Properties Framework and Typed Endpoints for Scalable Group Communication

Krzysztof Ostrowski†, Ken Birman†, and Danny Dolev§
†Cornell University and §The Hebrew University of Jerusalem

Abstract

Group communication is a powerful tool that simplifies the development of dependable systems, but widespread adoption of the paradigm has been limited. The main problem is that existing systems lack important forms of scalability and clean OS embeddings that can sustain high performance. QuickSilver[1] is a new platform designed to enable casual use of groups on a massive scale. Our approach relies on a new way of constructing hierarchical, scalable protocols. Groups are accessed via typed communication endpoints; an underlying properties framework promotes flexibility and modularity.

1. Motivation

Dependable systems must tolerate failures, typically by replicating data or services. However, replication brings its own problems: to maintain consistency, replicated data must be updated in a coordinated manner and service replicas need to synchronize potentially conflicting actions. Such tasks are greatly simplified by various sorts of reliable multicast, offered by *Group Communication Systems* (GCS). Prior work proved GCSs to be a useful tool in small deployments, but several factors have inhibited widespread adoption by distributed system developers.

First, prior GCSs have lacked scalability in at least one of several dimensions that, we will argue, are *all* required for success. Scalability research has focused primarily on scaling with the number of nodes; much less so on scaling with the number of groups. Yet if GCSs are used in a casual way as a fundamental programming paradigm, in a deployment with thousands of nodes there could be thousands of groups of varying sizes, heavily overlapping, and any given node might send or receive in many groups. Moreover, such systems will be under continuous stress from minor disruptions, such as scheduling delays, packet loss, resource contention or garbage collection. Existing GCSs handle such disruptions poorly.

We believe it is natural to consider creating a separate group for each service in a service-oriented architecture, each class of events in a publish-subscribe system, each product in a commercial platform, or each replicated data item. A security architecture might use groups to replicate keys, or to update nodes holding information about a particular security policy. In a trading system, a separate group could exist for each stock being traded and include all servers and clients that track it. In systems that reliably store documents or that enable collaborative editing, perhaps by multiple users at a time, a separate group could exist for each document, and thousands of such groups might span sets of nodes throughout a corporate network. In a large data center, processing orders or generating pages to display to consumers could involve hundreds of services responsible for different parts of the order, different data items to be accessed or updated, or different parts of the pages to generate. Separate groups might exist per user session: nodes that participate in a session could join its group for the duration of the session.

In the extreme, groups might be created as casually as files in operating systems. Groups could represent various sorts of distributed "live content" that can be accessed and modified by multiple clients, and clients might be able to browse through thousands of groups accessible to them just as they can browse through their file systems. Enabling this style of programming would dramatically simplify construction of dependable systems.

But the issue isn't just scalability and robustness to disruptive events. Another factor limiting the popularity of GCSs is that there has been little agreement on the most appropriate API. Existing systems are also poorly integrated with the existing standards, often tied to a specific platform, and not cleanly embedded into the OS; typically, they take the form of a library. When one considers the programming power that database systems have achieved by exploiting modern OS embeddings, we see a convincing case that GCSs could also benefit from flexible, generic ways of exposing their functionality to the applications, similar in flavor to web services, COM, and other service interoperability standards. These technologies have represented a true breakthrough in facilitating cross-platform and cross-language interoperation between distributed components. Below, we will argue that these same benefits can be leveraged by a GCS.

Finally, although a variety of GCSs have been implemented and some are highly configurable, most offer a limited set of features, and are extensible only to a limited degree. For example, extending an implementation of a virtually synchronous stack to use a different dissemination scheme, or to support a different way of collecting ACKs, typically requires changes to source code, because GCS components are tightly integrated. Yet such changes are often needed when fine-tuning a protocol to optimize bandwidth, latency, loss rate, churn, exploit IP multicast, tunnel through firewalls, etc. We need an approach for building much more modular and extensible GMSs.

[1] Our effort is supported by grants from AFRL/IFSE, AFOSR, and Intel. Contacts: {krzys,ken}@cs.cornell.edu, dolev@cs.huji.ac.il

2. Feasibility

The feasibility of our vision revolves around the question of scalability: can we design a system that supports thousands of groups overlapping irregularly? Work by our group over the past two years suggests that this problem may be solvable. We developed[2] and evaluated QuickSilver Scalable Multicast [1], a new GCS with simple and useful reliability guarantee based on periodic exchange of ACKs. QSM delivers throughput comparable to the network bandwidth, with 10-25ms latency, in configurations of 110 nodes and 8192 groups (we plan larger tests, though we see no sign of a looming scalability limit). QSM tolerates perturbations such as garbage collection, bursts of losses, churn, excessive send rates, node failures or periods of unresponsiveness triggering massive recovery. QSM is still a work in progress, and its peak performance is sensitive to group overlap. However, it already seems clear that casual use of groups is feasible.

QSM achieves scalability via two basic mechanisms, hierarchical protocol composition (extended and generalized in [2]) and clustering of nodes based on group membership to amortize overheads across multiple groups (Figure 1). The latter idea is based on the observation that in any set of nodes that are members of the same groups, all nodes receive the same data, participate in recovery of the same packets, and experience similar traffic patterns. This makes it possible to use one communication channel (a single IP multicast address, one buffer for batching messages on each sender, a common flow and rate control scheme etc.) to deliver messages in all groups to nodes in such set. It also makes it feasible to run a single protocol among all nodes in this set to perform local recovery for all messages simultaneously, irrespectively of the group or the source. Every control message exchanged by a pair of nodes may carry recovery information for all groups or senders. Furthermore, such information can be efficiently packed, so that the size of such control packets is totally independent of the number of groups, and grows almost linearly with the number of senders that are actively multicasting. In effect, we can amortize many costs across a large number of overlapping groups.

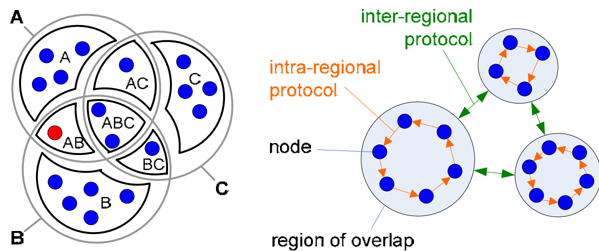

Figure 1. Left: Groups A, B, C form 7 regions of overlap. Right: Protocols in QSM run inside and across regions.

[2] QSM is open to the public and available for downloading from http://www.cs.cornell.edu/projects/quicksilver/QSM/index.htm

To this end, all QSM nodes are clustered into *regions*. Nodes **x** and **y** are in the same region iff **G**(**x**) = **G**(**y**), where for any **x**, **G**(**x**) is the list of names of all groups of which **x** is a member. To deal with potentially large regions, these are further subdivided into *partitions*.

Nodes in a partition run a local recovery protocol to repair packet losses locally between partition members, and to calculate properties that describe the entire partition, such as the list of packets delivered or cleaned up at all partition members etc. A single node in each partition acts as a *leader*. Partition leaders run, among themselves, a separate protocol to perform recovery across the different partitions in a region. Whenever a partition **x** learns that another partition **y** has packets that **x** is missing, inter-partition forwarding is performed. At the same time, partition leaders aggregate the partition properties mentioned earlier across the entire region. The aggregate values are collected by a *region leader*. Another, third layer of protocols runs between the region leaders and each sender, where the regional properties are used to request retransmission to the region, perform cleanup etc. Although brevity precludes a detailed discussion of the approach, we use hierarchical constructions of this sort ([2]) in support of the full range of mechanisms presented in the remainder of the present paper.

3. Our Approach

3.1. The "Backbone" Framework

Most reliable protocols, including QSM, rely on external or self-contained *Group membership Services* (GMS) to provide nodes with consistent membership change notifications. The GMS abstraction frees the protocol from having to handle issues such as network partitioning and failure detection. The GMS may be thought of as providing higher layers of a GCS with consensus-like semantics.

In QuickSilver, we generalize this idea and propose that the GMS also be responsible for establishing a hierarchy that serves as a basis for constructing scalable protocols. In QSM, the GMS calculates the regions of overlap and provides nodes with consistent membership notifications that are used to build the 3-level hierarchy of groups, regions and partitions, establish peer-to-peer recovery structures, elect leaders etc. In [2], we go one step further and offload many such tasks to the GMS, which notifies nodes what roles they play, which other nodes they should peer with, and what recovery protocols they should run.

The key objective is to separate the way reliability and other properties are ensured from the way GCS achieves scalability. To understand this, note that most protocols achieve scalability by building some sort of a hierarchy. In some cases, as in RMTP, this hierarchy is simple and set manually, while in other cases, as with p2p content delivery networks, it is built autonomously, and may be complex. Neither of these approaches is perfect. The

autonomous p2p approach is more robust and maintainable, but the way hierarchy arises is often arbitrary, paying little respect to administrative domain boundaries or the network topology, and this may be undesirable. Maintaining hierarchy also complicates protocols; makes it harder to prove their properties and to modify them. Accordingly, Quicksilver separates hierarchy construction from the aspects of the protocols that are concerned with ensuring reliability or other desired properties.

Our *backbone framework* is a network of *Scope Managers* (SM) replicated services, which maintain persistent and non-persistent state for, and act on behalf of, *scopes*. SMs communicate to build tree-like scope hierarchies for publish-subscribe *topics*. For every topic, they maintain a distributed view of its hierarchy, from *root scope* of that topic, through intermediate scopes, down to the level of *clients*, and notify each other about changes to this hierarchy in response to churn or failures. All updates are numbered, ordered, and consistent. The backbone framework is, in essence, a scalable generalization of a GMS. Scopes may be defined statically, e.g. to span administrative domain boundaries, or dynamically and autonomously. Regions and partitions in QSM are examples of the latter.

In [2] we explain how such per-topic scope hierarchies may be used to build scalable dissemination and recovery protocols (see also Section 3.2). However, this mechanism can have a more general use. The per-topic scope hierarchies can be thought of as an analogue of folder hierarchies. Each scope in our framework has a linearly evolving state, part or all of which may be persisted. Consistent hierarchy update notifications may be thought of as a versioning system for this virtual folder hierarchy. The resulting structure is useful for a variety of purposes, such as consistent failure detection, membership and naming services, possibly even storage or aggregation.

3.2. The "Properties" Framework

Quicksilver implements *a properties framework* that permits the construction of scalable protocols that are expressed as sets of rules. Here, the key observation is that if we separate dissemination from reliability, most reliable protocols can be expressed as state machines that operate in the following manner. A number of "properties" associated with individual nodes are defined to represent the state of the recovery process. Examples of such properties include the set of numbers of all packets delivered, cached, persisted, missing etc. The values of these properties are then used to trigger certain behaviors. These behaviors may include setting the values of other properties, e.g. a list of messages safe to cleanup, deliver etc., and simple actions, e.g. message forwarding or retransmission. Our experience suggests that behaviors of most protocols can be completely and accurately captured by sets of simple rules operating on such properties.

For example, a simple reliability scheme based on the exchange of acknowledgements might include properties such as **Cached**(**x**) or **Missing**(**x**), parameterized by node name **x**, and taking as their values the sets of identifiers of messages that are cached and missing at node **x**, respectively. These values are calculated by each node individually, and might be shared with other nodes. A rule of the form "**Cached**(**x**) $\wedge$ **Missing**(**x**) $\rightarrow$ **Forward**(**x**,**y**)" represents a requirement that whenever one node has a copy of a message missing elsewhere, the message should be forwarded. **Forward**(**x**,**y**) is a "derived" property. It represents the set of messages to be forwarded from **x** to **y**, and it has significance only to **x** and **y.** Upon learning of an update to this property, **x** and **y** may initiate push or pull forwarding. Note that we abstract from the way properties are calculated, where their values "materialize" and how they are propagated to the nodes for which they have significance. Such implementation concerns are irrelevant to the semantics or correctness of the algorithm. Instead, we implement such features in the "properties" framework as a generic mechanism, to be used by any protocol.

In the above example, values of properties can change: a message may be detected as missing after a timeout, but no longer considered missing after it arrives, forwarded or retransmitted. Similarly, nothing is cached indefinitely. For other properties, e.g. **Stable**(**x**) in virtual synchrony, we may require monotonicity. The properties framework provides means to achieve a monotonicity of properties.

The abstraction just described, together with an underlying mechanism for calculating and propagating properties, and a small set of generic features such as state transfer, is sufficient to implement a wide selection of reliability guarantees. For example, our implementation of virtual synchrony, complete with cleanup, flushing and with various optimizations, requires 19 properties, 2 of which are monotonic, and 11 rules to link them all together.

Finally, note that properties and rules such as those we mentioned above can be applied at any level: not only to nodes, but also to sets of nodes. For example, **Cached**(**x**) for a set of nodes **x** may represent the set of messages that are cached anywhere in **x**, while **Stable**(**x**) may represent the messages that have been, or are guaranteed to be, received by all nodes in **x**. Similarly, **Forward**(**x**,**y**) for sets of nodes **x**, **y** can be interpreted as messages, which **x** and **y** should arrange to be forwarded. Recall that QSM used a similar approach. Properties arose there in context of individual nodes, partitions, or regions.

Now, let us divide the set of all nodes in the system in a hierarchical manner, like we did in QSM, thus obtaining a hierarchy of subsets of the group, ordered by inclusion. Call these subsets *domains*. Assume that with each such domain, we can associate a set of properties, and that at each level where a higher-level domain decomposes into multiple lower-level domains (e.g. in QSM, a region decomposes into partitions, or a partition decomposes into a

set of nodes etc.), we can "install" a set of rules, similar to the forwarding rule above, and a mechanism that links the properties of lower-level domains to properties of higher-level domains, such as aggregating lower-level properties into higher-level properties (e.g. by calculating a sum of **Cached**(**x**) for all **x** in some **y** to obtain **Cached**(**y**), or intersecting **Stable**(**x**) for all **x** in this **y**, to get **Stable**(**y**)). It is not hard to see that the resulting structure produces a hierarchical protocol. Rules "installed" at the lower levels result in local actions, e.g. our forwarding rule "installed" within a partition in QSM implements local repair within the partition. Rules installed at higher levels correspond to global actions, such as an inter-partition recovery or a retransmission to the entire region in QSM.

The "properties" framework is a generic infrastructure for implementing protocols in the manner just described. It relies on the backbone framework to generate a tree-like hierarchy for each topic. With every element of such hierarchy, we associate an abstract entity called a *domain*. Domains mirror the hierarchy of scopes for the topic, and also form a tree. This tree of domains is used for recovery in a manner similar to QSM. For each domain, an "agent" is created on a physical node that maintains the "state" of this domain and act on behalf of it. In QSM, similar roles were played by partition and region leaders. Agents are connected to form distributed structures that aggregate and propagate values of properties, and that implement the rules that produce the desired protocol behavior.

Quicksilver's properties framework is designed to be extensible. We use a simple translation mechanism to represent the state machine and properties associated with a protocol in a tabular form executable by our agents, and then deploy the resulting information. Because the same domain may be reused for different topics, we can amortize overhead and achieve scalability in the same way we did it for regions in QSM. Thus while our initial focus has been on scalable versions of reliability models such as virtual synchrony, we should also be in a position to explore scalable security protocols or scalable protocols providing other dependability guarantees, such as replicated transactional database records.

3.3. Typed Communication Endpoints

At the outset of this paper, we emphasized the importance of achieving a clean OS embedding in order to exploit the powerful component integration frameworks that modern operating systems support as part of their "service architectures." For Quicksilver, we are developing *typed endpoints*, a generalization of web services. A typed endpoint is "owned" by an application, or a communication channel. It captures all constraints, expectations, requirements, and capabilities, of the entity it represents towards the entity it will be attached to. Pairs of endpoints can be connected provided that their contracts match. For example, an application endpoint may be connected to an endpoint presented by a group if that group provides the semantics required by the application, and the application implements the behaviors or functionality that the group expects of its members.

Our approach leverages existing support for strongly typed communication, available as a basic part of modern programming languages such as Java and C#, and which are now the focus of work on service oriented architectures and standards. Although the initial implementation was undertaken in the context of Microsoft's .NET environment and the associated web services platform, we plan to port Quicksilver to Linux in the future.

Quicksilver's endpoint specifications resemble WSDL definitions. They include a list of *named slots* corresponding to message exchanges, each listing any unidirectional *message flows* within its message exchange. Each flow contains the description of a message data type, serialization, encoding, transformations such as compression or encryption, annotations such as signatures, and *control features*. The control features could include specification of when a flow is considered to have succeeded (e.g. a requirement that it must reach all or quorum of destinations) or a list of acknowledgements expected to be returned for this flow. It can also include a list of all *functionality provided or required* by the endpoint, together with specification of interfaces used to access it, as implied by the semantics of the protocol to be used to implement the message flow, such as caching if the message is to be delivered reliably, delivery suppression if atomicity is to be guaranteed, persistence, commit or abort, and a number of other features related to reliability, security etc., including any user-defined extensions.

A special type of endpoint is a *factory*, equivalent to a constructor. Factories can instantiate endpoints of a fixed type. This paradigm is used for expressing such semantics as view change in virtual synchronous groups. In the latter case, both group and application endpoints are factories, and for every view, a new pair of internal endpoints is created. Factories can also express semantics related to reconnecting, synchronization, failure handling etc.

The *functionality provided or required*, described as a part of the "control" aspect of a message flow within the endpoint, is where the application "negotiates" the protocol semantics with the GCS, and the interfaces listed there are those that the GCS or the application uses to "talk" to the other party. The limited space precludes us from discussing the details of how this functionality and interfaces are described. Instead, we give a simple example to let the reader build some intuition. Consider a virtually synchronous group used for exchanging messages of type **Event**, and let us focus on the sender-side. Senders expect to be able to send messages of type **Event**, hence the sender's endpoint will contain a slot with a single data flow, initiated by the sender, annotated with **Event** as the data type carried by the message. The virtual synchrony semantics

requires that the sender be able to recreate messages, for the purpose of retransmission, and the sender may declare the *caching* capability among the control features for this flow, with the interface perhaps including calls **get**(**k**) and **purge**(**k**), to retrieve and cleanup messages, respectively. The group endpoint would look similar, except that flows would be reversed and any declared "capabilities" would be swapped with the "requirements". A receiver endpoint would also look similar, but with a single flow directed towards the receiver, and with control features including *caching* and *delivery suppression*, the latter with interface perhaps including a single call **safe**(**k**) to signal that the delivery is permitted for message with identifier **k**.

The above example is very simplistic, but it illustrates an important point. In contrast to interfaces offered by most GCSs, the endpoint contract does not need to hide aspects such as numbering of messages, buffering etc. from the application. Rather than providing a simplistic "send" interface with a black box implementation and a long list of options to set and control the protocol behavior, Quicksilver exposes (as much as possible) internal aspects of the protocol relevant to the application, to provide maximum flexibility for the developer. The user may still use "standard" components to convert a "low-level" endpoint to one with a simpler interface, using a mechanism similar to type casting.

There is a strong mapping from the definitions of endpoints to the "properties" used in the "properties" framework to express protocol semantics as a set of rules. The constraints placed by a group endpoint on the client endpoint includes any functionality, such as caching, delivery suppression, commit / abort pattern etc., that the application must provide, in order for the GCS to be able to implement all the "properties" that are required by the "properties" framework to run that group's protocol.

3.4. Operating System Embedding

Typed endpoints may be used directly by applications, but they may also be used in much the same way that applications use files. The mechanism is based on the one used by the OS to associate programs with file name extensions. It enables programming using a form of "point and click" browsing. A system service, acting as a scope manager, and configured with addresses of other scope managers, compiles the list of available topics in the background. The topics might be listed in a virtual folder ("My Topics") together with brief descriptive information. When the user selects a topic, the system compares the endpoint specification of that topic with the specifications of endpoints of registered clients, and presents the user with a dialogue, listing clients that could be used to "access" the topic. After the selection is made, the endpoints of the group and of the selected client are obtained and connected to each other. The clients mentioned above can be any applications hosting endpoint factory services. Addresses of such factory web services are added to a list maintained by the local scope manager.

Clients registered and used in this manner could include news readers, replicated filesystem clients, document collaboration tools, multi-player game clients, clients that can remotely execute scripts on cluster nodes, execute database transactions etc. Group communication now becomes a component integration technology, complete with a dynamic type system that maps down to strong properties in the underlying properties framework, and that scales well by exploiting hierarchical structure, aggregation and the other mechanisms employed in QSM.

4. Conclusions

We presented a novel perspective on the construction of scalable GCSs, based on the idea of separating scalability aspects from the protocol semantics. Our *backbone* and *properties* frameworks make it possible to implement a reliability property as complex as virtual synchrony using a short list of rules, which are translated by the system into a scalable, hierarchical protocol. The hierarchy can be further fine-tuned to match the specific setting or amortize overheads between groups, and the parts of the protocol running at different levels of the hierarchy, or in different parts of the network, may vary, both in terms of semantics, as well as the way it is realized. Such versatility and flexibility has not been possible with prior architectures. We believe that the techniques used in our work are potentially applicable in other GCS systems and perhaps event in transactional database replication or other sorts of parallel computing settings.

We proposed a new mechanism for exposing the functionality of GCSs to applications, leveraging OS support for runtime type checking and for web services. By explicitly formulating mutual contracts, our *typed endpoints* decouple applications from GCSs, and thus make the GCSs easier to use and more interoperable. The typed endpoints also make it possible to treat groups the way one treats files, in a "point and click" mode, as a "content" that can be easily browsed and "accessed". We believe that this represents a major step in making group communication a well-integrated, essential component of the OS, and in giving it a user-friendly look and feel.

Although Quicksilver is a work in progress, most components described here are relatively stable, and the QSM scalable multicast framework is available to the public. We expect that all aspects of the Quicksilver system will be running, at least in our lab, by the end of 2006.

5. References

[1] K. Ostrowski, K. Birman, and A. Phanishayee. QuickSilver Scalable Multicast. In submission, 2006.

[2] K. Ostrowski, K. Birman. Extensible Web Services Architecture for Notification in Large-Scale Systems. To appear in IEEE ICWS 2006.