaying only objects close to the user, but in this particular application, displaying all objects would be preferred. To keep resource usage manageable, one might want to be able to view distant objects at a lower level of detail, for example receive their coordinates less frequently. Similarly, an object that represents a virtual event where tens of thousands of avatars might gather to watch it, all at once, one might not want to keep track of precise movements of tens of thousands of avatars at the same time, but perhaps just a few in proximity to one's own. Avatars standing further away might be rendered with smaller precision, and their movements visualized less frequently and after a longer delay. The ability to differentiate between objects closer and further away and adjust the rate of multicast traffic based on this information would determine whether the given application is feasible or not.

In the sorts of applications mentioned above, there could be a great many (perhaps

infinitely many) different “degrees” of access to a single object, determined by factors such as a distance in the virtual space. To support such applications, we’d need replication protocols and data structures that can support different “degrees” of membership. We might also need to revisit the model, and try to understand whether having proxies with a large number of different “degrees” of connectivity within the same object should be somehow reflected in our type definitions or language abstractions; for example, an endpoint type might constrain latency as a function of the “degree” of connectivity.

Managing Resources Required for Correct Operation

Many objects explicitly or implicitly depend on certain resources, which are themselves not live objects, and are not explicitly referenced and covered by the model. For example, multicast channels might depend on the availability of IP multicast, and the support for IGMP snooping in the networking hardware or the presence of dedicated services on the network. Objects that persist state may depend on the presence of certain files or simply on the availability of a certain amount of space on a local filesystem. Multicast objects might implicitly depend on the availability of tunnels or repeaters that would duplicate traffic between different networks. Objects that have a user interface and may refer to textures that might be dynamically downloaded might depend on the presence of textures somewhere on the network, occupying space on the web server.

The challenge is to express such relationships, perhaps within the XML references, but also to design a scheme for allocating resources, perhaps on demand, garbage collecting them, and handling situations where resources are no longer available because they have been reclaimed. A part of the challenge is to handle resources such as IP multicast addresses, which may be local; migrating objects with such dependencies might make no sense, or it might require dynamically allocating new resources or objects, such as VPN tunnels, when the migrated reference is activated in a new location.

6.1.5 Language

Our model focused on type-safe composition, as the most important aspect of our object-oriented embedding; indeed, one can view the XML format of our live object references as a primitive form of a live object "language". We are excited about this perspective, but we also recognize the limitations of the "language" supported by the existing prototype. In the immediate future, we plan to focus on exploring the following problem areas.

Object Specifications Embedded in Object References

There are two aspects to our composition language: the ability to describe primitive building blocks, and the ability to composed those building blocks into larger entities. In context of the former mechanism, our language currently only allows referring to objects defined in external .NET libraries. In some sense, this is analogous to embedding references to ActiveX controls in HTML pages; one can embed custom controls and then arrange them within a page, but the controls have to be defined externally.

We are planning to extend this capability by making it possible to embed descriptions of live object proxy logic directly within object references, much in the way one can embed JavaScript in web pages, thus making object references completely self-contained. To this purpose, we are looking for languages that could capture live object's proxy logic in a concise form while retaining some degree of generality. Our novel Properties Language, discussed in Chapter 5, has a chance to be such language, but in the present form, it is perhaps overly specialized towards state machine replication and hierarchical protocols.

If we compare our live objects designer with a Windows Forms designer in Visual Studio, we'll notice that both currently support composition by dragging and dropping reusable building blocks and connecting them in various ways, but Visual Studio also allows the programmer to open one of those building blocks, and customize its behavior

by writing code. We believe that live objects need a similar capability, hence the need for a general-purpose declarative language that can express live object proxy logic in a form concise enough to be embedded directly in the XML references.

High-Level Constructs in our Composition Language

The language used to describe references in our existing prototype is very simple, but it does include equivalents of a number of higher-level constructs. For example, parameterized objects are, in some sense, equivalent to method calls: the parameterized template can be thought of as a method, and references of embedded objects given to it as parameters can be thought of as method arguments. This analogy intuitively makes perfect sense: most pre-installed components bundled with our prototype are templates, and references passed as parameters to these templates are used to launch proxies that are dynamically connected either to each other, or to internal proxies, created programmatically. Thus, most templates have embedded composition logic, encapsulated and hidden from the developer.

Similarly, a composite reference, in which several internal objects are connected is, in some sense, an analogue of a list of local variables or member fields followed by explicit *connect* statements. Finally, references to repository objects are an analogue to operations on collections.

It seems natural to try extending our language with additional higher-level mechanisms, for example conditional branching and loops. One of the future challenges will be to understand which language mechanisms make sense in this context, and which are feasible.

As far as feasibility is concerned, there are a number of constraints that need to be respected. First, since these mechanisms would form a non-customizable part of the language (much like built-in keywords in Java), we would want the mechanisms to be

generic. Second, a reference expressed using these mechanisms should still allow types of its constituent parts to be determined statically, and permit efficient type checking at design time. Third, we would like to retain the high-level declarative style of our references, and the ability to represent them visually through an intuitive block diagram within a drag and drop design tool. Finally, we need to define a clear boundary between the parts of the object logic that belong to the composition language, and the parts of it that belong to object specifications, and that could be expressed in multiple dialects. The latter is, in some sense, similar to the distinction between HTML, which defines the structure of a web page, and the scripting logic that can be embedded within HTML, and that could be expressed in multiple languages.

In this context, one possibility could be to allow only static configuration to be expressed within the composition language, and leave event handling logic to embedded specifications. One could relax this, for example, by supporting the handling of built-in events, such as endpoint *connect* and *disconnect*, within the composition language, to make it possible to express certain types of reconfiguration, such as reconnecting on failure.

Controlling Inflation of the Size of Object References

Most live object references used in our current prototype and in student projects we supervised are relatively short: a typical reference is 2–3 KB in size, and the most complex one is 6 KB. Compressing these references can shrink them to a few hundred bytes. One can easily imagine, though, that in realistic scenarios, references could grow to tens of kilobytes, and if we permit object logic to be expressed within these references, as we have just postulated above, the sizes of some of them might grow even larger, perhaps by an order of magnitude. If these references are used as parts of compound documents, the documents themselves could easily reach tens of megabytes. Deploying such doc-

uments via email would be problematic. Also, since activating an object might require downloading substantial amounts of information, the platform would be perceived by the users as unresponsive.

Indirection, implemented through external references, can trim the size of object references, but pushes the overhead to run time, forcing clients to download a considerable amount of data at the time an object is first accessed. Techniques such as caching and prefetching could mitigate the problem, but raise the question about the "freshness" of references that might have been downloaded a long time ago, and reused. Our composition language will need to be extended to support constraints on the freshness of cached references, an equivalent of a *volatile* keyword in C++, etc., and the runtime will need to be extended with mechanisms that minimize storage and processing overhead resulting from reference size and the extra levels of indirection while respecting these constraints.

6.2 Multicast Infrastructure

Although the live objects model is meant to be general, we expect that most live objects used in real systems would be internally powered by some form of multicast. Whether the model can be adopted on a massive scale thus depends to a large degree on whether we can build a multicast infrastructure that can support the common patterns of usage.

Unfortunately, at this point it is not at all obvious what the common patterns of usage would be. It seems reasonable to assume that the system would need to scale in several dimensions, such as the total number of nodes, the average number of nodes in a single multicast group, the total number of multicast group, the average number of groups that a single node is a member of, etc. It would also seem reasonable to assume a heavy-tailed popularity distribution, some form of regularity and clustering based on interest; we have discussed these aspects in Chapter 4. Beyond that, however, we can only speculate, and there is no other way to tell than trying to deploy an initial prototype

in a real setting, to see how it is being used, and where it breaks, and use this feedback to refine the design.

Accordingly, we have built a prototype multicast platform with a simple reliability property that is sufficient for simple types of objects, such as channels carrying sensor values, airplane coordinates, or buildings colors, and that could be used in combination with higher-level logic for locking and state transfer to support replicated documents. We have described this prototype in Chapter 4. The system currently only works in LAN settings where IP multicast is available, and does not offer stronger forms of reliability on its own. Also, the novel technique we proposed to scale with the number of groups, while promising, is currently implemented only in a simplified form: it can yield benefits only in systems with very regular overlap, and degrades in irregular overlap scenarios; we discuss the limitations of the system and our ongoing work towards addressing them further in this section. Nevertheless, despite all the limitations of the current platform, our prototype has already been useful, in that it allowed us to demonstrate the following.

1. *One can build a high-performance system in .NET*. Our prototype is implemented entirely in .NET, mostly in C#, except about 6000 lines of managed C++ code required to access Windows I/O completion port API, and it can be used by .NET applications as a library to send and receive managed objects via a managed API. A simple managed application, running on a 1.3GHz cluster node on a 100Mbps LAN can send over 8000, and receive over 9500 kilobyte-sized messages per second, and the performance degrades by only a few percent as the system scales to 100s of nodes. On the same hardware, the system can send or receive 5000 messages per second with only 30% processor usage. With 2.6GHz nodes, the system comes close to saturating the network, and it has an even smaller CPU footprint. These results contrast with most popular packages for Java/.NET; for example, JGroups saturates CPU on the same hardware at a throughput an order of mag-

nitude smaller. This result is important; our model depends on mechanisms such as strong types or reflection, which are characteristic of modern runtime environments. Our work demonstrates that working in a managed environment does not have to mean sacrificing high performance or incurring high CPU overheads.

2. *Network-level reliable multicast can scale to 100s of nodes*. Our prototype guarantees delivery if the sender does not fail; if it does, the guarantees are only probabilistic, on a per-region basis. We have experimentally confirmed that the system scales from 2 to 200 nodes with only a few percent performance penalty at the highest data rates, which is a competitive result, and our theoretical analysis suggests that it should still be able to support high data rates at configurations a few times larger. This is achieved without the use of techniques such as filtering in software, or proxies that consume resources and introduce latency and bottleneck. In the existing systems, the scalability of network-level reliable multicast has been limited by the ability to sustain stable throughput with large numbers of recipients. We have demonstrated that it is possible to sustain stable throughput in such configurations.

3. *One can amortize overhead across multicast groups*. We devised a novel scheme that allows amortizing overhead between instances of multicast protocols, by exploiting regularities in the patterns of group overlap. The variant of the algorithm we have implemented is very simple, and calculates regions based on the full set of groups; consequently, the current prototype can perform well only in scenarios with very regular overlap, and degrades quickly otherwise. With perfect overlap, our system scales to as many as 8192 groups with only a few percent performance penalty. Of course, real systems would rarely be this regular, but it has been reported that even in systems with irregular overlap, one could arrange for regularities to occur by partitioning the set of all groups into more regular hierarchies

[300]. This partitioning scheme has not yet been integrated into our system, and its usefulness has yet to be confirmed in experiments, but the simulation results are promising.

4. *One can leverage IP multicast without triggering state explosion.* Our prototype leverages IP multicast for transmission, but it utilizes relatively few multicast addresses, at most one per node. This way, our prototype avoids the state explosion problem that plagues large data centers. In the current system, this ability comes at a cost of higher overall network bandwidth utilization: to send a single message, multiple UDP transmissions may be needed if a group spans over many regions. The partitioning scheme outlined earlier has the potential to balance the two factors more efficiently, slightly increasing the number of regions, but reducing the average number of regions per group; evaluating this is a work in progress.

5. *Reactive approaches do not have to carry the risk of broadcast storms.* Many existing multicast platforms, when perturbed, tend to melt down in a bad way, leading to broadcast storms that can shut down a data center; we have discussed these phenomena in Section 4.3.7. Because of this, many researchers have turned to proactive techniques, such as FEC. Our prototype uses a purely reactive approach, yet it handles many types of perturbations gracefully. In nearly a thousand experiments we have run, each involving millions of messages, our system always predictably responded to perturbations such as losses, freezes, crashes, and churn, by reducing the rate of multicast, or entirely suppressing it, and always eventually stabilized.

6. *Efficient recovery does not have to involve high overhead.* Our system, by default, circulates its tokens at a rate of 1 token per second, and the ACK/NAK feedback provided to the sender is also generated at this rate. This represents an extremely low rate of control overhead, and is among the reasons why the CPU utilization in

our system is exceptionally low. Despite this, our system is very reactive; it can quickly recover from very long bursts of packet loss and other disruptions through a massively parallel recovery process that often involves dozens of other nodes.

7. *One can improve performance by controlling local scheduling and memory usage*. We have described an intriguing connection between performance and scalability and aspects of the protocol stack architecture such as the order of event processing or memory consumption. Controlling these factors allowed us to achieve competitive performance. We have summarized our experiences in the form of a number of performance insights and general design guidelines; we listed those in Chapter 4.

Overall, our prototype has been a successful feasibility study. The ability to achieve high performance in a managed environment, scale sub-linearly with the number of groups, marry scalability and reliability, exploit hardware support in data centers, or understand and deal with broadcast storms or other instabilities: any of these could have turned out to be a show-stopper, yet so far, we have not come across any unsurmountable obstacles.

Besides being a successful feasibility experiment, our prototype also allowed us to get a better understanding of the phenomena related to performance and scalability. As a result, we ended up with a highly efficient architecture, which served as a great starting point for future refinement. Indeed, the architecture described in Chapter 3, as well as the Properties Framework based on it, introduced in Chapter 5, are direct extensions and generalizations of our existing prototype; the relationships can be understood as follows.

1. In our existing prototype, nodes running a protocol could play many roles; a node, besides representing itself, could collect state and act on behalf of its partition or region. The roles a node plays depend on the membership views handed out by the GMS. In the architecture of Chapter 3, we generalized this idea, and proposed to model hierarchical protocols as networks of interconnected protocol agents that

are installed at the individual nodes by the external management entity (GMS). We propose to think of the GMS as an entity that does not just passively hand out membership views, but that actively orchestrates the execution of the protocol by delegating and actively controlling agents that, in a sense, work "on behalf of" it.

2. In our existing prototype, protocol "agents" form a hierarchy; a recovery protocol running in each group is effectively composed of "microprotocols" running between protocol agents of different types, simultaneously at multiple levels. In the architecture of Chapter 3, we evolve this idea into a general-purpose hierarchical protocol "decomposition" scheme, and propose to view each protocol as a structural composition of multiple microprotocols that have been glued together into a single structure through a hierarchy of membership views provided by the GMS.

3. In our existing prototype, the hierarchy of agents has only three levels. Microprotocols used at each level are fixed, implemented as token rings at the two bottom levels, and a star topology, with the sender in the middle, at the top level. In the architecture of Chapter 3, we propose to relax these assumptions: there could be arbitrarily many levels in the hierarchy, each microprotocol could use a different distributed structure, e.g., a token ring vs. a binary tree vs. a scheme based on gossip, and the sender does not need to play a distinguished role as the central point at which all agents are interconnected; the "root" agent might be executed anywhere, and a single protocol instance could accommodate multiple senders.

4. In our existing prototype, the guarantees provided by the protocol and its structure are fixed, and so is the logic of each microprotocol; however, we have used a hand-coded structure similar to that described in Chapter 5. The protocol state in our prototype is represented in a form resembling a set of properties associated with agents, such as the set of identifiers of packets received, stable, missing, or ready for cleanup. Our token rings are used as means of aggregating and disseminating

the values of these properties. In our ongoing work on the Properties Framework, we are trying to pursue this idea in a more systematic manner, and generalize this scheme to allow other protocols, with different reliability guarantees, to be represented as distributed data flows between sets of properties maintained by agents.

5. In our existing prototype, the method of dissemination is fixed, and relies on support for IP multicast, but it also has the compositional flavor: a logical channel to a group splits into per-region channels, in a way controlled by the groups-to-regions mapping generated by the GMS. In the architecture of Chapter 3, we have generalized this idea and proposed to use our hierarchical protocol composition approach to dissemination as well. As an example of this approach, we have described how one can use it to construct hierarchical overlays.

6. In our existing prototype, microprotocols running within each region perform recovery simultaneously for all groups overlapping on the region. We have made sure to retain this important feature in the architecture of Chapter 3; a recovery domain in this architecture is not associated with a particular protocol instance, and may be assigned to perform recovery for multiple protocols simultaneously when multiple sessions are externally "installed" in it; this has been described in Section 3.2.10, and illustrated on Figure 3.25 and Figure 3.31. This allows, for example, a single token ring to carry recovery records for multiple protocols.

7. In our existing prototype, several protocol instances can be effectively "collapsed" into one within each region thanks to the fact that messages issued by senders, in addition to being indexed on a per-group basis, are also indexed on a per-region basis, across groups. The ability to do so depends, to a degree, on the fact that dissemination and recovery in our current prototype use the same structure: regions associated with IP multicast addresses are the same regions that are used as a ba-

sis for forming token rings that run recovery. It also partially depends on the fact that senders "know" the target regions. In the generalized architecture of Chapter 3, where dissemination is decoupled from recovery, and within hierarchical overlays of the sort we have described, senders in general will not have information about the target regions. Directly carrying over the protocol "compression" scheme used in our prototype into this generalized architecture is currently not possible. Adding this capability is a future work. We discuss this issue in more detail below.

8. In our existing prototype, the membership service is centralized. We have noticed, however, that to perform their roles as partition members, partition leaders, or region leaders, nodes do not, in fact, need to have full membership information. In the architecture of Chapter 3, we have made use of this observation. We have outlined a protocol, through which membership information can be maintained by a hierarchy of scope managers in a completely decentralized, scalable manner, so that each scope manager controls only a certain part of the hierarchy, at a certain level of granularity, and dispatches a subset of protocol agents. This might potentially enable scalability to hundreds of thousands of nodes. It also yields an additional benefit: by decentralizing the GMS in the way we proposed, we can now achieve full encapsulation, in accordance with the object-oriented principles.

Currently, neither the hierarchical structures of Chapter 3, nor the Properties Framework are implemented; both systems are a work in progress. The challenge is not only to build these systems, but also to formally prove that they can indeed deliver strong guarantees. The approach we propose is controversial in that, unlike the majority of systems, it does not use mechanisms such as systemwide consistent membership views, which makes it difficult to define and prove its properties; consistency relies on a distributed hierarchy of interconnected membership views that evolve in a manner that is synchronized to a

degree, but the entire view hierarchy never materializes in any place of the system.

Until we demonstrate a working prototype, and a formal model for reasoning about consistency and reliability within this architecture, the question as to whether the architecture is feasible and practically useful remains open. Our experiences with the model and early experiments suggest that chances for a positive answer are high. We have built [238] an early prototype of the Properties Framework, in which protocols are run by a set of agents loosely synchronized by a hierarchy of views, just as in the architecture of Chapter 3, and run simple programs in our Properties Language. The prototype is still far from being usable in real applications, for the programs are partially hand-compiled, the membership service that deploys agents and supplies them with membership views is centralized, and the entire system runs in a simulated environment, where communication and failure detection have been abstracted. Nevertheless, the way membership views are calculated and the way protocol agents are synchronized in this prototype conforms to what we have described in Chapter 3, and the reliability and consistency characteristics of this prototype are most likely either identical to, or very closely approximate what a real instance of a Properties Framework would look like.

Our experiences with this early prototype, some of which have been discussed in Section 5, and the results on Figure 5.7 and in [238], strongly suggest that the architecture we have described has sufficient power to allow protocols to make irreversible decisions and achieve progress, despite packet losses, reordering and asynchrony, high rates of churn, and node crashes. In particular, it would seem that it is possible for protocols running within the architecture to calculate monotonic properties such as which messages are stable, reinstate crashed agents and allow new nodes to join the protocols in a way that does not violate monotonicity, and then use such monotonic properties to make irreversible decisions on cleanup and delivery. In combination with simple loss recovery logic, this appears to be sufficient to implement basic reliability properties, such

as last copy recall, and even atomicity. If the model can also support ordering, as we believe it does, it should support replicated objects of the sort we used in our prototype of the live objects platform.

Even after completing the Properties Framework prototype, many practical problems will remain unsolved; we discuss the problem areas in the remainder of this section.

6.2.1 State

In the current version of the live objects platform, replicated state is generally stored in checkpointed communication channels: reliable channels that provide total ordering and a simple form of state transfer, whereby a newly activated proxy receives a checkpoint created on-demand from one of the existing proxies, in a manner synchronized with the stream of updates. This functionality, offered by the non-scalable, TCP-based multicast substrate that the platform is configured with by default, is sufficient for most uses. Our scalable multicast prototype of Chapter 4, however, does not provide total ordering, and in the absence of total ordering, it cannot provide state transfer, either. One can implement the desired semantics by combining scalable channels with the totally ordered channels provided by the TCP-based substrate: scalable channels could be used for dissemination, and totally ordered channels for synchronization and state transfer; an example of object composition that our platform was designed for. Next steps shall involve the following.

Limiting Dependence on Costly Flavors of Multicast

We just noted that out scalable multicast substrate lacks total ordering and state transfer. This functionality is generally hard to implement in a scalable manner, for it cannot be decentralized; there always has to be a central "orderer" that assigns message numbers. However, there exist many protocols that minimize overhead in specific scenarios, for

example by migrating the “orderer” role to the actively multicasting node, which works great if messages are transmitted in longer sequences, but might be counterproductive in applications, in which messages are isolated. We are not aware of any “one size fits all” ordering scheme, and we believe that no such scheme exists. We envision that the best way to approach the problem in practice is to provide the user with a large number of reusable building blocks that implement various standalone ordering, locking, state transfer and reconciliation, versioning, and a variety of other synchronization primitives, independent of any particular multicast substrate, and optimized for different patterns of usage, and allow the user to freely combine those primitives with different types of non-synchronized, unordered multicast channels, at the application level. We believe that in an overwhelming majority of applications, total ordering, in fact, is not strictly necessary, and may be substituted by other forms of synchronization, which may be coarse-grained, or used only for a subset of all communication. Many types of updates are idempotent or commute. Temporary inconsistency often is not fatal, and can be detected and resolved through other means; recall our discussion of collaborative editing in Section 1.3.

In this context, the key challenge will be to design synchronization primitives, data structures, language abstractions, and live object design patterns that can facilitate building application-level synchronization logic to complement FIFO-ordered and unordered channels. We are hoping that with the right set of such tools, users would be able to create application objects that depend on costly ordering and synchronization mechanisms only to a very limited degree, and that perhaps for 99% of communication channels used in practice, it should suffice to use weaker, but also more scalable forms of reliability.

Backup and State Persistence for Replicated Objects

Currently, our prototype does not support persistence. Replicated state in checkpointed communication channels exists for as long as there exist some proxies of an application object using the channel, and vanishes otherwise. In applications such as collaborative editing or gaming, this is undesirable, for it would mean that a document, or an entire virtual island, with all its data and configuration, vanishes when the last user leaves.

One way to avoid this would be to ensure that there always exist some $m \geq 1$ dedicated "backup" proxies connected to the channel, solely for the purpose of mirroring the replicated state; this can be easily done today, and it works in any scenario. A more sophisticated technique would be to track the set of proxies of a replicated object either by a membership service or through some other means, and ensure that if the number n of proxies of an application object replicating the state is smaller than some threshold m, then there are always $m-n$ backup proxies active; the backup proxies in this scenario would be dynamically spawned and retired as clients arrive and depart. This would also provide a controlled level of fault-tolerance, but it would shift the burden of replication entirely to clients as long as sufficiently many of them are present. One can easily come up with many other variants of this scheme, perhaps adjusting the number m according to the perceived churn. Exploring the space of solutions is an important future work.

It is well worth noting that the problem of persistence provides an excellent opportunity for a successful marriage of our live distributed objects model with the established architecture based on data centers. We envision that as much as the majority of multicast communication should occur directly between clients, the data centers - with their well provisioned and highly available resources - would be the perfect platform for the entire persistence infrastructure, including membership services that monitor access to replicated objects, as well as standby nodes that can be enlisted to host backup replicas.

Using Locally Cached Configuration State in Proxies

In our current prototype, object state is generally replicated and stored in communication channels, but there exist exceptions to this rule. For example, in our three-dimensional desktop example shown on Figure 2.6, the position of the camera, or its association with a stream of coordinates, are local for each user. Each proxy of the desktop maintains its own private configuration settings, and those are not replicated. In this example, the local settings are discarded and reset to defaults when the user closes and reopens the object, but we could imagine storing this data locally, much in the same way today's web pages can store cookies on user's machines, and reloading it next time the object is accessed. There are other types of such local state; for example, in case of a shared folder that may contain large files, it would make sense to allow each user to locally decide whether a replica of a given file will be stored locally; the user might be interested in only a subset of the files, might not have enough resources, or might not be able to download the entire file in a single session. Local object state, such as which blocks have been downloaded, and which are pending download, would need to be stored between subsequent accesses. Currently, we lack mechanisms that would enable this. Challenges include deciding how to allocate storage resources for local state, when and how to garbage collect it, and how to share such state between different proxies of an object running on the same machine.

A part of the challenge is also to understand how the decision about making a certain part of the object's state to be local vs. making it replicated affects the environment from the scalability, security, resource consumption and usability perspective, and whether the runtime should enforce some rules constraining this choice.

Synchronizing Replicated State Stored in Documents

The ability to embed live objects in office documents, such as presentations and spreadsheets, while tremendously useful, creates a problem; unlike in most user-interface objects in our prototype, here the replicated state is not discarded when the proxy terminates, it is also persisted within the document itself. The document, with a copy of the replicated state persisted in it, could then be freely passed around. This situation resembles, to a degree, problems that distributed commit protocols have been designed for, but here the situation is even more complex, for a single state replica can itself be further arbitrarily replicated as the document is passed around, and documents with this replica could then be activated on multiple clients. There are multiple technical challenges here.

First, each document, besides storing the channel's reference, will also need to store some private local state of the underlying replication protocol, associated with this document copy. The main issue here is how this state should be persisted and synchronized with data in the channel; documents usually do not support log append style of updates, so it may be difficult to use traditional commit protocols. Changes might need to be applied in batches, and the document may need to be saved as a whole, in a non-destructive manner; the latter may require creating temporary versions and replacing the document atomically. This would need to be smoothly integrated into existing office applications.

Secondly, the possibility of replicating individual document replicas, with their embedded state, requires a replication scheme that binds the embedded state to the individual machines, users, and locations, so that once the document is copied over to another machine, the embedded state in the new copy is ignored. This seems to be necessary for any quorum-based scheme to work, but poses a problem from the usability perspective. The challenge is to find a scheme that permits consistent updates without unnecessarily constraining the user or creating too much of a burden in common office scenarios.

Modeling Nested Transactions as Object Composition

Our prototypes currently do not offer any support for database-style nested transactions. While a single transaction, or a sequence of transactions among a set of participants, could be modeled as atomic commit and wrapped as a live object built on top of a reliable multicast channel, and we believe that many database replication schemes could be implemented in this manner, it is not obvious how to express nested transactions in our architecture. One possibility is to model nesting of transaction as a composition. If this turns out to be expressive enough, we hope to incorporate transactions as one of the abstractions that can be "imported" and used within our object composition language.

6.2.2 Scalability

In terms of scalability of the live objects paradigm, the most important problem appears to be the ability to support large numbers of objects that might run simultaneously across overlapping sets of nodes. As we have argued before, most objects rely on some flavor of reliable multicast, therefore solving this problem for multicast is the necessary first step. Other dimensions of scalability, such as support for large numbers of nodes, have been explored extensively in the prior literature. We believe that the architecture of Chapter 3 and the Properties Framework will scale sufficiently well to support multicast and other types of live objects shared by very large numbers of users.

Our approach towards achieving scalability with the number of overlapping objects, based on leveraging regular overlap patterns, is promising, but as explained previously, the variant of it implemented in our current prototype is too simplistic too yield benefits in most practical scenarios. Further work will need to focus on improving this scheme, and evaluating its alternatives. Our future plan involves the following steps.

Making Systems Regular through Clustering of Objects

We have already mentioned ongoing research focused on partitioning the set of multicast topics in an irregular system into more regular subsets [300]. One line of work, currently in progress, focuses on understanding the patterns of overlap occurring in real systems, and devising clustering schemes to optimize for various simple performance metrics, such as the number of regions per node, etc. Another line of work will need to focus on deriving accurate analytical models and realistic performance metrics that could predict performance of complex systems, such as our multicast substrate described in Chapter 4, given a pattern of overlap, and that could be used to more precisely define the regularity metric in a given application domain. Such more accurate metrics could then be used to drive the partitioning algorithm. We have started working on a more accurate analytical model of performance of our multicast substrate, which could be used for this purpose.

Indexing Schemes for Amortizing Work Across Objects

As mentioned earlier, the ability of our multicast prototype to amortize traffic in a region across multiple multicast groups relies on the fact that senders can index their own messages on a per-region basis, a property that the architecture of Chapter 3 lacks due to a strict enforcement of the principle of encapsulation and decoupling of senders from receivers. As a result, although the Properties Framework can use shared structures, such as token rings, across groups (we have evaluated this in our simulations, results are reported in [238]), it still needs to maintain separate recovery records for each protocol instance, and the sizes of control packets and their processing times grow linearly with the number of overlapping protocol instances (also reported in [238]). In order to be able to amortize overhead, agents that represent members of a recovery domain would need to have access to what we call an *index*, a total ordering of message identifiers for a certain set of protocol instances P, and a certain set of senders Q. Having access to

such index, instead of carrying *PQ* recovery records, for each individual sender and protocol instance separately, agents could use a single recovery record to carry cumulative information. For example, if position k in the index represents message with number m generated by sender s in protocol p, *Received* is a "virtual" property stored in the token and circulated among agents, then $k \in Received$ would represent the receipt of message k from sender s in protocol p. Since each index k could map to a message in different protocol instance, and generated by a different sender, a single value of *Received* could effectively carry information about messages from multiple senders in multiple protocol instances simultaneously. Assuming that all agents have access to the common index, each of them would be able to encode such information into the token, and decode the computed aggregate. The index itself could be built by the group of agents on the fly, as a part of the aggregation protocol.

We believe that indexing schemes of this sort could be a very useful extension of our architecture, but this work is currently in the conceptual stage; it is unclear whether the computational overhead and the added complexity would justify the savings in the size of tokens and data structures. The answer will likely depend on how heavily and how regularly protocol instances overlap in practice.

6.2.3 Communication

Our existing multicast prototype works only in LANs. We have outlined an architecture that should make it possible to deploy live objects in WANs and wireless settings, but our architecture is currently not implemented, and there are a number of problem areas that we have not addressed at all. The most important next steps include the following.

Connecting Clients Separated with NATs and Firewalls

Although our model is compatible with data centers and client-server interactions, and the architecture of Chapter 3 involves a management infrastructure that could be rooted in data centers, we are assuming that the actual work performed by the protocols, such as transmitting data, peer-to-peer recovery, and other coordination, will be done directly between peers. If our technology is used in application scenarios of the sort described in Section 1.2, most clients will be home computers, hidden behind NAT boxes and firewalls, and unable to contact each other directly. To some degree, the architecture of Chapter 3 can address this; forwarding policies setup by scope managers could take connectivity into account and use techniques similar to "rendezvous peers" in JXTA [128], but this might only be possible in some scenarios. For a more general solution, we hope to adopt techniques such as NAT traversal [139], and leverage experience with applications such as Skype [271], to develop a suite of live objects specialized for discovery, naming, and dissemination through NATs and firewalls.

Incorporating Mechanisms for Privacy and Anonymity

Another concern common to nearly all peer-to-peer architectures is the question of privacy; in the architecture of Chapter 3, the main area of concern is the fact that nodes can trace each other's access to objects, and infer other user interest and browsing patterns. Hiding agent stacks deeper within the runtime, and even the operating system or networking hardware could help to make this type of information less accessible, but it could still easily be obtained by attackers with the help of packet capturing software.

One way to address this issue would be to leverage anonymity-preserving protocols that might involve dummy nodes and misleading routes to. In contrast, we plan to focus more on leveraging the isolation and encapsulation aspects of our architecture and leverage the fact that unlike in systems based on various hashing schemes or random

graphs, which cause arbitrary pairs of nodes to become peers, in our architecture peering relationships are formed within scopes that may impose explicit administrative policies. Thus, for example, for nodes in a corporate network, all direct peering relationships would be formed within the network, and agents that per with nodes outside the network might be placed on dedicated “gateway” nodes to mask any internal details regarding object access from the outsiders. This would lead to a sort of hierarchical anonymity, where the degree of privacy is a function of the distance in the hierarchy.

Embedding Content-Based Publish-Subscribe Objects

While group communication and topic-based publish-subscribe systems can be embedded cleanly into our model, by representing individual topics and groups as separate live objects, in case of content-based publish-subscribe systems the mapping is not obvious. One could integrate a content-based publish-subscribe system with our framework even today, by treating it as a single, monolithic object, in which content filtering constraints are expressed by each client by passing a special type of event to the endpoint it is connected to, but this would be a crude embedding, for it would not allow us to truly make use of the strong typing and component integration capabilities of our platform. On the other hand, one could consider representing each subscription as a separate live object, and the entire content-based publish-subscribe system as a network of interconnected objects, but it is unclear whether such a representation would be meaningful, and it would certainly lead to inflation in the size of references, an issue signaled earlier. The relationship between our model and content-based publish-subscribe is unclear, and the question of whether the two can be cleanly and efficiently integrated remains unresolved.

Infrastructure Objects in Support of Ad-Hoc Networks

Although our model is intended to support peer-to-peer applications, the architecture of Chapter 3 is more of a hybrid architecture, in which the peers are managed by a network of infrastructure nodes. To test the limits of our model, we'd like to provide support for scenarios where no infrastructure can be relied upon, and where clients literally have to build the entire protocol stack from scratch. Ad-hoc networking is a particularly clean example of this approach, and a useful one; the ability to bootstrap basic networking infrastructure in such settings, in a modular manner, out of the most primitive live objects, would open the way to a number of applications in sensor networks, urban sensing, etc. We are especially inspired by the WiPeer [117] project, and the sorts of collaborative applications it supports. We hope to port these ideas into our system, in the form of a library of live objects that provide simple ad-hoc routing, naming, or transport services.

6.2.4 Configuration

Although, as mentioned earlier, some technical details of the architecture of Chapter 3, such as detailed network protocols for the basic patterns of interaction between scopes and members, failure detection, or delegation of agents are missing, and the implementation is a work in progress, the work is well under way, and we do not expect to encounter major obstacles at this point. Assuming that our hierarchical management infrastructure will eventually be completed, the next steps will involve the following.

Application Model Based On Distributed Membership

As mentioned earlier, one of the novel aspects of the architecture of Chapter 3 is that it uses a decentralized membership model, and rather than relying on the clients to self-organize based on a globally uniform membership information, it leaves decisions such as which clients will play which roles, and at which levels in the hierarchy, to a hierar-

chical management infrastructure. This removes the need to distribute the membership information across the entire system, and opens the way to a better scalability.

We believe that this approach can be useful not only for dissemination, loss recovery, and the sorts of protocols we could support through the Properties Framework, but also in general, as a new model for structuring scalable services. In this approach, scalable services could be modeled as compositions of uniform, stackable building blocks (agents deployed by the management infrastructure), each of which would live in an abstract environment, similarly how we illustrated it on Figure 3.27.

We have already mentioned the analogy between agents and live object proxies. The code of such building blocks in a scalable service would thus be simply a live object's proxy code, and the scalable service in this model would be a hierarchical composition of a number of live objects laid over one another, and glued together to form a single whole using decentralized membership information maintained by the membership service of Chapter 3. This idea is essentially an extension of the Properties Framework approach.

We believe that a number of scalable services that might not be expressible in the Properties Language could still be implemented in this model, and that the architecture of Chapter 3 could be a useful runtime and development environment on its own, independently of the Properties Language. A developer building a service within such environment would be freed from worrying about building hierarchical structures needed for scalability, or assigning roles to individual nodes; the roles and structure would be given, and the only task left to the developer would be to fill the provided skeleton of a service with business logic. Unlike the Properties Framework, the logic might be coded in an imperative language; indeed, the Properties Framework could be built as a layer running on top of such environment. The key challenge here will be to formally model the properties of a decentralized membership service, and the guarantees this structure

can provide.

General-Purpose Scalable Role Delegation Framework

Although we believe that the Properties Language could be evolved into a very elegant, general programming model that is easy to understand and use, the management structure of Chapter 3 that it is based on does not currently have a sufficiently generic structure; rather, it is a fairly elaborate system that requires a fairly large amount of pre-existing infrastructure, and the language primitives of the Properties Framework have complex implementations. To address this problem, we have started working on modeling the architecture of Chapter 3 through live objects. A single layer in this infrastructure, consisting of a superscope, a set of subscopes, a protocol linking these, and the protocol through which the superscope delegates agents to its subscopes, could be viewed as a single live object, and the entire infrastructure as a set of live objects laid over one another. This resembles the idea just mentioned above, but here, the structure, rather than depending on an external entity managing it, would be created and administered by the users, and emerge as a result of user's administrative actions performed via drag and drop. Despite this, we hope to be able to represent the management infrastructure, and scalable services it might support, using the same uniform model (indeed, we are hoping that the management infrastructure may even be expressible in the Properties Language, although the current version of our language still is not powerful enough to enable this).

If such representation is possible, we hope to then evolve the system into a general-purpose role delegation framework. The live object that the management infrastructure is recursively composed from would represent a general-purpose tool that allows a functionality to be delegated from a single endpoint to a set of endpoints, and when recursively laid over itself, it could delegate a functionality of a single centralized service to

an arbitrarily large set of endpoints in a scalable manner. The main challenge is to design a generic structure of a delegation object that might play such universal role, design its interfaces, the way that a "role" to be delegated is expressed and represented, and the delegation object's types. We believe this work can lead to interesting and deep insights into live object types, and could serve as an ultimate test of our model and architecture.

Self-Discovery and Bootstrapping with Gossip Objects

Our work so far assumes that the management infrastructure that maintains distributed membership and assigns roles is deployed by the users, through drag and drop actions; eventually, as postulated above, this would happen within the live objects paradigm, by dragging and dropping delegation objects to deploy parts of the system on infrastructure nodes. If the system becomes popular, however, administering it manually will become infeasible. We are exploring options for making this process fully automated by leveraging gossip protocols. The first step, already a work in progress, will be to develop a suite of live objects implemented as scalable gossip protocols that could be used in a purely peer-to-peer fashion, and that could compute simple data structures, such as graphs of network topology, annotated with latencies, node uptimes, and other structural information, and with some probabilistic consistency guarantees. These objects could then be used, in a semi-automated manner, by the management infrastructure, e.g., to assign and migrate the scope manager role within a scope, or to determine scope boundaries automatically within a wide-area network. Some of the high-level concepts have already been laid out [45], but much work remains to be done.

6.3 Summary

In this dissertation, we have proposed a vision for a new style of distributed systems development where distributed protocols are treated as first-class objects, a vision that, as

we postulated in the introduction, the Web is likely heading towards. We have described the programming model and multicast architecture supporting it, and evaluated our ideas with two prototype systems. Our work is just a small step, and as the discussion of this section demonstrates, there are many problems that need to be solved to really make the system feasible in practice; some of these problems raise fundamental research questions, and some are simply engineering challenges. Nevertheless, the platform we have already built is working and quite versatile, and the goal seems within reach. We hope that our work will inspire other researchers to pursue the live objects vision.

BIBLIOGRAPHY

[1] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Trans. Database Syst.*, 14(2):264–290, 1989.

[2] Luca Aceto and Andrew D. Gordon. Algebraic process calculi: The first twenty five years and beyond. *Process Algebra. http://www.brics.dk/NS/05/3/BRICS-NS-05-3.pdf*, 2005.

[3] Activeworlds Corporation. Active worlds. *http://www.activeworlds.com/*, 2008.

[4] B. Adamson, C. Bormann, M. Handley, and J. Macker. Nack-oriented reliable multicast protocol (norm). *2004.*, 2004.

[5] Adobe. Adobe air. *http://www.adobe.com/products/air/*, 2008.

[6] Adobe. Adobe flex. *http://labs.adobe.com/technologies/flex/*, 2008.

[7] Gul Agha. Actors: A model of concurrent computation in distributed systems. *Doctoral Dissertation. MIT Press.*, 1986.

[8] Jeremy Allaire. Macromedia flash mx: A next-generation rich client. *http://download.macromedia.com/pub/flash/whitepapers/richclient.pdf*, 2002.

[9] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[10] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. Eden system: A technical review. *IEEE Transactions on Software Engineering. Vol. SE-11, no. 1, pp. 43-59. 1985*, 1985.

[11] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[12] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In *European conference on object-oriented programming on ECOOP '87*, pages 234–242, London, UK, 1987. Springer-Verlag.

[13] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The spread toolkit: Architecture and performance. *Technical Report CNDS-2004-1.*, 2004.

[14] Emmanuelle Anceaume, Bernadette Charron-Bost, Pascale Minet, and Sam Toueg. On the formal specification of group membership services. Technical Report TR95-1534, 25, 1995.

[15] David Pope Anderson. *A grammar-based methodology for protocol specification and implementation*. PhD thesis, 1985.

[16] Brent Ashley. Ashley it remote scripting sources... and home of jsrs. *http://www.ashleyit.com/rs/*, 2007.

[17] Ryan Asleson and Nathaniel T. Schutta. *Foundations of Ajax (Foundation)*. Apress, Berkely, CA, USA, 2005.

[18] J. Bacon et al. Generic support for distributed applications. *Computer, vol. 33, no. 3, Mar. 2000, pp. 6876*, 2000.

[19] L. Baduel, F. Baude, N. Ranaldo, and E. Zimeo. Effective and efficient communication in grid computing with an extension of proactive groups. *Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium, Colorado, 2005.*, 2005.

[20] B. Ban. Design and implementation of a reliable group communication toolkit for java. *Cornell University, September 1998.*, 1998.

[21] B. Ban. Performance tests jgroups 2. *5. June 2007. http://www.jgroups.org/javagroupsnew/perfnew/Report.html*, 2007.

[22] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. *In Proceedings of the 19th IEEE international Conference on Distributed Computing Systems. (ICDCS'99). IEEE Computer Society, Washington, DC, 262.*, 1999.

[23] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. *ACM SIGCOMM, Aug. 2002.*, 2002.

[24] Arindam Banerji, Claudio Bartolini, Dorothea Beringer, Venkatesh Chopella, Kannan Govindarajan, Alan Karp, Harumi Kuno, Mike Lemon, Gregory Pogossiants, Shamik Sharma, and Scott Williams. Web services conversation language (wscl) 1.0. *http://www.w3.org/TR/wscl10/*, 2002.

[25] J.S. Banino, A. Caristan, M. Guillemont, G. Morisset, and H. Zimmermann. Cho-

rus: An architecture for distributed systems. *HAL: inria-00076519, version 1*, 1980.

[26] P. Baran. On distributed communications networks. *Communications, IEEE Transactions on [legacy, pre - 1988]*, 12(1):1–9, Mar 1964.

[27] M. P. Barcellos, M. Nekovee, M. Daw, J. Brooke, and S. Olafsson. Reliable multicast for the grid: a comparison of protocol implementations. *Proceedings of the UK E-Science All Hands Meeting, Nottingham (UK), 2004.*, 2004.

[28] M. P. Barcellos, M. Nekovee, M. Koyabe, M. Daw, and J. Brooke. Evaluating high-throughput reliable multicast for grid applications in production networks. *In Proceedings of the Fifth IEEE international Symposium on Cluster Computing and the Grid (Ccgrid'05) - Volume 1 - Volume 01 (May 09 - 12, 2005). CCGRID. IEEE Computer Society, Washington, DC, 442-449.*, 2005.

[29] A. Basu, M. Hayden, G. Morrisett, and T. von Eicken. A language-based approach to protocol construction. In *Proc. ACM SIGPLAN Workshop on Domain Specific Languages*, January 1997.

[30] Anindya Basu, J. Gregory Morrisett, and Thorsten von Eicken. Promela++: A language for constructing correct and efficient protocols. In *INFOCOM (2)*, pages 455–462, 1998.

[31] C. Begg and T. Connolly. Database systems: A practical approach to design, implementation, and management. *Pearson Education, England*, 2002.

[32] Jan A. Bergstra, Jan Willem Klop, and J. V. Tucker. Process algebra with asynchronous communication mechanisms. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 76–95, London, UK, 1985. Springer-Verlag.

[33] Berkman Center. Rss 2.0 specification. *http://cyber.law.harvard.edu/rss/rss.html*, 2003.

[34] T. Berners-Lee and D. Connolly. Hypertext markup language - 2.0, 1995.

[35] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[36] A. Bharambe, S. Rao, V. Padmanabhan, S. Seshan, and H. Zhang. The impact of heterogeneous bandwidth constraints on dht-based multicast protocols, 2005.

[37] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: a system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.*, 16(4):321–366, 1998.

[38] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian Milnes. Signatures for a network protocol stack: A systems application of standard ML. In *LISP and Functional Programming*, pages 55–64, 1994.

[39] M. Bickford and J. Hickey. An object-oriented approach to verifying group communication systems, 1998.

[40] K. Birman. The process group approach to reliable distributed computing. *Comm. of the ACM 36, 12 (Dec 1993), pp. 37-53.*, 1993.

[41] K. Birman. Reliable distributed systems. *Springer Verlag, 2005.*, 2005.

[42] K. Birman and T. Clark. Performance of the isis distributed computing toolkit. *Tech. Report TR-94-1432, Dept. of Computer Science, Cornell University.*, 1994.

[43] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM TOCS, Vol. 17, No. 2, pp 41-88, May, 1999.*, 1999.

[44] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *In Proceedings of the 11th ACM SOSP; Austin, Texas, Nov. 1987, 123-138.*, 1987.

[45] K. Birman, A.-M. Kermarrec, K. Ostrowski, M. Bertier, D. Dolev, and R. van Renesse. Exploiting gossip for self-management in scalable event notification systems. *ICDCS'07 Workshop on Distributed Event Processing Systems and Architecture (DEPSA). Jun 2007.*, 2007.

[46] K.P. Birman, T.A. Joseph, T. Raeuchle, and A. El Abbadi. Implementing fault-tolerant distributed objects. *Software Engineering, IEEE Transactions on*, SE-11(6):502–508, June 1985.

[47] A. Birrell, G. Nelson, S. Owicki, and W. Wobber. Network objects. *In Proc. of the 14th ACM SOSP, pp. 217-230, Asheville, NC (USA), Dec 1993.*, 1993.

[48] Stefan Birrer and Fabian E. Bustamante. The feasibility of dht-based streaming multicast. *mascots*, 0:288–298, 2005.

[49] Blizzard Entertainment. World of warcraft. *http://www.worldofwarcraft.com*, 2008.

[50] U. Borghoff and G. Tegge. Application of collaborative editing to software-engineering projects, 1993.

[51] C. Bormann, J. Ott, H.-C. Gehrcke, T. Kerschat, and N. Seifert. Mtp-2: Towards achieving the s. *E.R.O. Properties for Multicast Transport. International Conference on Computer Communications Networks, San Francisco, California, September 1994.*, 1994.

[52] D. Box, L. F. Cabrera, C. Critchley, D. Curbera, D. Ferguson, A. Geller, et al. Web services eventing (ws-eventing). *http://www.ibm.com/developerworks/webservices/library/specification/ws-eventing/*, 2004.

[53] Don Box. *Essential COM*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. Foreword By-Grady Booch and Foreword By-Charlie Kindel.

[54] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.

[55] Ed Brinksma. A tutorial on lotos. In *Proceedings of the IFIP WG6.1 Fifth International Conference on Protocol Specification, Testing and Verification V*, pages 171–194, Amsterdam, The Netherlands, The Netherlands, 1985. North-Holland Publishing Co.

[56] J.-P. Briot, R. Guerraoui, and K. P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys, Sep 1998.*, 1998.

[57] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM TOCS, Vol. 8, No. 1, Feb 1990, pp. 18-36.*, 1990.

[58] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[59] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. *Proceedings of ACM SIGCOMM, 1998, pp. 56–67.*, 1998.

[60] L. F. Cabera, M. B. Jones, and M. Theimer. Herald: Achieving a global event

notification service. *In Proceedings of Workshop on Hot Topics in Operating Systems, Elmau, Germany, May 2001.*, 2001.

[61] Felipe Cabrera, George Copeland, Tom Freund, Johannes Klein, David Langworthy, David Orchard, John Shewchuk, and Tony Storey. Web services coordination (ws-coordination). *http://msdn2.microsoft.com/en-us/library/ms951231.aspx*, 2002.

[62] L. F. Cabrera et al. Web services atomic transaction (ws-atomictransaction). *http://specs.xmlsoap.org/ws/2004/10/wsat/wsat.pdf*, 2005.

[63] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *FoSSaCS '98: Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*, pages 140–155, London, UK, 1998. Springer-Verlag.

[64] N. Carriero and D. Gelernter. Linda in context. *CACM 32, 4 (Apr 1989), pp. 444-458.*, 1989.

[65] J.B. Carter, D. Khandekar, and L. Kamb. Distributed shared memory: where we are and where we should be headed. *hotos*, 00:119, 1995.

[66] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems, 19(3):332-383, Aug 2001.*, 2001.

[67] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM.

[68] M. Castro, P. Druschel, Y. Hu, and A. Rowstron. Exploiting network proximity in distributed hash tables, 2002.

[69] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. *SOSP'03, Lake Bolton, NY, October, 2003.*, 2003.

[70] M. Castro, P. Druschel, A.-M. Kermarrec, and A.I.T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, Oct 2002.

[71] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM.

[72] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[73] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[74] J. M. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems, vol. 2, no. 3, Aug. 1984, pp. 251–73.*, 1984.

[75] David Chen and Chengzheng Sun. Comparison of real-time text chat and collaborative editing systems. *Lecture Notes in Computer Science, Volume 3190/2004, Springer Berlin / Heidelberg*, 2004.

[76] D. Cheriton and W. Zwaenepoel. Distributed process groups in the v kernel. *ACM TOCS 3, 2 (May 1985), pp. 77-107.*, 1985.

[77] Roger S. Chin and Samuel T. Chanson. Distributed, object-based programming systems. *ACM Comput. Surv.*, 23(1):91–124, 1991.

[78] D. M. Chiu, S. Hurst, J. Kadansky, and J. Wesley. Tram: A tree-based reliable multicast protocol. *Sun Microsystems Laboratories Technical Report Series, TR-98-66, 1998.*, 1998.

[79] G. Chockler, I. Keidar, and W. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computer Surveys, 33(4):1, pp. 43, Dec 2001.*, 2001.

[80] Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 14–25, New York, NY, USA, 2007. ACM.

[81] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1. *1. http://www.w3.org/TR/wsdl.*, 2001.

[82] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. A case for end system multicast.

IEEE Journal on Selected Areas in Communication (JSAC), Special Issue on Networking Support for Multicast, Vol. 20, No. 8, 2002., 2002.

[83] Andrew Clinick. Remote scripting. *http://msdn2.microsoft.com/en-us/library/ms974566.aspx*, 1999.

[84] Bram Cohen. Incentives build robustness in bittorrent, 2003.

[85] Croquet Consortium. Cobalt. *http://www.opencroquet.org/index.php/Cobalt*, 2008.

[86] Eric Costello et al. Remote scripting with iframe. *http://developer.apple.com/internet/webcontent/iframe.html*, 2002.

[87] William Cox, Felipe Cabrera, George Copeland, Tom Freund, Johannes Klein, Tony Storey, and Satish Thatte. Web services transaction (ws-transaction). *http://dev2dev.bea.com/pub/a/2004/01/ws-transaction.html*, 2004.

[88] Stewart Crawford and Elizabeth Boese. Actionscript: a gentle introduction to programming. *J. Comput. Small Coll.*, 21(3):156–168, 2006.

[89] O.-J. Dahl and K. Nygaard. Class and subclass declarations. *IFIP Working Conference on Simulation Programming Languages, Oslo May 1967.*, 1967.

[90] Partha Dasgupta, Raymond C. Chen, Sathis Menon, Mark P. Pearson, R. Ananthanarayanan, Umakishore Ramachandran, Mustaque Ahamad, Richard J. LeBlanc, William F. Appelbe, Jose M. Bernabn, Phillip W. Hutto, M. Yousef Amin Khalidi, and C. J. Wilkenloh. The design and implementation of the clouds distributed operating system. *Computing Systems*, 3(1):11–46, 1989.

[91] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.

[92] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. *Lecture Notes in Computer Science, Volume 3896/2006*, 2006.

[93] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale xml dissemination service (2004). 2004.

[94] C. Diot, B.N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment

issues for the ip multicast service and architecture. *Network, IEEE*, 14(1):78–88, Jan/Feb 2000.

[95] D. Dolev and D. Malkhi. The transis approach to high availability cluster communication. *Communications of the ACM 39(4), April 1996, pp 87-92.*, 1996.

[96] S. Donatelli, J. Hillston, and M. Ribaudo. A comparison of performance evaluation process algebra and generalized stochastic petri nets. *6th International Workshop on Petri Nets and Performance Models, Durham, North Carolina, 1995.*, 1995.

[97] ECMA. ECMAScript language specification, December 1999. ECMA Standard 262, 3rd Edition.

[98] ECMA International. Standard ecma-335 common language infrastructure (cli). *http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf*, 2006.

[99] Clarence A. Ellis, Simon J. Gibbs, and Gail Rein. Groupware: some issues and experiences. *Commun. ACM*, 34(1):39–58, 1991.

[100] Ernesto. Bittorrent continues to dominate internet traffic. *http://torrentfreak.com/bittorrent-dominates-internet-traffic-070901/*, 2007.

[101] Warren Ernst and John J. Kottler. *Presenting Active X.* Sams.Net Publishing, 1996.

[102] M. R. Eskicioglu. A comprehensive bibliography of distributed shared memory. *http://www.cs.umd.edu/k̃eleher/bib/dsmbiblio/dsmbiblio.html*, 1995.

[103] P. Eugster, C. Damm, and R. Guerraoui. Towards safe distributed application development. *In Proceedings of the 26th International Conference on Software Engineering (May 23 - 28, 2004).*, 2004.

[104] P. Eugster and R. Guerraoui. Distributed programming with types events. *IEEE Software, 2004.*, 2004.

[105] P. Eugster, R. Guerraoui, and C. H. Damm. On objects and events. *OOPSLA 2001.*, 2001.

[106] P. Eugster, R. Guerraoui, and J. Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. *ECOOP 2000, pp. 252-276.*, 2000.

[107] P. Th. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *dsn*, 00:0443, 2001.

[108] Erik Fair. Usenet, spanning the globe. *Unix/World, 1 (November, 1984), pp. 46-49*, 1984.

[109] Aiguo Fei, Junhong Cui, Mario Gerla, and Michalis Faloutsos. Aggregated multicast with inter-group sharing. *Networked Group Communications*, 2001.

[110] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http, 1999.

[111] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[112] David Flanagan. *JavaScript: the definitive guide*. Fourth edition, 2002.

[113] S. Floyd, V. Jacobson, C. Liu, S. Mccanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking 5, 6 (December 1997), 784-803.*, 1997.

[114] E. Freeman, S. Hupfer, and K. Arnold. Java-spaces principles, patterns, and practice. *Addison-Wesley*, 1999.

[115] G. Frehse, Zhi Han, and B. Krogh. Assume-guarantee reasoning for hybrid i/o-automata by over-approximation of continuous interaction. *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, 1:479–484 Vol.1, 14-17 Dec. 2004.

[116] A. O. Freier and K. Marzullo. Mtp: An atomic multicast transport protocol, technical report, no. *90-1141, Computer Science Department, Cornell University.*, 1990.

[117] Roy Friedman, Vadim Drabkin, Gabriel Kliot, et al. Wipeer. *http://www.wipeer.com/*, 2006.

[118] Rachele Fuzzati and Uwe Nestmann. Much ado about nothing? *Algebraic Process Calculi: The First Twenty Five Years and Beyond. Process Algebra. http://www.brics.dk/NS/05/3/BRICS-NS-05-3.pdf*, 2005.

[119] X. Gabaix, P. Gopikrishnan, V. Plerou, and H. E. Stanley. A theory of power-law distributions in financial market fluctuations. *Nature 423, 267-270. May 2003.*, 2003.

[120] Syam Gadde, Jeffrey S. Chase, and Michael Rabinovich. Web caching and content distribution: a view from the interior. *Computer Communications*, 24(2):222–231, 2001.

[121] B. Garbinato, P. Febler, and R. Guerraoui. Protocols classes for designing reliable distributed environments. *ECOOP 1996.*, 1996.

[122] B. Garbinato and R. Guerraoui. Using the strategy pattern to compose reliable distributed protocols. *In Proce. of 3rd USENIX COOTS, Portland, OR, Jun 1997.*, 1997.

[123] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, 1985.

[124] David K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, New York, NY, USA, 1979. ACM.

[125] B. Glade, K. Birman, R. Cooper, and R. van Renesse. Light-weight process groups in the isis system. *Distributed Systems Engineering, 1(1):29-36, Sep 1993.*, 1993.

[126] S. Glassman. A caching relay for the world wide web, 1994.

[127] A. Goldberg and D. Robson. Smalltalk-80: the language and its implementation. *Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA. 1983.*, 1983.

[128] Li Gong. Jxta: a network programming environment. *Internet Computing, IEEE*, 5(3):88–95, May/Jun 2001.

[129] Li Gong, Roger Needham, and Raphael Yahalom. Reasoning About Belief in Cryptographic Protocols. In Deborah Cooper and Teresa Lunt, editors, *Proceedings 1990 IEEE Symposium on Research in Security and Privacy*, pages 234–248. IEEE Computer Society, 1990.

[130] Google. google-caja. *http://code.google.com/p/google-caja/*, 2008.

[131] Google. Google Web Toolkit. *http://code.google.com/webtoolkit/*, 2008.

[132] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in lcf. In *POPL '78: Proceedings of the 5th ACM SIGACT-*

SIGPLAN symposium on Principles of programming languages, pages 119–130, New York, NY, USA, 1978. ACM.

[133] J. Gosling, B. Joy, G. Steele, and G. Bracha. The java language specification second edition, 2000.

[134] S. Graham, P. Niblett, D. Chappell, A. Lewis, N. Nagaratnam, J. Parikh, et al. Web services brokered notification (ws-brokerednotification). *http://www.ibm.com/developerworks/library/specification/ws-notification/*, 2004.

[135] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[136] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[137] Irene Greif, Robert Seliger, and William E. Weihl. Atomic data abstractions in a distributed collaborative editing system. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 160–172, New York, NY, USA, 1986. ACM.

[138] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. Soap version 1.2 part 1: Messaging framework (second edition). *http://www.w3.org/TR/soap12/*, 2007.

[139] Saikat Guha and Paul Francis. Characterization and measurement of tcp traversal through nats and firewalls. In *IMC'05: Proceedings of the Internet Measurement Conference 2005 on Internet Measurement Conference*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[140] M. Haahr et al. Filtering and scalability in the eco distributed event model. *International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000), IEEE CS Press, 2000, pp. 8392*, 2000.

[141] J. Halpern, R. Fagin, Y. Moses, and M. Vardi. Reasoning about knowledge. *MIT Press, 1995.*, 1995.

[142] M. Handley, S. Floyd, B. Whetten, R. Kermode, L. Vicisano, and M. Luby. The reliable multicast design space for bulk data transfer. *Internet informational RFC 2887, August 2000*, 2000.

[143] C. Hänle and M. Hofmann. Performance comparison of reliable multicast protocols using the network simulator ns-2. In *LCN '98: Proceedings of the 23rd Annual IEEE Conference on Local Computer Networks*, page 222, Washington, DC, USA, 1998. IEEE Computer Society.

[144] T. Harrison, D. Levine, and D.C. Schmidt. The design and performance of a real-time corba event service. *12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 97), ACM Press, 1997, pp. 184200*, 1997.

[145] M. Hayden. The ensemble system. *Ph.D. dissertation, Cornell University Dept. of Computer Science, 1998.*, 1998.

[146] Matthew Hennessey and James Riely. Type-safe execution of mobile agents in anonymous networks. pages 95–115, 1999.

[147] Thomas A. Henzinger, Marius Minea, and Vinayak Prabhu. Assume-guarantee reasoning for hierarchical hybrid systems. In *HSCC '01: Proceedings of the 4th International Workshop on Hybrid Systems*, pages 275–290, London, UK, 2001. Springer-Verlag.

[148] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. *IJCAI*, 1973.

[149] Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for ensemble layers. In R. Cleaveland, editor, *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 1579, pages 119–133. Springer-Verlag, Berlin Germany, 1999.

[150] M. Hidell and P. Sjdin. Performance of nack-oriented reliable multicast in distributed routers. *Proceedings of the 2006 IEEE Workshop on High Performance Switching and Routing (HPSR), Poznan, Poland, June 2006.*, 2006.

[151] M. Hiltunen and R. Schlichting. The cactus approach to building configurable middleware services, 2000.

[152] M.A. Hiltunen and R.D. Schlichting. A configurable membership service. *Computers, IEEE Transactions on*, 47(5):573–586, May 1998.

[153] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[154] H. W. Holbrook, S. K. Singhal, and D. R. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. *SIGCOMM Comput. Commun. Rev. 25, 4 (Oct. 1995), 328-341.*, 1995.

[155] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[156] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[157] B. Hudzia and S. Petiton. Reliable multicast fault tolerant mpi in the grid environment. *International Conference GRIDnet, October 2004.*, 2004.

[158] Michael N. Huhns and Munindar P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[159] M. Humphrey and G. Wasson. Architectural foundations of wsrf.net. *International Journal of Web Services Research. 2(2), pp. 83-97, April-June 2005.*, 2005.

[160] Norman C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng.*, 17(1):64–76, 1991.

[161] IBM, BEA Systems, Microsoft, SAP AG, and Siebel Systems. Business process execution language for web services version 1.1. *http://www-128.ibm.com/developerworks/library/specification/ws-bpel/*, 2007.

[162] Nonnenmacher J. and E. Biersack. Optimal multicast feedback. *In Proceedings of IEEE INFOCOM, San Francisco, CA, USA, March 1998.*, 1998.

[163] Paul Jackson. The nuprl proof development system (version 4.2) reference manual and user's guide. *Unpublished manuscript, Cornell University, 1996*, 1996.

[164] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole. Overcast: Reliable multicasting with an overlay network. *In Proc. of the 4th OSDI, Volume 4 (San Diego, California, October 22 - 25, 2000). USENIX Association, Berkeley, CA, 14-14.*, 2000.

[165] K. Jeacle and J. Crowcroft. Reliable high-speed grid data delivery using ip multicast. *Proceedings of UK E-Science All Hands Meeting, UK, September, 2003.*, 2003.

[166] Bengt Jonsson. Compositional specification and verification of distributed systems. *ACM Trans. Program. Lang. Syst.*, 16(2):259–303, 1994.

[167] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM TOCS, 6(1), July 1988, pp. 109-133.*, 1988.

[168] Brian Kantor and Phil Lapsley. Network news transfer protocol: A proposed standard for the stream-based transmission of news. *RFC 977*, 1986.

[169] David Karr. Specification, composition, and automated verification of layered communication protocols. *Doctoral dissertation. Technical Report number TR97-1623. Department of Computer Science, Cornell University, Ithaca, NY, March 1997.*, 1997.

[170] Idit Keidar and Danny Dolev. Increasing the resilience of atomic commit at no additional cost. In *Symposium on Principles of Database Systems*, pages 245–254, 1995.

[171] Idit Keidar and Roger Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. In *International Conference on Distributed Computing Systems*, pages 344–355, 2000.

[172] Idit Keidar, Roger I. Khazan, Nancy Lynch, and Alex Shvartsman. An inheritance-based technique for building simulation proofs incrementally. *ACM Trans. Softw. Eng. Methodol.*, 11(1):63–91, 2002.

[173] Spencer Kelly. Identity at risk on facebook. *http://news.bbc.co.uk/2/hi/programmes/click_online/7375772.stm*, 2008.

[174] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. *ECOOP'97, vol.1241, pp. 220-242.*, 1997.

[175] Michael J. Knister and Atul Prakash. Distedit: a distributed toolkit for supporting multiple group editors. In *CSCW '90: Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, pages 343–355, New York, NY, USA, 1990. ACM.

[176] A. Koifman and S. Zabele. Ramp: A reliable adaptive multicast protocol, in ieee infocom'96, pp. *1442–1451, March 1996.*, 1996.

[177] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh, 2003.

[178] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, New York, NY, USA, 2007. ACM.

[179] Bent Bruun Kristensen and Daniel C. M. May. Activities: Abstractions for collective behavior. In *ECCOP '96: Proceedings of the 10th European Conference on Object-Oriented Programming*, pages 472–501, London, UK, 1996. Springer-Verlag.

[180] C. Krumvieda. Distributed ml: Abstractions for efficient and fault-tolerant prgramming. *Technical Report, TR93-1376, Cornell University (1993).*, 1993.

[181] M. Lacher, J. Nonnenmacher, and E. Biersack. Performance comparison of centralized versus distributed error recovery for reliable multicast. *IEEE/ACM Transactions on Networking 8 (2) (2000) 224–238.*, 2000.

[182] L. Lamport. The temporal logic of actions. *ACM Toplas 16, 3 (May 1994), pp. 872-923.*, 1994.

[183] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) 51-58.*, 2001.

[184] R. G. Lane, D. Scott, and X. Yuan. An empirical study of reliable multicast protocols over ethernet-connected networks. *Perform. Eval. 64, 3 (Mar. 2007), 210-228.*, 2007.

[185] B. N. Levine and J. J. Garcia-Luna-Aceves. A comparison of reliable multicast protocols. *Multimedia Systems 6: 334-348, 1998.*, 1998.

[186] B. N. Levine, D. B. Lavo, and J. J. Garcia-Luna-Aceves. The case for reliable concurrent multicasting using shared ack trees. *In ACM Multimedia (1996), pp. 365–376.*, 1996.

[187] Frank Leymann. Web services flow language (wsfl 1.0). *http://xml.coverpages.org/WSFL-Guide-200110.pdf*, 2001.

[188] Kai Li. A shared virtual memory system for parallel computing. 1988.

[189] Linden Research, Inc. SecondLife. *http://secondlife.com/*, 2003.

[190] B. Liskov. Distributed programming in argus. *CACM 31, 3 (Mar. 1988), pp. 300-312.*, 1988.

[191] B. Liskov and R. Schieffler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM TOPLAS 5:3 (July 1983).*, 1983.

[192] J. D. C. Little. A proof of the queueing formula. *Operations Research, 9, 383-387 (1961)*, 1961.

[193] H. Liu, V. Ramasubramanian, and E. G. Sirer. Characteristics of rss, a publish-subscribe system for web micronews. *Client and Feed In Proceedings of Internet Measurement Conference (IMC), Berkeley, California, October 2005.*, 2005.

[194] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. *In Proc. of the 17th ACM Symposium on Operating System Principles (SOSP 1999), pp. 80-92.*, 1999.

[195] Chris Loosley. Rich internet applications: Design, measurement, and management challenges. *http://www.keynote.com/docs/whitepapers/RichInternet_5.pdf*, 2006.

[196] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. *PODC'87, pp. 137-151.*, 1987.

[197] J. P. Macker and P. B. Adamson. The multicast dissemination protocol (mdp) toolkit. *Military Communications Conference Proceedings, 1999. MILCOM 1999. IEEE.*, 1999.

[198] S. Maffeis and D. Schmidt. Constructing reliable distributed communication systems with corba. *IEEE Communications Magazine feature topic issue on Distributed Object Computing, Vol. 14, No. 2, Feb 1997.*, 1997.

[199] M. Maimour and C. Pham. An active reliable multicast framework for the grids. *In Proceedings of the international Conference on Computational Science-Part II (April 21 - 24, 2002). Lecture Notes In Computer Science, vol. 2330. Springer-Verlag, London, 588-597.*, 2002.

[200] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro.

Fragmented objects for distributed abstractions, pages 170–186. IEEE Computer Society Press, 1994.

[201] John Markoff. Why cant we compute in the cloud? *The New York Times. August 24, 2007. http://bits.blogs.nytimes.com/2007/08/24/why-cant-we-compute-in-the-cloud/*, 2007.

[202] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry R. Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. Owl-s: Semantic markup for web services, November 2004.

[203] Antoni W. Mazurkiewicz. Trace theory. In *Proceedings of an Advanced Course on Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986-Part II*, pages 279–324, London, UK, 1987. Springer-Verlag.

[204] S. Mccanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. *In Conference Proceedings on Applications, Technologies, Architectures and Protocols For Computer Communications (Palo Alto, California, United States, August 28 - 30, 1996). M. Steenstrup, Ed. SIGCOMM '96. ACM Press, New York, NY, 117-130.*, 1996.

[205] P. Mckinley, R. T. Rao, and R. F. Wright. H-rmc: A hybrid reliable multicast protocol in the linux kernel. *Tech. Rep. MSU-CPS-99-22, Department of Computer Science, Michigan State, East Lansing, Michigan, April 1999.*, 1999.

[206] S. Mena, X. Cuvellier, C. Gregoire, and A. Schiper. Appia vs. cactus: comparing protocol composition frameworks. *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 189–198, 6-18 Oct. 2003.

[207] Microsoft. Microsoft message queueing. *http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx*, 2003.

[208] Microsoft. Microsoft office groove 2007. *http://office.microsoft.com/en-us/groove/FX100487641033.aspx?ofcresset=1*, 2008.

[209] Microsoft. Silverlight. *http://silverlight.net/*, 2008.

[210] Microsoft. Sos debugging extension. *http://msdn2.microsoft.com/en-us/library/bb190764(VS.80).aspx*, 2008.

[211] Sun Microsystems. Javafx technology - at a glance. *http://java.sun.com/javafx/*, 2008.

[212] N. Milanovic and M. Malek. Current solutions for web service composition. *Internet Computing, IEEE*, 8(6):51–59, Nov.-Dec. 2004.

[213] K. Miller, K. Robertson, M. White, and A. Tweedly. Starburst multicast file transfer protocol (mftp) specification. *Internet Draft, Internet Engineering Task Force, April 1998.*, 1998.

[214] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[215] R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[216] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. Technical Report -86, 1989.

[217] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. *In Proc. of 21st ICDCS, pp. 707-10, Phoenix, Arizona, 2001.*, 2001.

[218] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Experience with modularity in consul. *Software - Practice and Experience*, 23(10):1059–1075, 1993.

[219] T. Montgomery, B. Whetten, M. Basavaiah, S. Paul, N. Rastogi, J. Conlan, and T. Yeh. The rmtp-ii protocol. *Internet Draft, Internet Engineering Task Force. April 1998.*, 1998.

[220] A. Montresor, R. Davoli, and O. Babaoglu. Enhancing jini with group communication. *Distributed Computing Systems Workshop, 2001 International Conference on*, pages 69–74, Apr 2001.

[221] Rajas Moonka, Peter C. Chane, Manish Gupta, and Nicholas Lee. Using viewing signals in targeted video advertising. *United States Patent Application 20080066107*, 2008.

[222] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, et al. Totem:

A fault-tolerant multicast group communication system. *Communications of the ACM, 39(4), 54-63p*, 1996.

[223] S.J. Mullender, G. van Rossum, A.S. Tananbaum, R. van Renesse, and H. van Staveren. Amoeba: a distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[224] Elie Najm and Frank Olsen. Protocol verification with reactive promela/rspin, 1996.

[225] National LambdaRail. National lambdarail. *http://www.nlr.net/*, 2008.

[226] G. C. Necula. Proof-carrying code. *In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Paris, France, January 15 - 17, 1997). POPL '97.*, 1997.

[227] NetFlix. Netflix. *http://www.netflix.com*, 2008.

[228] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 111–120, New York, NY, USA, 1995. ACM.

[229] E. Di Nitto and D. Rosenblum. the role of style in selecting middleware and underwear, 1999.

[230] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52–60, Aug 1991.

[231] J. Nonnenmacher, E. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. *In SIGCOMM '97, pp. 289–300, Cannes, France, September 1997.*, 1997.

[232] R. J. Oberg. Understanding & programming com+. *Prentice Hall*, 2000.

[233] Object Management Group. Notification service standalone document. 2000.

[234] Object Management Group. Event service specification. *CORBAservices: Common Object Services Specification*, 2001.

[235] K. Obraczka. Multicast transport protocols: a survey and taxonomy. *IEEE Communications Magazine 36(1): 94-102, Jan 1998.*, 1998.

[236] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: An architecture for extensible distributed systems. *ACM SOSP, 1993.*, 1993.

[237] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems, 10(2):110-143, May 1992.*, 1992.

[238] K. Ostrowski, K. Birman, and D. Dolev. The quicksilver properties framework. *OSDI'06 Poster Session, Seattle, WA, November 2006.*, 2006.

[239] Jeremiah Owyang. Social network stats: Facebook, myspace, reunion (jan, 2008). *http://www.web-strategist.com/blog/2008/01/09/social-network-stats-facebook-myspace-reunion-jan-2008/*, 2008.

[240] François Pacull, Alain Sandoz, and André Schiper. Duplex: a distributed collaborative editing environment in large scale. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 165–173, New York, NY, USA, 1994. ACM.

[241] Venkata N. Padmanabhan and Lili Qui. The content and access dynamics of a busy web site: findings and implicatins. In *SIGCOMM*, pages 111–123, 2000.

[242] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and Mohr. A. E. Chainsaw: Eliminating trees from overlay multicast (2005). 2005.

[243] C. Papadopoulos and G. Parulkar. Implosion control for multipoint applications. *In Proceedings of the 10th Annual IEEE Workshop on Computer Communications, Sept. 1995.*, 1995.

[244] C. Papadopoulos, G. Parulkar, and G. Varghese. An error control scheme for large-scale multicast applications. *In Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing (Puerto Vallarta, Mexico, June 28 - July 02, 1998). PODC '98. ACM, New York, NY, 310.*, 1998.

[245] S. Parastatidis, J. Webber, S. Woodman, D. Kuo, and P. Greenfield. Soap service description language (ssdl). *Technical Report, University of Newcastle, CS-TR-899, 2005.*, 2005.

[246] Joachim Parrow and Bjrn Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. *Proceedings of LICS'98*, 1998.

[247] Adam Pasick. Livewire - file-sharing network thrives beneath the radar. *Yahoo! News. http://in.tech.yahoo.com/041103/137/2ho4i.html*, 2003.

[248] S. Paul, K. K. Sabnani, J. C.-H. Lin, and S. Bhattacharyya. Reliable multicast transport protocol (rmtp). *IEEE Journal on Selected Areas in Communications, special issue on Network Support for Multipoint Communication, 15(3):407-421.*, 1997.

[249] Mark O. Pendergast and Doug Vogel. Design and implementation of a pc/lan-based multi-user text editor. In *Proceedings of the IFIP WG 8.4 confernece on Multi-user interfaces and applications*, pages 195–206, Amsterdam, The Netherlands, The Netherlands, 1990. Elsevier North-Holland, Inc.

[250] Carl A. Petri. Kommunikation mit automaten. *Ph. D. Thesis. University of Bonn.*, 1962.

[251] S. Pingali, D. Towsley, and J. F. Kurose. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. *SIGMETRICS'94, pp. 221-230.*, 1994.

[252] J. Postel and J. K. Reynolds. RFC 959: File transfer protocol, October 1985.

[253] P. RADOSLAVOV, D. ESTRIN, and R. GOVINDAN. Exploiting the bandwidth-memory tradeoff in multicast state aggregation, 1999.

[254] V. Ramasubramanian, R. Peterson, and E. G. Sirer. Corona: A high performance publish-subscribe system for the world wide web. *In Proceedings of Networked System Design and Implementation (NSDI), San Jose, California, May 2006.*, 2006.

[255] Jeff Ramsdale. Jini distributed events specification. *http://www.jini.org/wiki/Jini_Distributed_Events_Specification*, 2006.

[256] M. Reiter and K. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems, May 1994, 16(3), pp. 986-1009.*, 1994.

[257] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhauser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Computer Supported Cooperative Work*, pages 288–297, 1996.

[258] Dennis M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[259] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *Computer Communication Review, 27(2):24–36, April 1997.*, 1997.

[260] L. Rizzo. Pgmcc: A tcp-friendly single-rate multicast congestion control scheme. *In Proceedings of SIGCOMM'2000, Stockholm, August 2000.*, 2000.

[261] L. Rizzo and L. Vicisano. Rmdp: an fec-based reliable multicast protocol for wireless environments. *Mobile Computing and Communications Review 2 (2) (1998) 23–31.*, 1998.

[262] L. Rodrigues, K. Guo, P. Verissimo, and K. Birman. A dynamic light-weight group service. *Journal of Parallel and Distributed Computing 60: 12 (Dec 2000), 1449-1479.*, 2000.

[263] B. M. Roehner. Patterns of speculation: A study in observational econophysics. *Cambridge University Press (ISBN 0521802636). May 2002.*, 2002.

[264] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Programming Research Group, Oxford University*, 1986.

[265] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems, 2002.

[266] D. Schmidt, D. Box, and T. Suda. Adaptive: A dynamically assembled protocol transformation, integration, and evaluation environment. *Concurrency: Practice and Experience, vol. 5, pp. 269-286, Jun 1993*, 1993.

[267] F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computng Surveys. 22, 4 (Dec. 1990), pp. 299-319.*, 1990.

[268] M. W. Shields. Concurrent machines. *Computer Journal, (1985) 28 pp. 449-465*, 1985.

[269] Dale Skeen. A quorum-based commit protocol. Technical report, Ithaca, NY, USA, 1982.

[270] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *Concurrency control and reliability in distributed systems*, pages 295–317, 1987.

[271] Skype Limited. Skype. *http://www.skype.com*, 2008.

[272] D.A. Smith, A. Kay, A. Raab, and D.P. Reed. Croquet - a collaboration system

architecture. *Creating, Connecting and Collaborating Through Computing, 2003. C5 2003. Proceedings. First Conference on*, pages 2–9, 31 Jan. 2003.

[273] David A. Smith, Andreas Raab, David P. Reed, and Alan Kay. Croquet user manual v.0.01. *http://www.opencroquet.org*, 2008.

[274] R. B. Smith and D. Ungar. Programming as an experience: The inspiration for self. *In Proc. of the 9th European Conference on Object-Oriented Programming. Lecture Notes In Computer Science, vol. 952. Springer-Verlag, London, 303-330.*, 1995.

[275] R. B. Smith, M. Wolczko, and D. Ungar. From kansas to oz: collaborative debugging when a shared world breaks. *Commun. ACM 40, 4 (Apr. 1997), 72-78.*, 1997.

[276] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, New York, NY, USA, 1986. ACM.

[277] A.Z. Spector, J. Butcher, D.S. Daniels, D.J. Duchamp, J.L. Eppinger, C.E. Fineman, A. Heddaya, and P.M. Schwarz. Support for distributed transactions in the tabs prototype. *Software Engineering, IEEE Transactions on*, SE-11(6):520–530, June 1985.

[278] A.Z. Spector, R.F. Pausch, and G. Bruell. Camelot: a flexible, distributed transaction processing system. *Compcon Spring '88. Thirty-Third IEEE Computer Society International Conference, Digest of Papers*, pages 432–437, 29 Feb-3 Mar 1988.

[279] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information flow based approach to message brokering. *In Proceedings of International Symposium on Software Reliability Engineering, Paderborn, Germany, Nov. 1998.*, 1998.

[280] Chengzheng Sun and David Chen. Consistency maintenance in real-time collaborative graphics editing systems. *Computer-Human Interaction*, 9(1):1–41, 2002.

[281] Chengzheng Sun and Clarence A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Computer Supported Cooperative Work*, pages 59–68, 1998.

[282] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, 1998.

[283] Sun Microsystems. Java message service specification. *http://java.sun.com/products/jms/docs.html*, 2002.

[284] Sun Microsystems, Inc. Jinitm discovery & join specification. *http://java.sun.com/products/jini/2.1/doc/specs/html/discovery-spec.html*, 2005.

[285] Sun Microsystems, Inc. Jinitm join utilities specification. *http://java.sun.com/products/jini/2.1/doc/specs/html/joinutil-spec.html*, 2005.

[286] Sun Microsystems, Inc. Jinitm lookup service specification. *http://java.sun.com/products/jini/2.1/doc/specs/html/lookup-spec.html*, 2005.

[287] Sun Microsystems, Inc. Jinitm transaction specification. *http://java.sun.com/products/jini/2.1/doc/specs/html/txn-spec.html*, 2005.

[288] D. Teodosiu. End-to-end fault containment in scalable shared-memory multiprocessors, 2000.

[289] Daniel Terdiman. Second life: Don't worry, we can scale. *http://www.news.com/Second-Life-Dont-worry,-we-can-scale/2100-1043_3-6080186.html?tag=nefd.lede*, 2006.

[290] D. Thaler and M. Handley. On the aggregatability of multicast forwarding state. *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 3:1654–1663 vol.3, 26-30 Mar 2000.

[291] Tibco Software. Tibco Rendezvous. *http://www.tibco.com/software/messaging/rendezvous/*, 1994.

[292] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky. Hierarchical clustering of message flows in a multicast data dissemination system. *PDCS 2005.*, 2005.

[293] David Ungar and Randall B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242, 1987.

[294] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. *Software Practice and Experience. 28(9), Aug. 1998, pp. 963-979.*, 1998.

[295] R. van Renesse, S. Maffeis, and K. Birman. Horus: A flexible group communications system. *Communications of the ACM. 39(4):76-83. Apr 1996.*, 1996.

[296] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in horus. In *Symposium on Principles of Distributed Computing*, pages 80–89, 1995.

[297] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, 1999.

[298] J. Venkataraman and P. Francis. Chunkyspread: Multi-tree unstructured peer-to-peer multicast, 2006.

[299] L. Vicisano, L. Rizzo, and J. Crowcroft. Tcp-like congestion control for layered multicast data transfer. *In INFOCOM (3), pages 996-1003, 1998.*, 1998.

[300] Y. Vigfusson, K. Ostrowski, K. Birman, and D. Dolev. Tiling a distributed system for efficient multicast. *Unpublished manuscript.*, 2007.

[301] V. Vishnumurthy and P. Francis. On heterogeneous overlay construction and random node selection in unstructured p2p networks. *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–12, April 2006.

[302] William W. Wadge and Edward A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[303] J. Waldo. The jini architecture for network-centric computing. *CACM 42, 7 (Jul. 1999), pp. 76-82.*, 1999.

[304] J. Waldo. The end of protocols, or middleware for the new millennium. *Middleware 2000 Keynote. Hudson River Valley, NY, USA, April 2000. http://java.sun.com/developer/technicalArticles/jini/protocols.html*, 2000.

[305] Marcel Waldvogel and Roberto Rinaldi. Efficient topology-aware overlay network. *SIGCOMM Comput. Commun. Rev.*, 33(1):101–106, 2003.

[306] A. C. Weaver. Xpress transport protocol version 4. *In Proceedings of the IEEE International Workshop on Factory Communication Systems, pp. 165-174. Leysin, Switzerland, October 1995.*, 1995.

[307] Y. Weinsberg, D. Dolev, T. Anker, and P. Wyckoff. Hydra: A novel framework for making high-performance computing offload capable. *In Proceedings of the 31st IEEE Conference on Local Computer Networks (LCN 2006). Tampa, November 2006.*, 2006.

[308] B. Whetten, S. Kaplan, and T. Montgomery. A high performance totally ordered multicast protocol. *In Selected Papers From the international Workshop on theory and Practice in Distributed Systems (September 05 - 09, 1994). Lecture Notes In Computer Science, vol. 938. Springer-Verlag, London, 33-57.*, 1994.

[309] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[310] J. Widmer and M. Handley. Tcp-friendly multicast congestion control (tfmcc): Protocol specification, internet experimental rfc 4654, august 2006. 2006.

[311] J. Widmer and Handley. M. Extending equation-based congestion control to multicast applications. *In Proceedings of ACM SIGCOMM (San Diego, CA), Aug. 2001.*, 2001.

[312] Matthias Wiesmann and Andre Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):551–566, 2005.

[313] Eric W. M. Wong, Andy K. M. Chan, and Tak-Shing Peter Yum. Analysis of rerouting in circuit-switched networks. *IEEE/ACM Transactions on Networking*, 8(3):419–427, 2000.

[314] P. S. Woods. *Macromedia Flash(TM) 5 Developer's Guide*. McGraw-Hill Professional, 2001.

[315] WorldForge. Worldforge. *http://www.worldforge.org/*, 2008.

[316] Tim Wu. Network neutrality, broadband discrimination. *Journal of Telecommunications and High Technology Law, Vol. 2, p. 141, 2003*, 2003.

[317] W. Wu, Y. Xu, and J. Lu. Srm-tfrc: A tcp-friendly multicast congestion control scheme based on srm. *2001 International Conference on Computer Networks and Mobile Computing (ICCNMC'01).*, 2001.

[318] Qiwen Xu and Mohalik Swarup. Compositional reasoning using the assumption-commitment paradigm. *Lecture Notes in Computer Science*, 1536:565–583, 1998.

[319] Z. Xu, M. Mahalingam, and M. Karlsson. Turning heterogeneity into an advantage in overlay routing. *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, 2:1499–1509 vol.2, 30 March-3 April 2003.

[320] R. Yavatkar, J. Griffioen, and Sudan. M. A reliable dissemination protocol for interactive collaborative applications. *In MULTIMEDIA '95: Proceedings of the third ACM international conference on Multimedia (New York, NY, USA, 1995), ACM Press, pp. 333–344.*, 1995.

[321] J. Yoon, A. Bestavros, and I. Matta. Adaptive reliable multicast. *Technical Report No. BU-CS-1999-012, Sep. 1999.*, 1999.

[322] YouTube, LLC. YouTube. *http://www.youtube.com*, 2008.

[323] Xin Yan Zhang, Qian Zhang, Zhensheng Zhang, Gang Song, and Wenwu Zhu. A construction of locality-aware overlay network: moverlay and its performance. *Selected Areas in Communications, IEEE Journal on*, 22(1):18–28, Jan. 2004.

[324] Z. Zhang, S. Chen, Y. Ling, and R. Chow. Capacity-aware multicast algorithms on heterogeneous overlay networks. *Parallel and Distributed Systems, IEEE Transactions on*, 17(2):135–147, Feb. 2006.

[325] Michael Zink, Philip C Starner, and Bill Foote. *Programming HD DVD and Blu-Ray Disc*. 2007.

GLOSSARY

behavioral type
Type of an object expressed in terms of patterns of events it exchanges with the software environment in which it executes. 53

controlled element
A software component embedded within and directly interfacing an application that controls the delivery of messages, but does not implement any peer-to-peer or management logic. 129

distributed type
See "protocol type". 92

endpoint
See "live object endpoint". 53

filter
A software component that restricts forwarding of messages between scopes. 111

forwarding policy
A set of rules that govern the way messages are disseminated within a certain management scope. 111

functional compositionality
The ability to build larger software components from different parts that play different functional roles. 21

live distributed object
An instance of a distributed multiparty protocol executed by some set of software components distributed across the network, communicating with each other through network messages, and with the software environment through bidirectional event channels. 3, 15, 20, 51

live object
See "live distributed object". 3, 15, 51

live object endpoint
A bidirectional event channel through which a live object proxy communicates with its software environment. 53

live object proxy
A software component participating in the execution of an instance of a distributed protocol; a part of a live object. 52, 53

live object reference
A set of instructions for constructing a live object proxy. 55, 56

live object type
: A class of live objects that exhibit the same externally visible behavior, expressed in terms of the patterns of events that flow in and out of the live object's proxies through instances of its endpoints. 61

local controller
: A software component that implements peer-to-peer logic on behalf of a certain node, but does not directly interface applications, and does not own management decisions. 129

management scope
: A collection of machines that are jointly managed, and are governed by a common set of administrative policies. 111

multicast channel
: A mechanism for disseminating messages within a certain scope. 111

protocol agent
: A software component participating in the execution of an instance of a distributed protocol. 110, 111

protocol type
: A class of distributed multiparty protocols that exhibit the same behavior, expressed in terms of the patterns of interaction between the software components running an instance of the protocol and their software environment. 32

recovery domain
: An instance of a recovery protocol running within a certain management scope, and for a certain set of topics. 111

recovery protocol
: A subset of logic of a reliable multicast protocol that governs aspects such as loss recovery, atomicity, persistence, or view synchrony. 111

scope
: See "management scope". 111

scope manager
: A service that monitors, maintains information about, and makes administrative decisions regarding a certain management scope. 114

session
: A logical period of time in the history of execution of a single protocol instance, often associated with a specific membership or protocol configuration. 111

structural compositionality
: The ability to build larger software components from different parts that play the same role, but within different scopes, or in different parts or regions of a larger system. 21