THE ENSEMBLE SYSTEM

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

Mark Garland Hayden

January 1998

THE ENSEMBLE SYSTEM

Mark Garland Hayden, Ph.D.
Cornell University 1998

Ensemble is a group communication system that demonstrably achieves a wide range of goals. It is a general-purpose communication system intended for constructing reliable distributed applications; it is a flexible framework for carrying out research in group ware protocols; it is a large-scale, system-style implementation built in a state-of-the-art programming language; and it is also a mathematical object designed to be amenable to formal analysis and manipulation. Thus, Ensemble straddles a number of disciplines of computer science ranging from systems architectures to formal methods. The principal advances described in this thesis are the creation of the Ensemble system and the demonstration that it exhibits the properties just mentioned.

The thesis begins by presenting the Ensemble architecture, as well as background in group communication. We describe the various components of the architecture, give examples of their interactions, and compare this architecture with that of other layered communication systems.

The Ensemble protocols make heavy use of layered micro-protocols. We describe optimization techniques that greatly reduce the performance overheads introduced by layering and show how the architecture facilitates these optimizations. In addition we show how to formalize these optimizations in type theory and implement them using the Nuprl theorem prover.

Ensemble is implemented in a dialect of the ML programming language. We describe how the use of ML impacted the system, and present a wide range of comparisons between Ensemble and a similar system implemented in C.

BIOGRAPHICAL SKETCH

Mark Hayden is originally from Davis, California. He received his Bachelor of Arts in Computer Science from the University of California at Berkeley in 1991.

ACKNOWLEDGEMENTS

Ken Birman, Robert Constable (my advisors), Greg Morrisett, Úlfar Erlingsson, Robert Harper, Jason Hickey, Takako Hickey, Peter Lee, Robbert van Renesse, and Samuel Weber contributed valuable comments on earlier versions of this thesis. Contributors to the implementation of Ensemble include Tim Clark, Chris Driggett, Pedro Fheas, Roy Friedman, Takako Hickey, Ohad Rodeh, Robbert van Renesse, Alexey Vaysburd, Werner Vogels, and Zhen Xiao. Robbert van Renesse was the primary implementor of the Horus system, from which many ideas were adopted in Ensemble.

Chet Murthy played a valuable role in this work by suggesting a number of important early directions, as well as through his infectious enthusiasm.

Robbert van Renesse contributed to the optimization architecture in Chapter 3. Jason Hickey and Christoph Kreitz contributed to the formalization of the optimizations in Chapter 5. In particular, Christoph Kreitz implemented the protocol optimization framework in Nuprl. The design of the Ensemble message buffers in Chapter 4 arose out of discussions with Jason Hickey and Werner Vogels.

The work reported in this dissertation was supported in part by ARPA/ONR grant N0014-96-1-10014. Any opinions, findings, or recommendations presented in the following pages, however, are my own and do not necessarily reflect the views either of the Advanced Research Projects Agency or of the Office of Naval Research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The construction of fault-tolerant distributed applications involves overcoming a number of challenges. To begin with, it is difficult to get distributed processes to coordinate on tasks that require consistent actions to be taken by different components of the system. The protocols typically employed to achieve such goals are extremely complex and notoriously error prone. The additional needs to survive faults in different parts of the system, adapt to other changes in the environment, and also meet performance requirements considerably complicate the task. One successful approach to meeting these challenges, called *group communication*, involves structuring applications into cooperating groups of processes. This thesis addresses a number of issues in the design of the Ensemble group communication toolkit.

Ensemble was originally conceived as a bridge between several disciplines in computer science. It is a general-purpose communication system intended for constructing reliable distributed applications; it is an extremely flexible framework for carrying out research in group ware protocols; it is a large-scale, system-style implementation built in a state-of-the-art programming language; and it is also a mathematical object designed to be amenable to formal analysis and manipulation. These several views of Ensemble place it in a unique position straddling disciplines of computer science ranging from systems architectures to formal methods. Before moving on to the issues that form the primary focus of this thesis, we first give a picture of this larger context.

Group communication was first introduced as a framework for structuring critical, high availability systems. The *process group* abstraction is the central feature of group communication systems. It typically provides operations for processes to join and leave groups, and to communicate within a group. Group communication is a well-accepted approach to providing tools for building reliable distributed systems. Early group communication toolkits such as V [CZ85] demonstrated that process groups and group communication can be highly efficient and scalable. The ISIS [BvR94] system demonstrated the usefulness of group communication in settings such as stock exchanges, a major air traffic control system, VLSI fabrication process planning software, and other significant, critical applications. In applications such as these, group communication is used to coordinate distributed actions such as updates to replicated state. Today, group communication seems likely to make a transition into mainstream commercial environments through efforts such as Phoenix (IBM), NT clusters (Microsoft), the Surface Combatant 21 standard (the Navy), etc.

A major reason for the success of group communication is that groups are a good vehicle for introducing *properties* into distributed systems. There are a rich set of properties that have been found to be useful in group communication settings. Properties such as broadcast atomicity and consistent failure notification help application developers reason about the behavior of distributed systems, especially in complex failure scenarios. In addition, there are other properties regarding message ordering, state transfer, authentication and privacy, scalability, etc.

The richness of the properties used in group communication generates new challenges for systems designers. Communication systems such as ISIS were limiting because they provided a fixed (albeit large) set of properties for applications. The problem is that there are many classes of applications, each with its own communication structure and desired properties. Any fixed set of properties is nearly certain to be poorly matched with important classes of applications. Part of the reason is that it is often best to provide just the properties needed by an application but *no more* than are needed, because additional properties can introduce additional overheads. The ability to provide a large (and extensible) range of properties to applications requires additional flexibility from group communication systems.

Systems such as the x-Kernel [PHOA93] and Horus [vRBM96] were designed to add such flexibility through modular architectures whereby sets of *micro-protocols* (or layers) can be composed into high-level protocols. This allows applications to select the exact set of protocols that meet their needs. In some cases, the available set of protocols may still not fit the need, but the application designer also has the option to extend the system with en-

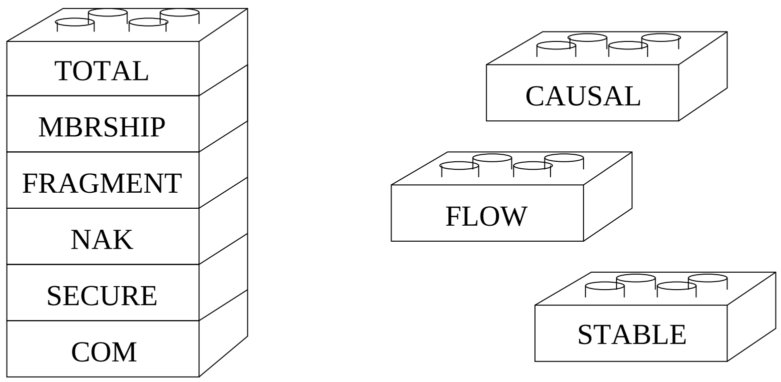

Figure 1.1: A set of a protocol layers, some of which have been composed into a protocol stack.

tirely new protocols. See Figure 1.1 for a depiction of such a system where layers can be viewed as Lego (TM) blocks. The idea is that a set of layers can be "snapped together" to instantiate a protocol stack with the desired properties.

Providing the ability to compose micro-protocols has its drawbacks, however. Communication protocols are notoriously difficult to get right. Even the core group membership protocols provided by ISIS (now more than 10 years old) are still not fully understood and developing specifications and verifying such protocols is currently an active area of research [CHTCB95, FLS97]. Introducing composable layers makes the system more complex by greatly increasing the number of configurations.

It should also be noted that in the background of this drive for flexibility, there is the even more powerful demand for high-performance. In many settings, developers are unwilling or unable to sacrifice performance for greater flexibility in the underlying communication system. Often, it is the case that these two objectives, flexibility and performance, push in opposite directions. This is because increasing flexibility improves the modularity of the system, and this in turn often prevents low-level optimizations from being made.

In summary, there are several conflicting demands placed on group communication systems. The systems must provide rich sets of combinations of properties from which applications make selections. They must provide high performance. And they must provide high confidence in their correctness, fault-tolerance, and security guarantees, possibly through some kind of formal analysis.

1.1 Ensemble overview

We developed a communication toolkit, called Ensemble, in an effort to address these problems. Our goal with Ensemble has been to build the system so that it can provide the levels of performance and flexibility needed in emerging distributed applications, but designed in such a way as to support the application of formal methods in order to reason about the correctness of the protocols. We use a dialect of the ML programming language in a way that both allows the system developers to leverage powerful language support to attain high performance and flexibility, and that also makes the system amenable to the application of formal methods that increase confidence in the correctness of the system. Other systems have aimed at similar objectives, but we believe Ensemble is the first to simultaneously pursue this mixture of objectives.

For example, Distributed ML [Kru93] (DML) is a group communication system built in ML. The work on DML and Horus was done in parallel at Cornell. DML can be viewed largely as an attempt to re-implement the ISIS toolkit in ML, making use of improved support for abstraction and threading provided by the Concurrent ML programming language [Rep91]. However, the DML implementation never reached maturity. It also lacked the facilities for protocol composition that Horus supported, did not support switching protocols on-the-fly (as Ensemble does), nor did it provide very good performance.

The x-Kernel is a modular communication system designed primarily to support point-to-point protocols. The architecture is based on gluing different protocol components together to construct protocol graphs. They demonstrated that modular implementation of TCP/IP could provide better performance that standard implementations. They explored optimization issues through techniques called Integrated Layer Processing (see Chapter 3). However, the x-Kernel did not support the ability to change protocols on-the-fly, not was it designed to support the use of formal methods for optimization or verification of the protocols.

As our work is done primarily using ML, one of the issues that we attempt to tackle is to understand what place there is for advanced programming languages in such systems. However, this is by no means the only focus of the thesis. Because many of the issues are independent of the programming language in use, we try to generalize beyond particular programming languages. We view our use of ML as a means to expose the fundamental issues through clean design.

1.2 Organization of thesis

The organization of the remainder of this thesis is as follows. Chapter 2 presents the Ensemble architecture, providing the reader with background to understand the later chapters. We describe the various components of the architecture, give examples of component interactions, and compare this architecture with that of a selection of other layered communication systems.

Chapter 3 addresses concerns about efficiency that arise from the heavy use of layering in Ensemble. Although layering provides a large number of advantages, including flexibility and modularity, it also has costs. We enumerate these disadvantages and then proceed to describe, in a programming language independent fashion, a series of optimizations that can be applied to eliminate the layering overheads. We have applied these optimizations to Ensemble, and this has resulted in extremely low communication latencies.

Chapter 4 describes language issues that arise in building a group communication system. The focus is on advanced programming languages (especially those in the ML family) that provide support for strong static type checking, clear formal semantics, automatic memory management, etc. While it is always important to design systems with a high level of abstraction, it is perhaps even more so in a language such as ML because the language is not designed with much support for low level operations. If the system cannot be designed with a high level of abstraction while maintaining efficiency, then the use of ML has the potential to become a liability rather than an advantage. However, we show that a communication system can be successfully implemented in ML and we present a wide range of comparisons between the resulting system, Ensemble, and a similar system implemented in C, Horus.

In a programming language such as ML with a formal semantics, the layering optimizations from Chapter 3 have an elegant formalization. Chapter 5 shows how the optimizations can be formalized in type theory and implemented in a theorem prover. This chapter brings together a number of themes. It shows the importance of the architecture in Chapter 2; it shows how to formalize the layering optimizations; and it also demonstrates that the use of ML provides additional benefits to those discussed in Chapter 4. This chapter ends with an example application of these techniques to an actual Ensemble protocol stack.

In summary, we present a novel group communication system, Ensemble, that simultaneously achieves several important goals. It provides a high level of flexibility through dynamic composition of protocols and the ability to change protocols on-the-fly. It provides high performance, as demonstrated through low latencies. It provides a clean architecture which leverages the advantages of advanced programming languages, facilitates the fine-grain decomposition of protocols, and applies formal methods to both optimize and verify the protocols.

Our software is becoming widely used. It has been freely available since June, 1996, from the Cornell Computer Science Department's web site. There are a number of early users. These include the Ensemble CD Jukebox (a distributed audio server developed with Jason Hickey), BBN AquA, Configured Energy Systems (a software provider for utility companies), and the NILE project [MOR+96].

Chapter 2

Ensemble Architecture

Before proceeding to the body of the thesis, we first present a description of aspects of the Ensemble architecture relevant to the later discussion. The purpose of this chapter is to familiarize the reader with a layering architecture in order to provide context for the remainder of the thesis.

Our design builds upon prior work concerned with introducing group communications systems into the OS (the V system) [CZ85], structuring point-to-point protocols for modularity [Rit84, PHOA93], and doing so for group communication systems systems with strong properties [vRBM96]. We were heavily influenced by the Horus work, and indeed Ensemble was "born" as an attempt to build a reference implementation of Horus. Performance was so good that our research group moved over entirely to use the Ensemble substrate.

This chapter proceeds as follows. Section 2.1 presents the important components of the Ensemble layering architecture. These include processes, the network, endpoints, groups, messages, events, layers, stacks of layers, and applications. Section 2.2 compares the Ensemble architecture with that of the Horus system and STREAMS. Section 2.3 gives examples of the interactions of the components.

2.1 Components

Here we present a brief description of each component in the system. Although we attempt to progress from basic components to the more complex ones, a certain amount of circularity occurs in the interactions that make forward references unavoidable.

2.1.1 The Network

The *network* serves as a medium for transmitting messages between processes. Not surprisingly, it provides two operations: send and receive. The send operation takes an address and a message and transmits the message to the corresponding destination. The receive operation returns any messages waiting to be delivered to a process. Perhaps more surprising is what the network does not provide. It does provide timing or reliability guarantees, nor does it provide failure detection of network links. Indeed, one of the important uses of Ensemble is to introduce guarantees over otherwise unreliable networks[1].

2.1.2 Processes

A *process* is the unit of state and computation provided by operating systems for executing programs. For instance, processes in standard operating systems such as Unix contain (among other things) an *address space* and one or more *threads of control*. The notion of a process used in this description is also assumed to include one or more network addresses which can be used to send messages between processes.

2.1.3 Endpoints

An *endpoint* corresponds to the state and computation associated with a unique endpoint identifier. Endpoints are an abstraction used to structure a process' communication. Endpoints can be viewed as a finer grained division of processes. They are useful for logically structuring communication in a process because a process may contain any number of endpoints.

Endpoint identifiers do not contain addressing information. Because of this, endpoints are not confined to a particular process and may migrate to other processes, although at any one time they should be "located" in only one process. What it means for an endpoint to migrate is system-dependent.

[1] Ensemble also works in settings with reliable network links (such as the SP2 fast interconnect), where it can introduce stronger forms of reliability and other useful properties.

For the purposes of Ensemble, it just means that a different address should be used to send a message to the endpoint after it has migrated. The guarantees associated with communication to migrating endpoints are provided by the layers in the selected protocol stack.

2.1.4 Groups

As a group communication system, the concept of groups is central to Ensemble. Groups are an important structuring mechanism that has proven to be useful in a wide range of distributed applications. In the most abstract sense, groups correspond to some computational resources that are distributed across some endpoints, each of which coordinates with the others to provide a service. In fact, a service can be composed of a number of groups, each of which coordinates to provide some logical subset of the overall service. Clients of the service may require interacting with several of the groups. The endpoints in the groups may overlap or be distinct. Often, one or more endpoint in a process may participate in multiple groups. The point is that groups serve as a tool for structuring distributed computation and that an important part of designing a group-based applications involves the ways in which the groups are structured.

As an example, consider a replicated file system implemented using a group structure. Such a system could be constructed with servers and clients in a single group. When a client wishes to create or modify a file, it may broadcast a message describing the operation to the group (or subcast it to just the servers). Reading a file may only require sending a single point-to-point message to one of the servers followed by a reply from that server. Of course, a group structure is not enough because there are a number of other issues that would be important in building such a service, such as ensuring that updates are reliably delivered to all servers. Often, many of these issues are handled by a group communication system that provides additional support beyond just the group structure, thereby simplifying the construction of such services.

The notion of groups has a number of different senses that are important to keep clear. Even though we may think of a group as a distinct object in itself, groups are typically implemented so that the group only exists through the endpoints that are participating in it. For instance, in Ensemble, there is no "group" data structure. Groups only appear through identifiers that serve as a naming mechanism for endpoints to use when communicating. In fact, not only do groups not have any state or computational resources of their own, as with endpoints their identifiers do not contain any addressing information[2]. Messages are "broadcast to a group" by sending a message to the addresses of the processes that contain endpoints in a group. The message contains a further "sub-address" specifying the group as a context for delivering the message. Sending point-to-point messages is done in the same fashion, except that the sub-address specifies an endpoint as the unique destination.

Even though groups really only exist through identifiers, we commonly use the notion of a group to refer to a set of endpoints communicating using a particular group identifier. When viewing a group as a collection of actual endpoints, there are a number of operations on endpoints that affect the group. Endpoints can form singleton *partitions*[3]. Partitions can merge together into larger partitions, and they can split into smaller partitions (for instance, when network failures occur). Finally, endpoints can send messages to other endpoints in its partition. These messages can either be point-to-point messages to another endpoint, or broadcasts to all the endpoints in the partition. Some systems also support "subcasts," which are broadcasts to a subset of the endpoints in the partition. See Figure 2.1 for a time line diagram of a changing group.

Yet another use of the term group is to refer to just one of the partitions of a group, even though a group may concurrently have several such components.

2.1.5 Messages

Messages are the objects that are transmitted over the network. As might be expected for a communication system, messages are the most basic data structure of Ensemble. Most of the source code is involved with managing

[2]This is not entirely true. When broadcasting to the group with IP multicast, group identifiers are used to generate a hash value which is in turn used to select an IP multicast address. This use, however, could be eliminated.

[3]We use the term partition loosely. Mathematically, a partition is a division of a set into disjoint subsets. Similarly, groups are often partitioned into subsets of endpoints. We use the term partition to refer to one of these partitioned components of a group rather than the overall set of components that form a partition.

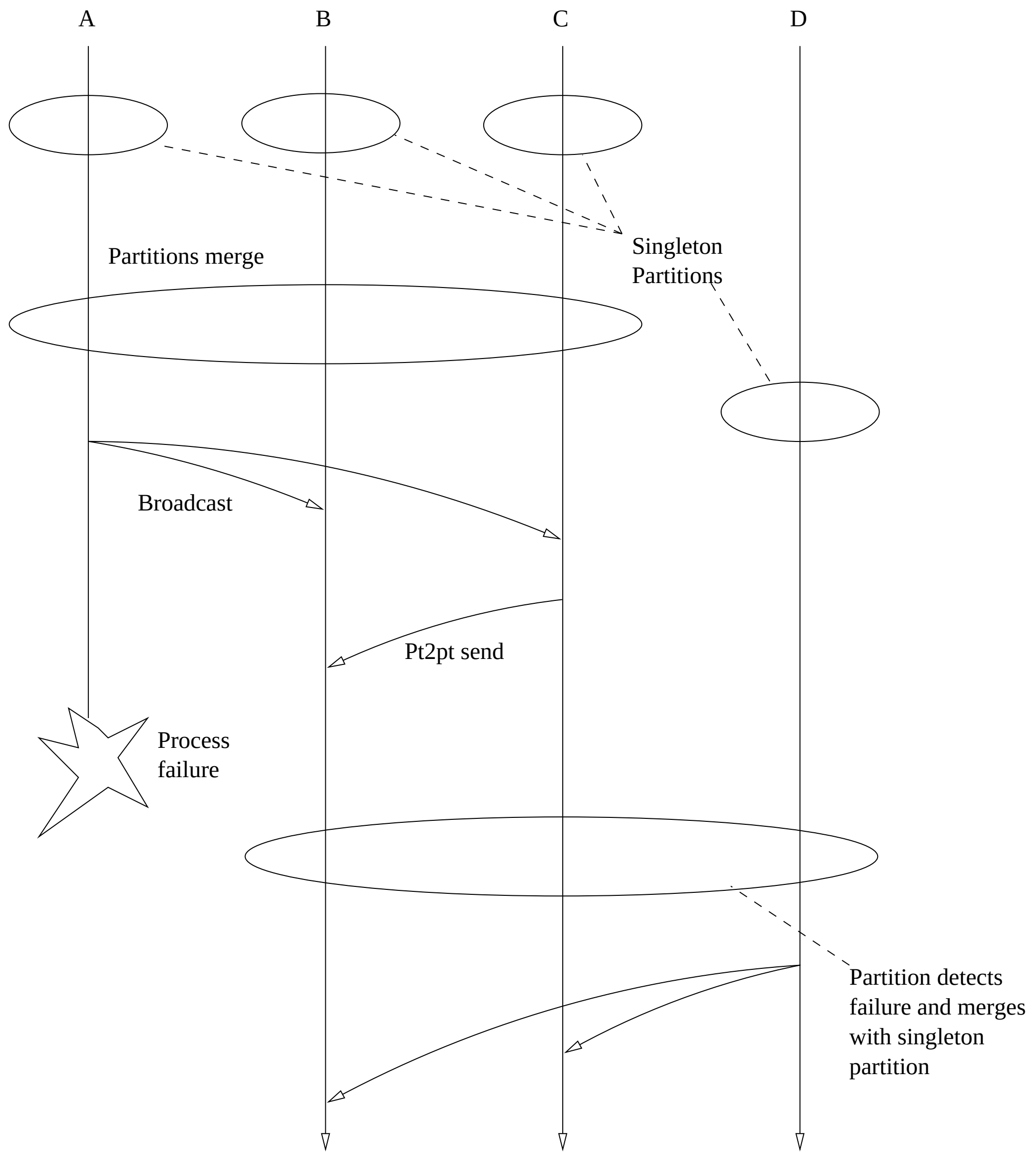

Figure 2.1: Time line of endpoints in a group. Each vertical line corresponds to an endpoint. Time proceeds down. Arcs are message transmissions. Each endpoint begins in a singleton partition. A, B, and C merge into a larger partition and communicate. Later, A fails and D creates a singleton partition. B, C, and D then merge to form new partition.

messages; most of the memory allocated is for messages; and most of the performance profile is involved in manipulating messages. Both architecturally and in terms of performance, communication systems are heavily influenced by their implementations of messages. The implementation of messages in Ensemble is described in detail in Section 4.4.

Ensemble messages are divided into two parts: the *payload* and *headers*. At the network and the application, messages take the form of flat sequences of bytes. This is because networks only transmit sequences of bytes, and because we wish to support application programs written in languages such as C, where messages are typically represented as sequences of bytes. Between the application and the network, protocol layers add information to messages in the form of headers. Headers are added at the sender by each layer and removed at the destination. For instance, headers are used to attach sequence numbers to messages in order to implement FIFO transmission protocols.

An important point to remember about headers is that access to a header is localized to the layer that generated it. As a message travels down the sender's protocol stack, each layer pushes one of its headers onto the message. At the receiver, each layer pops its header from the message in the opposite order (see Figure 2.2). Because all stacks in a partition use the same ordering of layers, each layer accesses headers generated by the same protocol (although a different instance of it). In addition, a layer cannot normally access the headers of other layers. Locality of headers is useful because it means that there are no interdependencies between different protocols on the structure of their headers, and so the header of one protocol can be changed without affecting other protocols.

Our approach to header formats takes a somewhat non-standard view of how to format messages. In contrast with many other systems [PHOA93, Pos81], the formats of headers for individual layers are not defined at the byte level. In addition, the headers of a stack of protocol layers is generally not the concatenation of the headers of the individual layers. As an example, consider TCP/IP where both the TCP and IP headers are defined at the byte level and both sets of headers are catenated together. In Ensemble, one string of bytes defines all the headers. This string of bytes could correspond to the catenation of the strings for each of the headers. However, Ensemble is free to format protocol headers any way so long as all endpoints in a group use the same format. This gives Ensemble a great amount of flexibility in how headers are represented and in the methods for optimizing them. But it also raises questions about the drawbacks of this approach. The main drawback is that there are no simple, static formats (such those used in TCP) to which programs must adhere when communicating with an Ensemble process. However, this would be difficult to achieve anyway because Ensemble embodies a more dynamic view of the world than most protocol architectures. Whereas the TCP protocol and its headers are not expected to change over the lifetime of a process, the Ensemble protocol stacks that an application uses do change dynamically. At any time, an application can request a protocol change that results in the process group switching to an entirely different set of protocols and headers. In addition, new protocols can be dynamically linked into Ensemble at run time. All of these forms of dynamicism argue against fixed header formats.

Many layered communication systems, such as Horus, view headers as extensions of the low-level representation of messages. In such systems, messages can be viewed as a "stack" of bytes and the application and protocols use operations to push and pop bytes onto and off of messages. This is a design feature of the x-Kernel, and Unix STREAMS has a similar mechanism. There are a variety of reasons for this. One reason may be the need to adhere to strict, standardized header formats, such as in implementations of TCP. Another reason is the expectation that low-level operations will be more easily optimized by compilers and help in achieving high performance. Unfortunately, such designs have costs. These include the programming costs for having protocols directly handle byte-ordering and word-size incompatibilities between hosts. In addition, the use of low-level operations may prevent high-level optimizations from being made to headers.

2.1.6 Events

Events are another data structure used for communication. Whereas messages are used for inter-process communication, events are used for intra-process communication. In other words, messages are used to communicate information between endpoints, and events are used to communicate within an endpoint's protocol stack. Events are never transmitted on the network. For instance, the protocol layers in a protocol stack use events to communicate with each other, both about messages as well as for other operations that may not involve messages.

An event is a record with a number of optional fields. The only field that

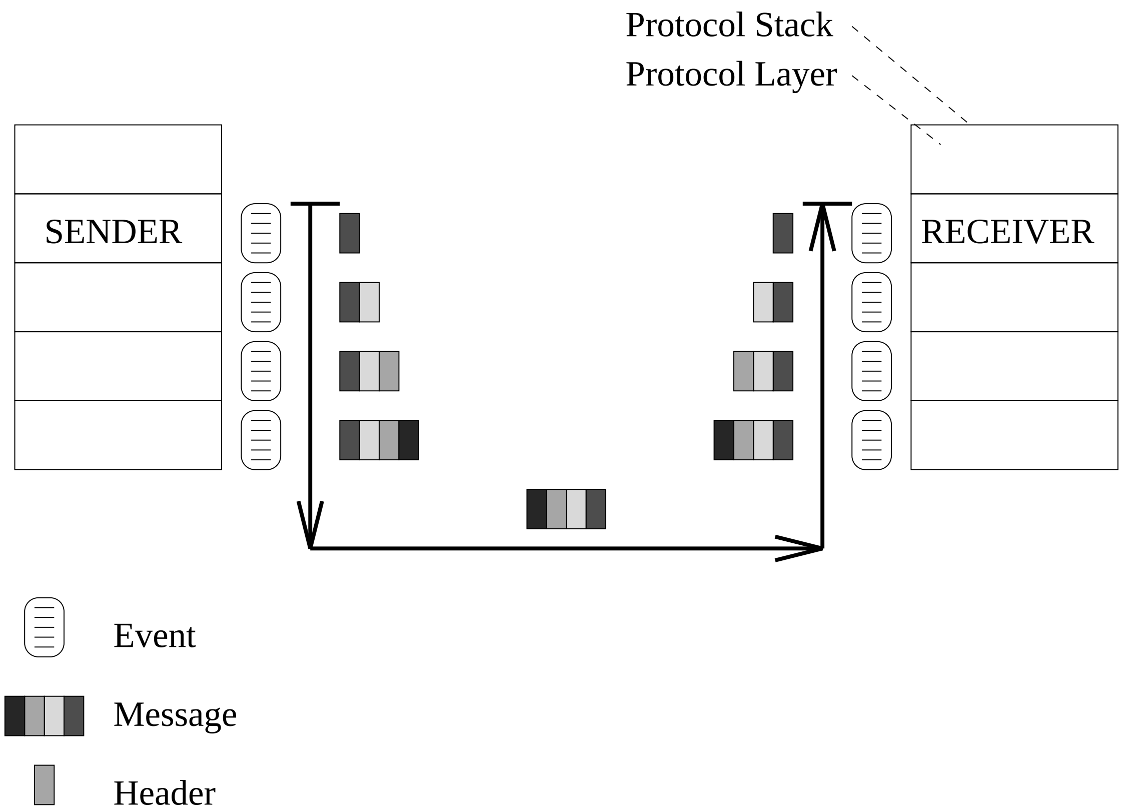

Figure 2.2: Diagram of use of headers and events. Message payloads are not depicted.

Table 2.1: Different types of Ensemble events.

event type	description
Account	output accounting information
Ack	acknowledge message
Block	block the group
BlockOk	acknowledge blocking of group
Cast	broadcast message
Dump	dump your state (for debugging)
Elect	I am now the coordinator
Exit	disable this stack
Fail	fail some endpoints
GossipExt	gossip message
Init	first event delivered
Invalid	erroneous event type
Leave	an endpoint wants to leave
LostMessage	a messages has been "lost"
MergeDenied	deny a merge request
MergeFailed	merge request failed
MergeGranted	grant a merge request
MergeRequest	request a merge
Migrate	change my location
Orphan	message was orphaned
Present	describe endpoints present in this view
Prompt	prompt for a new view
Protocol	request a new protocol
Rekey	request a rekeying
Send	point-to-point message
Suspect	endpoint is suspected to be faulty
Timer	request a timer
View	notify that a new view is ready
XferDone	notify that a state transfer is complete

Table 2.2: Some of the Ensemble event fields.

field name	description
Type	type of the event
Peer	rank of sender/destination
Ack	acknowledgement information
Address	new address for an endpoint
Failures	failed members
Presence	endpoints present in the current view
Suspects	suspected endpoints
SuspectReason	reasons for suspicions
Stability	stability vector
NumCasts	number of Casts seen
Mergers	list of merging endpoints
Contact	contact for a merge
HealGos	gossip for *Heal* layer
SwitchGos	gossip for *Switch* layer
ExchangeGos	gossip for *Exchange* layer
ViewState	state of next view
Protocol	protocol
Time	the current time
Alarm	alarm timeout
NoTotal	message is not totally ordered
ServerOnly	deliver only at servers
ClientOnly	deliver only at clients
History	debugging history

Table 2.3: Fields of a **ViewState** record.

field name	description
version	version of Ensemble
group	name of the group
protocol	protocol stack in use
params	protocol parameters
coordinator	initial coordinator
logical_time	logical time of this view
view	members in the view
address	addresses of members
out_of_date	who is out of date
clients	who are the clients in the group?
primary	is this the primary partition?
xfer_view	is this a state transfer view?
key	security keys in use
prev_ids	identifiers for previous views
uptime	time this group started

all events must contain is the "event type" field[4]. Some of the event types are listed in Table 2.1. Other fields that may appear in events include lists of failed members (in failure announcement events), the origin or destination of a message (in send events), a time value (in timer events), etc. Some of the event fields are listed in Table 2.2.

2.1.7 View state

Another important Ensemble data structure are the records used for configuring protocol stacks. These are called **ViewState** records. They contain all the information needed to initialize member endpoints of a partition, including information such as the name of the group, the membership list, the addresses of the processes for each endpoint, a description of the protocol to use, optional parameters, etc. Some of the fields are listed in Table 2.3.

[4]The use of type refers to a "tag" that takes one a set of enumerated constant values and is not related to the formal notion of types.

ViewStates are replicated in the sense that all endpoints in a partition are typically configured using equivalent **ViewState** records. Protocols establish new configurations of the system by disseminating new **ViewState** records. However, once created a **ViewState** record never changes (i.e., they are immutable), so new configurations are described by creating modified copies of previous **ViewState**s and there are no issues with reliably "updating" **ViewState**s.

2.1.8 Layers

High-level protocols in Ensemble are implemented by composing large numbers of protocol *layers* (often more than 15). Note that stacks (and the layers in them) are generated for each group that an endpoint joins, and that new stacks are created each time a partition reconfigures. Through heavy use of layering, we keep protocols (the most complex parts of a communication system) individually small. In addition, through careful design, layers can be combined in a large number of ways, thus providing a wide range of properties from which applications can select. Some of the properties that applications can select from Ensemble appear in Table 2.4 with brief descriptions. The layers for the default Ensemble protocol stack appear in Figure 2.3, also with brief descriptions. Details of these properties and protocols are beyond the scope of this thesis and are not presented here. The remainder of this section discusses the structure of individual layers; the next section discusses issues related to composing protocols into stacks.

Every Ensemble layer has three parts:

- A data type for its local state and a function for generating an initial state based on a **ViewState**.
- A data type for the headers it places on messages. Headers were described in Section 2.1.5.
- Handlers for communicating with layers above and below it in its protocol stack. This is the main body of the protocol.

Each instance of a layer maintains some *local state*. Different protocols use different types for their state records. For instance, a reliable transmission protocol usually maintains some kind of message buffer in its state. Each layer's state is maintained privately so that no other part of the system can access or modify it.

Table 2.4: Some of the properties currently supported by Ensemble. These draw upon a large background of research on group communication.

property	*description*
Agree	agreed (safe) delivery
Auth	authentication
Causal	causally ordered broadcast
Cltsvr	client-server management
Debug	adds debugging layers
Evs	extended virtual synchrony
Flow	flow control
Frag	fragmentation-reassembly
Gmp	group membership properties
Heal	partition healing
Migrate	process migration
Privacy	encryption of application data
Rekey	support for rekeying the group
Scale	scalability
Suspect	failure detection
Switch	protocol switching
Sync	group view synchronization
Total	totally ordered broadcast
Xfer	state transfer

protocol	*description*
Top	top-most protocol layer
Heal	partition healing
Switch	protocol arbitration and switching
Migrate	process migration
Leave	reliable leave
Inter	multi-partition view changes
Intra	single partition view changes
Elect	leader election
Merge	reliable merge protocol
Slander	failure suspicion sharing
Sync	view change synchronization
Suspect	failure detector
Stable	broadcast stability detection
Appl	application representative
Frag	fragmentation/reassembly
Pt2ptw	point-to-point window flow control
Mflow	multicast flow control
Pt2pt	reliable, FIFO point-to-point
Mnak	multicast NAK protocol
Bottom	bottom-most protocol layer

Figure 2.3: A sample protocol stack. This is the protocol stack created when using the default Ensemble properties {Gmp, Sync, Heal, Migrate, Switch, Frag, Suspect, Flow} (see Table 2.4). Ensemble provides a facility for translating from abstract properties to concrete protocol stacks.

Layers interact with their environment only through event communication with the layers above and below them in the protocol stack. For this they export handlers for receiving events and messages from the adjacent layers and in turn are given handlers for passing events and messages out to these layers. Layers have no direct access to the system clock or to system timers and must request alarms through event communication (we present below an example of a layer requesting a timeout).

Constraints on the interactions of layers are important throughout their design, implementation, optimization, and verification. The guarantee that layers cannot access other layers' states or headers means there are no dependencies between layers on these data types, so one layer's state or header can be changed without affecting other layers. Forcing layers to interact with their environment only through event communication guarantees that the behavior of a stack of layers is completely described through this event communication and the updates to the individual layer's states.

For example, consider the Horus system's layering model, where no constraints are placed on layers. They can (and do) make system calls, request timer callbacks, and spawn threads. Verifying such layers (or otherwise reasoning about them) requires the ability to model all the possible interactions of a layer with its environment, which greatly complicates such a task. In Ensemble, layers do not take such actions and therefore modeling their behavior is substantially simplified.

Of course, requiring layers to communicate only through events may have its disadvantages. For instance, the use of threads or shared state conceivably could simplify the construction of some layers. However, although this was initially a concern, we have found that all the protocol layers fit cleanly into this model and that instead of increasing the complexity of the layers, the constraints make them easier to understand because of the simplified model of execution. Requesting timer alarms is the only operation that is awkward to represent through event communication. We discuss the issues related to timer alarms below in the example interactions.

2.1.9 Stacks

Ensemble protocol stacks are linear compositions of layers which work together to implement high-level protocols. Because all protocol layers implement the same interface, all combinations of layers are "syntactically" valid. However, not all combinations of layers form useful protocol stacks. Ensemble provides a mechanism for selecting protocol stacks that implement a specified set of properties[5].

Layers are composed into protocol stacks following a *layering model*. The are two primary invariants required of the model. The first is that events are passed between layers in FIFO order. The second is each layer only execute a single event at a time (i.e., the execution of events is serialized at the layers).

There are number of advantages for defining this model. One is that it aid in reasoning about optimizations made to protocol stacks. Another advantage is that it allows there to be multiple mechanism for composing layers. For instance, Ensemble currently supports three kinds of composition: an event queue implementation, a threaded implementation, and a functional implementation. Abstractly, one can think of bi-directional event queues connecting each pair of layers, but the implementation can use any suitable mechanism.

Events emerging from the bottom-most layer in a stack cause a message to be transmitted on the network. The only events that emerge from the top-most layer are:

- A **NewView** event is generated when the protocol stacks determines it is ready to start a new view of the group. The event contains a **ViewState** record that is used to generate a new protocol stack (the creation of protocol stacks is further described below).

- An **Exit** event is generated when the protocol stack has nothing left to do. This usually occurs after it has determined that all endpoint have started the next view.

A new protocol stack is generated whenever the configuration of the group changes. For instance, when an endpoint fails or when two partitions merge together, a new stack is created for that new configuration at each endpoint. Using multiple protocols stacks is useful for two reasons. First, it simplifies many of the layers because they only need to implement their protocol for a single configuration. Second, it allows Ensemble to support on-the-fly protocol switching, where the new configuration actually uses a different protocol stack or set of parameters from the previous configuration.

[5]This is implemented through an ad hoc algorithm, though [Kar97] shows techniques for formalizing such an algorithm.

2.1.10 Application

The last component is the application. Supporting applications is of course the point of the architecture. Whereas normally the application would be considered to be everything in the process outside of Ensemble, we typically use the term application somewhat loosely to describe the handlers for one endpoint in one group. Thus, multiple "applications" may inhabit one process. As far as Ensemble is concerned, an application consists of a set of upcall handlers to use for delivering messages. In such a case, the different applications in a process can of course communicate locally through shared state.

Although most layering architectures place the application at the top of the stack, in Ensemble the application is considered part of one of the layers. An advantage of this approach is that the application can appear low in a protocol stack, thus eliminating the overhead of the layers above it when it sends messages. For instance, the group membership protocol layers are typically placed above the *Appl* layer (see Figure 2.3). The *Appl* layer serves as a proxy for the application in the protocol stack, inserting new messages into the stack for the application and redirecting application messages out of the stack to the application. The application gets several kinds of callbacks from the *Appl* layer for receiving messages and other information, and can generate a number of "actions" for introducing new messages in the system or for other purposes.

2.2 Comparison with Horus and STREAMS

Ensemble departs in a number of ways from previous layering architectures. We compare it here with the Horus [vRBM96] and STREAMS [Rit84] architectures.

- The application appears as part of one of the layers instead of at the top of the stack. This allows it to appear lower in the protocol stack, which improves performance. Horus and STREAMS both place the application at the top of the stack.

- Restricted layer interactions. Ensemble layers only interact with their environment though event communication. This aids both in the formal reasoning about and in the manipulation of layers. It is important to the optimizations in Chapter 3. Horus and STREAMS do not restrict layers, thus making such optimizations very difficult, if not impossible.

- New protocol stacks are generated when the configuration of the system changes. This allows Ensemble protocol layers to be switched on the fly. Horus and STREAMS can not support this.

- Decoupling of endpoints from processes. Ensemble endpoints are not directly connected with processes and thus can migrate between processes. Previous systems incorporate addressing information into endpoint identifiers, thereby trapping endpoints within single processes or requiring considerable "gymnastics" to extricate them.

2.3 Examples of component interactions

2.3.1 Network communication

Communication over the network occurs when an event, some message headers, and a message payload emerge from the bottom of a protocol stack. This causes the infrastructure to transmit a message on the network. The message is constructed by concatenating a connection identifier, the headers, and the message payload. A connection identifier exactly specifies the particular destination (one or more protocol stack within a process) of a message. At the destination, these are separated and the connection identifier is used to look up the protocol stack to which to deliver the headers and payload.

2.3.2 Timeouts

Protocol layers request timeouts through event communication. When a protocol layer needs a timeout to occur at some point in the future, it generates a **Timer** event with an **Alarm** field that specifies the time after which it wants to be woken, for instance, to trigger the retransmission of some message. This event is passed down the protocol stack until it emerges from the bottom. At this point, the infrastructure takes the timeout value and inserts it into a priority queue with other timeouts. When the timeout has expired, an **Timer** event is generated with the current time in a **Time** field. This event is passed up the protocol stack. Each layer that is waiting for a timeout checks the time, and if its timeout has expired it takes whatever action was scheduled to occur. The event is then passed on up the protocol stack.

As mentioned above, it may not be immediately clear that this is a good way to request timeouts. In particular, it is inefficient to pass an event through several layers instead of being able to directly request a timeout. However, there are several reasons why this does not turn out to be a problem and is in fact advantageous. First, all the Ensemble protocols use coarse-grained timeouts (on the order of 1 per second), so modest inefficiency in management of timers does not significantly affect system performance. Second, the optimizations we describe in Chapter 3 can eliminate all the overhead of the event communication. Third, the use of event communication provides the advantage that we can introduce clock synchronization protocols that can provide the abstraction of synchronized clocks to higher layers in the stack.

2.3.3 Sending and receiving a message

We now present a full example of an application sending and receiving a message. The application first allocates a payload buffer and puts the message information in the buffer. It then generates a **Send** action with the message payload and the name of the destination. This is passed to the *Appl* layer of the current protocol stack. The *Appl* layer then generates a **Send** event (distinct from the **Send** action) with the **Peer** field set to the same destination as in the action. This layer also generates a **Send**[6] header (again, distinct from the action and the event). The event, payload, and header are then passed down the protocol stack.

Most layers just add a trivial header to the headers and pass the information to the next layer. However, at the *Pt2pt* layer, which implements reliable, FIFO ordering of messages, more work is done. The *Pt2pt* layer buffers the payload and the headers from the above layers in case it needs to retransmit the message. It also puts a more complex header on the message,

[6]We present the headers as they are represented in the Ensemble system, where they are normal ML data objects. In the case of this **Send** header, the header is a constant header (it does not contain any other information). A non-constant header, **Data(seqno)**, is used below where the header also contains an integer field.

Data(seqno), where **seqno** is the sequence number of the current message. This header is pushed onto the previous headers and is passed along with the event and payload down to the next protocol layer. Eventually, these reach the bottom of the stack, are marshalled (see Section 4.4) into a sequence of bytes, and are transmitted on the network, as described above.

Assuming message arrives at the destination (i.e., the network does not drop it and the destination process has not failed), the headers, payload, and event are separated, the headers are unmarshalled, and all three are then delivered to the bottom-most layer of the destination protocol stack. Each layer pops off its header and usually then just passes the information to the next layer above it. At the *Pt2pt* layer, the **Data(seqno)** header is popped off and the sequence number is checked. If the sequence number is greater that what it has already received, it may buffer the information and send another message back to the origin with a **Nak(lo,hi)** header. This requests a retransmission of the earlier messages. However, we will assume the sequence number matches the next expected message, in which case the message is again passed up to the next layer. Eventually, the *Appl* layer will get the event, message payload, and the **Send** header (which is now the last header because the others have already been stripped off). The *Appl* layer passes the payload to the application's **receive** handler along with the origin of the message.

2.3.4 Stack creation

As mentioned above, a new protocol stack is generated whenever a configuration of the partition changes. This can be caused by the failure of a process, the merging of two partitions, the migration of an endpoint to another process, a request by the application to switch to using a different protocol stack, or any number of other causes. Regardless of the cause, the endpoints in the partition execute a reconfiguration protocol. When, the reconfiguration protocol is complete, each stack in the partition emits a **View** event from the top. This contains the **ViewState** record for the new partition. It is an invariant of the system that all endpoints in the same partition will use the same **ViewState** to construct their stacks, and so all endpoints are guaranteed to end up with compatible stacks. When the **NewView** event emerges, the infrastructure uses the **ViewState** to select the appropriate protocol layers, create a new local state for each layer, and to compose them. The stack is then connected to the network so that messages can be sent to it.

This is done by installing the connection identifiers that the stack accepts in a central hash table. When the external initialization is complete, an **Init** event is passed into the bottom of the stack to complete the initialization. This event passes up the protocol stack and each layer takes some action, such as requesting a first timeout.

The old protocol stacks may stay around for some time and operate in parallel with the new protocol stack. This is necessary, for instance, if some messages for the old stack need to be retransmitted. The two sets of protocol stacks are never able to communicate, however, because they use distinct connection identifiers on their messages. Sometime later, the old stack determines it is no longer needed and emits an **Exit** event from the top. This causes connections to the network to be disabled and eventually results in the state of all the layers being garbage collected.

2.4 Conclusion

The layering architecture is important to the rest of the work described here. The optimizations described in Chapter 3 and Chapter 5 depend on a number of properties of the architecture. The clean decomposition of protocol layers described in Chapter 4 is possible largely because of this architecture. In addition, it supports switching protocols on-the-fly and endpoint migration.

Chapter 3

Optimizing Layered Communication Protocols

This chapter addresses the question of how to achieve high performance in layered communication systems. Layers provide many advantages, but introduce serious performance inefficiencies. We describe where these inefficiencies arise, and then present optimization strategies that effectively eliminate them. This presentation relies heavily on the layering model presented in the previous chapter. In the remainder of this chapter, we point to ways in which features of the layering model enable the optimizations we describe. Most of the description is presented in a programming language neutral fashion because the optimizations are are not limited to systems implemented in ML. However, Chapter 5 presents an elegant formalization and implementation of the optimizations using a theorem prover that can be used when the system is written in ML.

The key idea of the approach we present is in the careful selection of the "basic unit of optimization." For optimization, our method automatically extracts a small number of common sequences of operations that occur in protocol stacks, which we denote *event traces*. We provide a facility for substituting optimized versions of these traces at runtime to improve performance. We show how these traces can be mechanically extracted from protocol stacks and that they are amenable to a variety of optimizations that dramatically improve performance. We recommend event traces be viewed as orthogonal to protocol layers. Protocol layers are the unit of development in a communication system: they implement functionality related to a single protocol. Event traces, on the other hand, are the unit of execution.

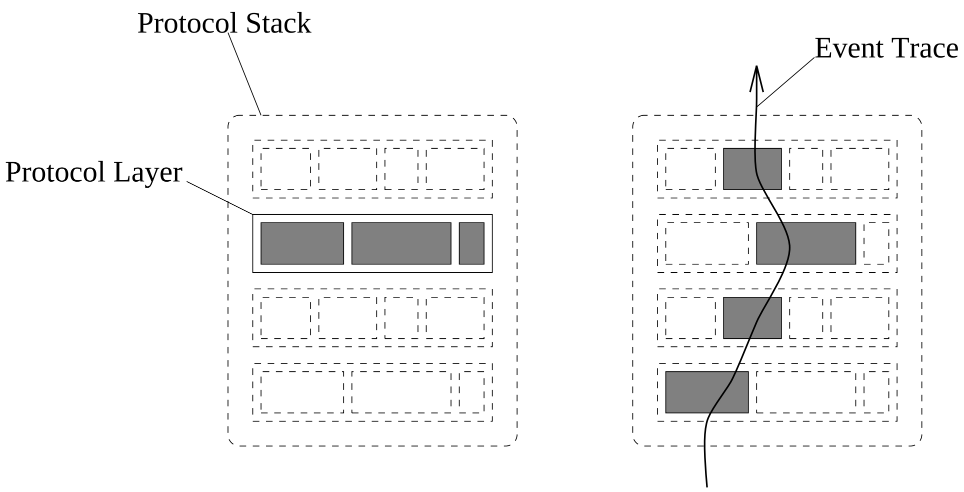

Figure 3.1: Comparison of protocol layers and event traces. Layers are the basic unit of functionality. Traces are the basic unit of execution.

Viewing the system in this fashion, we can understand why it is important to focus on protocol layers in development, but on event traces when optimizing execution (see Figure 3.1).

In addition to the high performance of the optimized protocols, our methodology benefits from its ease of use. The protocol optimizations are made after-the-fact to already-working protocols. This means that protocols are designed largely without optimization issues in mind. The optimizations require almost no additional programming. Only a minimal amount of annotation of the protocol layers is necessary (the annotation consists of marking the start and end of the common paths of the source code), and this annotation is only made to the small portions of the protocols which are in the common path. In addition, the optimizations place few limitations on the execution model of the protocol layers.

This chapter proceeds as follows. In Section 3.1, we present related work on optimizing layered communication protocols. In Section 3.2, we discuss the advantages and disadvantages of layering. In Section 3.3, we present the general approach to extracting common paths from protocol stacks. In Section 3.4, we present the optimizations for computation, for message headers, and for delaying operations. In Section 3.5, we present the performance of these optimizations.

3.1 Related work

The combination of advantages and disadvantages of layered protocol architectures has made layering optimization an active of area of research. Other approaches have been used to improve the performance of layered communication protocols. Work done in our research group on this problem has been described in [vR96]. In that work, protocols are optimized through the use of pre- and post-processing to move computation overhead out of the common path. Through this approach, the latency is greatly reduced, though the computation is not. Pre- and post-processing is done through a layering model where handlers are broken into the operations to be done during and after messaging operations (pre-processing for the next message is appended to the post-processing of the current message). This also demonstrated the use of small connection identifiers to compress headers from messages and the use of message packing to achieve higher throughput. A more detailed comparison is presented below in Section 3.5.1.

A somewhat orthogonal set of optimization techniques is called Integrated Layer Processing [PHOA93, CT90] (ILP). In general, the term ILP encompasses optimizations on multiple protocol layers, thus the optimizations we describe are a form of ILP. However, ILP techniques tend to focus on integrating data manipulations across protocol layers, whereas our optimizations focus on optimizing control operations and message header compression. ILP typically compiles iteration in checksums, presentation formatting, and encryption from multiple protocol layers into a single loop to minimize memory references. The Ensemble protocols almost never touch the application portion of messages. The only exceptions to this are security protocols which encrypt/decrypt the message or calculate cryptographic checksums. The optimizations we present focus on aspects of protocol execution that are compatible with these other approaches and we believe they could be combined in one system.

The Scout project [MP96, MPBO96] has explored optimization techniques revolving around *paths*, which are a similar construct to the traces described here. The two approaches differ in a number of ways, however. The Scout goals involve optimizing paths all the way from the network to other IO devices, such as disks and displays. In part because of the wide range of components involved, the Scout work has a loosely constrained execution model that prevents the formalization of arguments regarding the correctness of the optimizations, as can be done with our approach. In addition, apparently because of a lack of a carefully designed model, their optimizations only apply to limited kinds of traces and cannot optimize complex traces (as defined in Section 3.3.1). Merging of both approaches would have the potential benefit of extending our techniques to settings beyond communication protocols and at the same time increasing the ability to reason about the correctness of their optimizations and expanding the class of traces they can optimize.

[Bas97] (carried out subsequent to the work reported on here) adopted the layering architecture described in Chapter 2 and applied similar optimizations using a compiler, designed for optimizing layered communication protocols. The compiler was applied to modified versions of the Ensemble protocols and the Active Messages protocol suite [TVEB95], demonstrating significant performance improvements over the unoptimized protocols.

Other work from which we have adopted ideas includes work on optimizing TCP protocols. [CJRS89] demonstrated a number of techniques for optimizing TCP through determination of the important code paths in TCP, along with descriptions of methods to optimize those code paths. However, the approach presented there focussed on a single protocol architecture and did not address issues involved with generalizing the techniques to optimize a wide range of protocols.

PathIDs [Kay95] are another technique for improving the processing of messages by incorporating a field into message headers that causes normal case messages to be rapidly dispatched to hand-optimized code for handling those cases. This work does not address the issues involved in generating the optimized handlers. The use of PathIDs in an important ingredient in optimizing communication protocols because the time spent in dispatching messages becomes increasingly significant as the other overheads are reduced.

3.2 Advantages and disadvantages of layering

The optimizations described here are primarily targeted at removing the overhead introduced by using layered communication protocols. This begs the question of why we should be interested in layered communication protocols if their use causes serious performance degradation. Layered protocols have many advantages, some of which we list below:

- Layered protocols are modular and can often be combined in various ways, allowing the application designer to add or remove layers depending on the properties required. This way you only pay for what you use.

- When carefully decomposed into small layers, high-level protocols can be much more rapidly developed and tested than large, monolithic protocols.

- There are many cases where different protocols are interchangeable for one another. The variations may each have different behaviors under different workloads. In a layered system, application designers can select the suite of protocols most suited to their expected work load. In addition, Ensemble supports changing protocol stacks underneath executing applications, so the application can automatically adapt its protocol stack to a changing work load or environment.

- We are also interested in protocol verification. The current state of verification technology requires that the unit of verification be as small as possible. Small, layered protocols are just within the range of current verification technologies, whereas large, monolithic protocols are certainly outside this range.

The disadvantages of layered systems consist primarily of overhead, both in computation and in message headers, caused by the abstraction barriers between layers. Because a message often has to pass through many (10 or more) protocol layers, the overhead of these boundaries is often more than the actual computation being done. Different systems have reported overheads for crossing layers of up to $50\mu s$ [vRBM96] (on a Sparcstation 10). In Ensemble, this cost is as low as $5\mu s$ (also on a Sparcstation 10). The goal of this chapter is to mitigate these disadvantages so that the use of layers is a win-win situation.

3.3 Common Paths

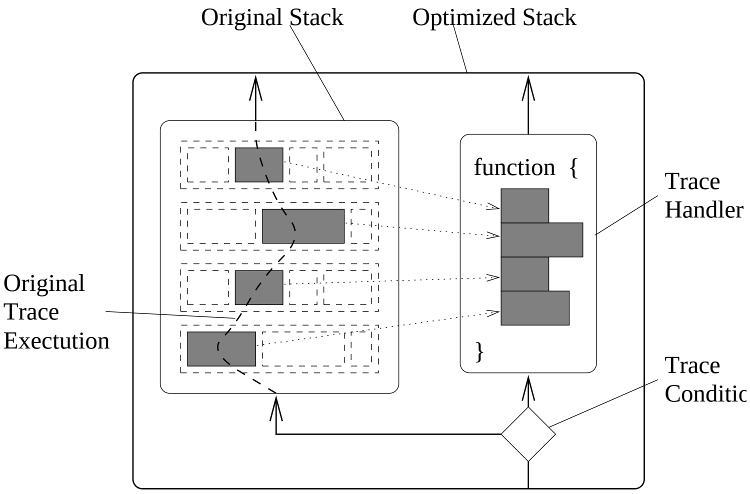

Figure 3.2: Event traces, trace handlers, and trace conditions. The original protocol stack is embedded in an optimized protocol stack where events that satisfy trace conditions are intercepted and execute through heavily optimized trace handlers. Pictured is the original execution of the event trace and the interception of that trace with a trace handler. In a full example, multiple traces are optimized with each trace having its own trace condition and handler. In addition there are traces starting also at the top of the protocol stack.

Our methodology focuses on the common execution paths of communication systems. The first step in optimizing the common path involves identifying it. The old adage, "90% of the time is spent in 10% of a program," suggests that most programs have common paths, even though it is often difficult to find the common path. However, carefully designed systems often do a good job in exposing this path. In layered communication systems, the designer is often able to identify the normal case for individual protocols and these cases can be composed together to arrive at global sequences of operations. It is these sequences, or *event traces*, that we focus on as the basic unit of execution and optimization. For each event trace, we identify a condition which must hold for the trace to be enabled, and a handler that executes all of the operations in the trace. For the purposes of optimization, we introduce three more components to the layering model (see Figure 3.2):

- *Event traces* are sequences of operations in a protocol stack. In particular, we use the term to refer to the traces that arise in the "normal case." An event trace begins with the introduction of a single event into a protocol stack. The trace continues through the various layers, where other events may be spawned either up or down. Often, a trace may be scheduled in various ways. It is assumed that one of these schedules is chosen for a particular trace.

- *Trace conditions* determine which event trace will be executed. This consists of a predicate on the local states of the layers in a protocol stack and on an event about to be introduced to the stack. If the predicate is true then the layers will execute the corresponding trace as a result of the event.

- *Trace handlers* is separate code that executes the sequence of operations in a particular event trace. If the trace condition holds for a trace handler, then executing the handler will be equivalent to executing the normal operations within the protocol layers.

For example, consider a type of event trace that occurs in many protocol stacks. When there are no abnormalities in the system, sending a message through a protocol stack often involves passing a *send* event straight through the protocol stack from one layer to the next. If messages are delivered reliably and in order by the network, then the actions at the receive side involve a *receive* event filtering directly up from the network, through the layers, to the application. Such an event trace is depicted in Figure 3.2. Both the send and receive event traces are called *linear traces* because (1) they involve only single events, and (2) they move in a single direction through the protocol stacks. The trace in Figure 3.2 enters the bottom of the stack and travels directly up through the protocol stack.

3.3.1 Complex event traces

Many protocol stacks have event traces which are not linear. *Non-linear traces* have multiple events that are passed in both directions through the protocol stack. Non-linear event traces are important because they occur in many protocol stacks. Examples of where they occur include token-based total ordering protocols, broadcast stability detection protocols, and hierarchical broadcast protocols. We will describe an example routing protocol in more detail below. Without support for such traces, such stacks could not be optimized.

A *hierarchical routing protocol* is one in which a broadcast to many destinations is implemented through a spanning tree of the destinations. The initiator sends the message to its neighbors in the tree, who then forward it to their children, and so on until it gets to the leaves of the tree which do not forward the message. Some of the traces in a hierarchical routing protocol would include the following, the first two of which are linear and the last of which is non-linear:

1. Sending a message is a linear trace down the protocol stack.

2. If a receiver is a leaf of the routing tree, then the receipt is a linear trace up through the stack.

3. If a receiver is not a leaf of the tree, the receipt will be a trace where: (1) the receive event is passed up to the routing protocol, (2) the receive event continues up to the application, and (3) another send event is passed down from the routing protocol to pass the message to the children at the next level of the tree (see Figure 3.3).

Determining event traces requires some annotation by protocol designers. They must identify the normal cases in the protocol layers, mark the conditions that must hold, and identify the protocol operations that are executed. From this, the traces can be generated by composing the common

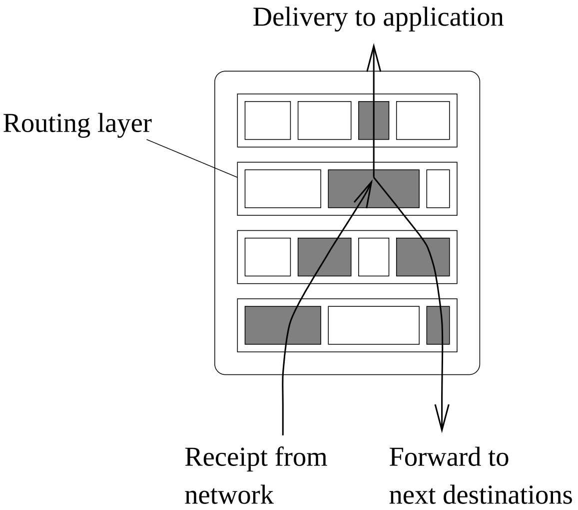

Figure 3.3: A complex, non-linear trace in a routing protocol stack. A message is receive from the network and passed to the routing layer. The routing layer forwards a copy down to the next destination and passes a copy to the network.

cases across multiple layers. Note that entire layers are not being annotated and no additional code is being written: the annotation is done only for the common cases, which are usually a small portion of a protocol layer.

Given the event traces of a protocol stack, we can build alternative versions of the code executed during those traces and modify the system so that before introducing an event into a protocol stack, the event conditions are checked to see if any are enabled. If not, the event is executed in the protocol stack in the normal fashion, and the checking has slowed the protocol down a little. If a trace condition holds, then the normal event execution is intercepted and we execute the trace handler instead. The performance improvement then depends on the percentage of events for which the trace condition is enabled, the overhead of checking the conditions, and how much faster the trace handler is.

The use of a trace handler assumes that there are no other events pending at the layers in the stack. If we assume the layering model (presented in Section 2.1.9) is implemented by pairs of event queues between layers, then the problem occurs if an event handler is executed when there are events in the intervening event queues. The event handler violates the layering model because the events in the trace would be executed out of order with regard to the previously enqueued event. Our solution to this problem relies on the flexibility of the layering model, and works by using a special event scheduler that executes all pending events to completion before injecting the next event (this is described in Chapter 5).

The transformation of the protocol stack maintains correctness of the protocols because trace handlers execute exactly the same operations as could occur in the normal operation of the protocol layers. If the original protocols were correct, then the trace protocols are too. This leverages off of the layering model in several ways. The first is that the layers are allowed to interact with their environment only through event communication. Thus we know that all of a layer's protocol is contained in its event communication and state updates. The second is that the layering model allows the operations executed by the layer to be executed in any fashion as long as it is equivalent to a legal scheduling of the events.

3.4 Optimizing Event Traces

The previous section described the general technique of extracting common paths as event traces. This section describes how these event traces are then optimized. We divide these optimizations into three classes: members of the first improve the speed of the computation; the second compress the size of message headers; and the third reorder operations to improve communication latency, without affecting the amount of computation.

3.4.1 Optimizing computation

The first optimizations are those that improve the performance of the computation in event handlers. The general approach here is to carry out a set of transformations to the protocol stack in order that traditional compilation techniques can be effectively applied.

First pass: extract the source code

The first step extracts the source code for the trace condition and trace handler from the protocol layers. We break the operations of a stack into two types: protocol and layering operations. Protocol operations are those that are directly related to implementing a protocol. These include operations such as message manipulations and state updates. Layering operations are those that result from the use of layered protocols. These include the costs of scheduling the event queues and the function call overhead from all the layers' event handlers. Layering operations are not strictly necessary because they are not part of the protocols. Given an event trace and annotated protocol layers, we use the annotations to textually extract from each layer the protocol operations for the trace.

Second pass: eliminate intermediate data structures

The second step removes the explicit use of events in the protocol layers. As described earlier, events are used to pass information between protocol layers. Each event must be allocated, initialized, and later released. Event traces encompass the life of the initial event and all spawned events so the contents of the events can be kept in local variables within the trace handler, which compilers are often able to place in registers.

As an example, consider a trace handler implemented in the C programming language (in ML this transformation is even simpler, but we present the case for C to demonstrate that that these optimizations are language-neutral). After extracting a trace handler, we end up with a function that allocates, initializes, and frees one or more events during each call to the function:

```
trace_handler(...) {
  event_t *ev ;
  ev = malloc(sizeof(event_t)) ;
  ev.type = Cast ;
  ...
  if (ev.type == Cast) {
    ...
  } else { ... }

  free(ev) ;
}
```

It is a moderately simple transformation to detect that the event allocated at the beginning of the function does not escape from the function and so can be allocated as a variable local to the function:

```
trace_handler(...) {
  event_t ev ;
  ev.type = Cast ;
  ...
  if (ev.type == Cast) {
    ...
  } else { ... }
}
```

A further optimization involves only allocating portions of the event that are actually used. For instance, suppose that only the **type** field is accessed in the **ev** variable. In such a case, we can just allocate a variable for the **type** field:

```
trace_handler(...) {
  event_type ev_type ;
  ev_type = Cast ;
  ...
  if (ev_type == Cast) {
    ...
  } else { ... }
}
```

Third pass: inline functions

The third step completely inlines all functions called from the trace handler. The payoff for inlining is quite large because the trace handlers form almost all of the execution profile of the system. Normally, code explosion is an important concern when inlining functions. However, code explosion is not an issue in this case because of several factors. There are only a small number of trace handlers which are each normally not too large: the inlining is focussed on a small part of the system so code growth will not be large. Also, the functions called from trace handlers are normally simple operations on abstract data types, such as adding or removing messages from buffers. These functions are not recursive and do not call many other nested functions, so fully inlining them typically adds only a fixed amount of code.

Fourth pass: traditional optimizations

The fourth step is to apply traditional optimizations to the trace handlers. These can be very effective because the previous passes create large basic blocks which compilers can optimize a great deal. In particular, constant folding and dead-code elimination can be effective due to the elimination of events. For instance, if one protocol layer marks an event record's field with some flag to cause an operation to happen at another layer, this constant value can be propagated through the trace handler so that the flag is never set at the first layer or checked at the second layer.

Continuing the example above, this optimization would propagate the value of the **ev_type** field to the conditional test, which in turn allows the **else** clause to be removed through dead code elimination.

3.4.2 Compressing protocol headers

The second class of optimization reduces the size of headers. The protocol layers in a stack push headers onto a message which are later popped off by the peer layer at the destination. We divide these headers into three classes, two of which we compress.

- *Addressing headers* are used for routing messages (they include addresses and other identifiers). They are treated opaquely: i.e., protocols are only interested in testing these headers for equality. These can be compressed through so-called path or connection identifiers, as described below.

- *Constant headers* include headers that are one of several enumerated constant values and specify the "type" of the message. For instance, a reliable transmission protocol may mark messages as being "data" or as "acknowledgments" with a constant header, and from this the receiver knows how to treat the message. These headers are compressed by our approach when they appear in the common path.

- *Non-constant headers* include all other headers, such as sequence numbers or headers used in negotiating reconfigurations. These are not compressed.

Protocol headers are compressed by using connection identifiers [vR96, Kay95, Jac90]. Connection identifiers are tuples containing addressing headers which do not change very often. These tuples are hashed into 32-bit values which are then used with hash tables to route messages to protocol stacks. MD5 (a cryptographic one-way hash function) is used to make hashing collisions very unlikely, and other techniques can be used to protect against collisions when they occur. The use of connection identifiers compresses many addressing headers into a single small value and all messages benefit from this compression. Although the main benefit of header compression is to improve bandwidth efficiency, small headers also contribute to improved performance in transmitting the messages on the underlying network and in the protocols themselves because less data is being moved around. We extend connection identifiers to contain an additional field called the "multiplexing index." This field is used to multiplex several virtual channels over a single channel. This use of connection identifiers allows constant headers to be compressed along with addressing headers. The constant headers in a trace handler are statically determined, a virtual channel is generated for that trace handler, and the constant headers are embedded in the code of the receiving trace handler.

Header compression significantly reduces the header overhead of the protocol layers. Even though each of the constant headers is quite small, the costs involved in pushing and popping them becomes significant in large protocol stacks. In addition, by encoding the constant values in the trace code, standard compiler optimizations such as constant folding and dead code elimination are possible. As an example, consider protocols in Ensemble. In many protocol stacks (including ones with more than 10 protocol layers), traces often only contain one non-constant field. Without trace optimizations, these headers add up to 50 bytes. With compression, the total header size decreases to 8 bytes. 4 bytes are a connection identifier. The other 4 bytes are a sequence number (for instance). There is not much room for improvement. This 8 byte header can be compared with those in similar communication protocols, such as TCP (40 bytes, 20 bytes for TCP with header compression)[1], ISIS [BvR94] (over 80 bytes), and Horus [vRBM96] (over 50 bytes).

Two related problems arise when additional header formats are introduced to protocol stacks that expect only a single format. The first problem occurs when a trace condition is not enabled for a message received with compressed headers (for example, out-of-order messages may not be supported by trace handlers). The message must be passed to the normal execution of the protocol but the message is not in the normal format. The second problem occurs when a trace handler inserts a message into a buffer and a protocol layer later accesses the message. The solution in both cases is to lazily reformat messages. Messages are reformatted by functions which regenerate constant fields and move non-constant fields to their normal location in the message. These reformatting functions can be generated automatically. For the first problem, the message is reformatted before being passed to the nor-

[1]Comparison with TCP and the TCP with header compression is more complicated than this. For instance, 20 bytes of the 40 byte TCP header is the IP header, which is overhead Ensemble also has when running over IP. In addition, TCP header compression is targeted toward serial links which allows more compression than is possible in the general case.

mal protocol stack. The protocol layers get the message as though it were delivered in the standard format.

In order to manage buffers containing messages in different formats, each message is marked as normal or compressed. Compressed messages are buffered along with their reformatting function. When a protocol accesses a compressed message, it calls the function to reformat the message. For most protocols, the normal case is for a message to be buffered and later released without further accesses. Lazy reformatting is efficient in these cases because messages are buffered in compressed form. Handling these buffers requires some modification of the protocol layers, but the modification is required only in layers with message buffers, and even then the modification is simple. The reformatting function needs to be stored with compressed messages, but this cost is offset by the decreased size of messages being buffered.

3.4.3 Delayed processing

The final class of optimizations improves the latency of the trace handlers without decreasing the amount of computation. The approach in [vR96] relies heavily on this class of optimization, whereas in our work this optimizations is made in addition to others that are more significant in our case. When a message is sent, there are some operations (such as determining a message's sequence number) which must be done before the message may be transmitted, and some which may be delayed until after the transmission (such as buffering the message). Similarly, at the receiver, some operations are delayed until after message delivery. The effect of reordering operations is to decrease the communication latency, but not the amount of computation done.

Fully automating delayed operations is a difficult problem, requiring some form of data flow analysis to determine which operations can be delayed while still retaining correctness. However, protocols can be annotated to specify which operations can and cannot be delayed.

3.5 Performance

An implementation of an automated protocol compiler for the Ensemble communication system is underway for the optimizations described above. Al-

Table 3.1: Performance comparisons for various protocol stacks.

version	code-latency	delayed ops	headers
normal	$1500\mu s$	none	50 bytes
trace/ML	$41\mu s$	$28 - 63\mu s$	8 bytes
trace/C	$26\mu s$	$37\mu s$	8 bytes

though currently we do not automatically generate trace protocols, we have constructed trace protocols by hand in a fashion that can be readily automated.

The protocols measured here implement FIFO virtual synchrony [vRBM96, BJ87] and consist of 10 or more protocol layers. In particular, the application has at least 10 layers below it. All the performance measurements are made on groups with 2 members, where the properties are roughly equivalent to those of TCP. Actual communication is over point-to-point (UDP or ATM) or multicast (IP Multicast) transports that provide best-effort delivery and a checksum facility. As we are interested in the overheads introduced by our protocols, our measurements are only of the code-latency of our protocols with the latencies of the underlying transports subtracted out. We focus on two measurements in this analysis. The first is the time between receiving a message from the network and sending another message on the network, where the application does minimal work. This is called the protocol code-latency. The second measurement is the time it takes to complete the delayed operations after one receive and one send operation. This is the amount of computation that is removed from the common path by delaying operations. All measurements were made on Sparcstation 20s with 4 byte messages. Measurements are given for three protocol stacks: the unoptimized protocols, the optimized protocols entirely in ML, and the optimized protocols where the trace conditions and handlers have been rewritten in C. See Table 3.1. The optimizations decrease the code latency from $1500\mu s$ to $26\mu s$, a $\times 57$ improvement. Including also the delayed operations, the improvement is $\times 23$. The C version of the protocol stacks has approximately $5\mu s$ of overhead in the code-latency from parts of the Ensemble infrastructure that are in ML. These could be further optimized by rewriting this infrastructure in C. There are no delayed operations in the unoptimized protocol stack.

The time line for one round-trip of the C protocol is depicted in Figure 3.4. Two Sparcstation 20s are communicating over an ATM network using U-net [BBVvE95] which has one-way latencies of $35\mu s$. At $0\mu s$, process A receives a message from process B off the network. $26\mu s$ later the application has received the message and the next message is sent on the network. At $61\mu s$, process B receives the message and sends the next message at time $87\mu s$. Process A completes its delayed updates by time $62\mu s$. The total round-trip time is $122\mu s$, of which Ensemble contributes $52\mu s$.

We carry out the experiments using groups of size 2 in order to be able draw comparisons with point-to-point protocol optimizations. Increasing the size of the group does not affect the normal case operation of sending and receiving messages in the protocols here because all of the operations take constant time in the size of the group. There are some background computations that grow in cost with the size of the group, but these do not occur in code paths measured above.

3.5.1 Comparison with Horus Protocol Accelerator

The Protocol Accelerator [vR96] achieves code-latencies of $50\mu s$ for protocol stacks of 5 layers (on a similar platform). The total time required for pre- and post-processing one message send and one message receive operation is approximately $170\mu s$, with a header overhead of 16 bytes. This can be compared to code-latencies of $26\mu s$ in Ensemble, protocol headers of 8 bytes, and total processing overhead for a receive followed by a send of $63\mu s$, with a protocol stack that has more than twice as many layers. In addition, there are other advantages of this approach over that of the Protocol Accelerator:

- This approach decreases actual computation and layering overhead in addition to latency. While latencies of both approaches are comparable, the computational overhead is significantly smaller through our compilation techniques.

- These optimizations can be applied to a larger class of protocols than that of the Protocol Accelerator, including routing and total ordering protocols.

- The Protocol Accelerator approach requires significant structural modifications to protocols. Our approach requires less annotation. We have demonstrated the use of our approach on a full-sized system.

- The primary requirement of our approach is the use of a strict layering model. The optimizations are not a necessary part of the layering model: protocol layers execute with or without the optimizations. This simplifies development because many optimization issues only need to be considered in the final stages of development.

3.6 Conclusion

This chapter began by introducing the question of how to achieve high performance in layered communication systems. After describing the causes of this performance inefficiency, the answer to this question was presented in several steps.

First, we argued that the layering architecture was crucial to our ability to perform optimizations that reduce or eliminate overheads. The layering architecture should provide an execution model in which the interactions of protocol layers with their environment is carefully controlled to enable one to reason about the correctness of cross-layer transformations.

Second, we presented a sequence of optimizations that focussed on the small numbers of code paths (or traces) that are commonly executed. The optimization steps extract these code paths and then proceed to eliminate the processing and header overheads in these traces.

Chapter 5 shows how these optimization strategies can be formalized in a theorem prover and describes work in progress in implementing the optimizations in this fashion.

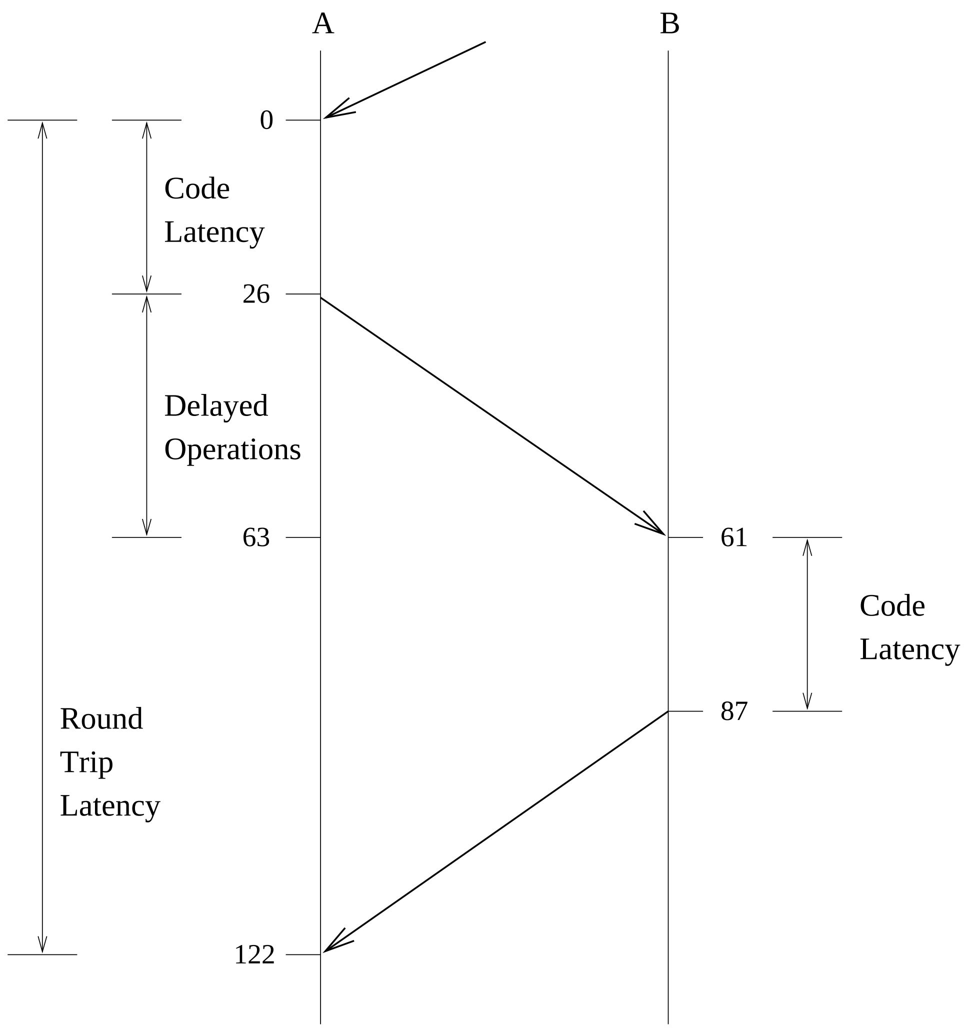

Figure 3.4: Round-trip latency time-line between two processes. Vertical scale is in μs.

Chapter 4

Impact of Using ML

This chapter addresses questions that revolve around the role of advanced programming languages such as ML in the design and implementation of communication systems. In particular, we are concerned with determining the impact that the use of ML had on Ensemble. This includes questions such as: where did ML aid in the design and implementation of Ensemble, how did it help or hinder in achieving high performance, does ML provide sufficient support for managing low-level details of the system, how did the implementation in ML compare with the previous system implementations in C, and how can ML be improved to better support system-style applications?

This chapter proceeds as follows. Section 4.1 related related work. Section 4.2 describes Objective Caml, the implementation of ML we use. Section 4.3 compares the implementations of Horus and Ensemble. Section 4.4 describes how messages are represented in Ensemble. Section 4.5 addresses issues with message buffers, which we needed to manage explicitly. Section 4.6 shows how inlining in ML can allow systems-style applications to make extensive use of abstraction barriers with very little cost. We then conclude with a summary of our lessons and a list of features which we feel are missing from ML.

4.1 Related work

Related work has been done in the Fox project [BHLM94] which demonstrated the use of ML for systems programming. They developed a complete TCP protocol stack in ML[1] that interfaces very closely with the network. However, Fox and Ensemble differ significantly. First, the Fox project implements TCP, which is a standard protocol, and so is constrained in many ways that Ensemble is not. For instance, TCP has fixed header formats that Fox TCP must adhere to, whereas Ensemble is free to set its own header formats and to change them as the system evolves (this issue is discussed at length in Section 4.4.1). In addition, the Fox design is deliberately very similar to TCP implementations in C because the developers wished to show that systems can be built in ML in a fashion similar to C. Ensemble, on the other hand, was not restricted in this fashion. One result of these differences was that Ensemble was able to achieve better performance than the implementation of Horus written in C, whereas the Fox TCP implementation is slower than implementations of TCP in C.

Other related work has been done with Erlang [AWWV96, Hau94]. Erlang, a product of Ericsson, is a functional language designed to support distributed telephone switching software. A number of impressive telecommunications products have been built using Erlang and they have found many of same advantages of using functional languages for distributed communication that we have. Erlang does not support static type checking, although there are several efforts underway to add this to the language. The approach with Erlang was to design a new language with support for distribution, whereas our approach has been to build libraries in an existing language.

4.2 Objective Caml

We use the Objective Caml (Ocaml) system [Ler97] which implements its own sub-dialect of the CAML [WL93] dialect of ML. Although this chapter is intended to be general, it is important to distinguish between the ML

[1] Fox uses Standard ML of NJ, whereas the implementation of ML we use is Objective Caml

family of programming languages and the particular implementation that we use. For instance, Standard ML is a language, Standard ML of NJ and Harlequin ML Works are implementations, and Ocaml is both a language and an implementation. The languages and implementations differ widely in number of ways. In this section, we briefly describe the Ocaml system, some features particular to Ocaml, and our experiences with it, both positive and negative.

4.2.1 Portability

The Ocaml system is actually two compilers. The first is a bytecode code compiler that provides rapid compilation, platform-independent bytecode, and good performance. The second is a native code compiler that gives slower compilation but generates higher-performance code. The compilers are interchangeable and run on a large number of platforms, including Windows NT, Windows 95, and most variants of UNIX. Porting Ensemble to new platforms has usually not involved modifying ML code, but revolved around issues outside of the control of the ML compiler (such as incompatibilities in the "make" program). The native code compiler provides very good performance. It gives efficient support for curried functions and support for inlining within and across module boundaries. We give a detailed example of the optimizations in Section 4.6. Other notable features of the system are a large library of UNIX system calls, support for automated marshalling of data structures, and features for object oriented programming[2].

4.2.2 Performance considerations

In designing Ensemble, we were careful to restrict the use of certain features that can hurt performance. As an example, consider higher-order functions. They have the problem that their use often requires allocation of closures (closures are dynamically generated function objects). Higher-order functions are used extensively in Ensemble, but only so that closures are not allocated in the normal case. Two techniques were used to achieve this. The first was a phased approach where closures are created when protocol stacks are initialized, but not during their normal execution. This technique is similar to one described in [BHLM94]. The second way was to use higher-order

[2]Ensemble does not use Ocaml's object oriented features

iterators for data structures such as list and array iterators. These are efficient because inlining of the iterator can eliminate closure allocation[3]. Thus the creation of closures only appears outside common execution paths or in ways that can be easily optimized.

4.2.3 Memory management

Ocaml supports garbage collected memory management. Although some ML implementations are perceived to require large amounts of memory [Mac93], the Ocaml system is known for its efficient use of memory and this has not been a problem for us. Ocaml uses a generational garbage collector with a stop-and-copy minor heap and an incremental mark-and-sweep major heap. Our experiences with the garbage collector have been positive, with two exceptions. The first is that there is no support for compacting the major heap, which means that long-running programs never release memory from the major heap. This causes problems with highly available server applications that run for weeks at a time (and longer)[4]. The other exception is that the major heap does not do a good job of managing large objects and tends to fragment over time. This problem was observed with the message buffers that Ensemble uses and eventually caused us to manage them through explicit reference counting (see Section 4.4.2 for more details). In summary, we found the Ocaml garbage collector extremely useful for almost all our data structures, but for some we had to take over and manage them ourselves.

4.2.4 Interoperability

Ocaml provides support for easily interfacing with C programs. ML programs can issue cross-language calls to C functions and vice-versa. In both cases, exceptions are handled correctly across any number of calls into and out of C. ML objects can incorporate pointers to C objects outside the heap and C code can declare references to ML objects in the heap. With this support, we implemented a C interface to the Ensemble system. This required writing a set of C stub routines for calling into Ocaml code. In addition, at

[3]The Ocaml compiler currently does not do this.

[4]The next version of Ocaml, which has been released since writing this, now supports compaction. This was also added in part because of the problems we reported running into.

our prodding, the Ocaml implementors added the capability to generate C libraries from ML programs. This allows Ensemble to be built as a normal C library and linked with C programs. The Ensemble library can then be treated as a black box from C code which does not need to know that ML was used.

Although it is not difficult to link ML and C programs in this manner, we have noticed that memory errors in C programs can easily corrupt the ML heap, which then usually causes the entire process to crash. C programs by chance may not access their own corrupted data structures, but the ML heap is regularly traversed by the garbage collector, so corruption of the heap is likely to result in a process failure. From the point of view of a C developer, even though the bugs in such cases are in the C code, the use of ML makes the entire program more "fragile" and makes tracking down problems in the C portion more difficult. This problem is compounded by the fact that most C debugging tools are unable to handle the ML heap. In the case of Ensemble, we avoid this problem by providing two versions of the C interface that appear identical to the application. The first, the "inboard" version, includes Ensemble and the ML runtime in the C process. This version provides the best performance, but exhibits fragile behavior when the C program is buggy. With the second, the "outboard" version, the C program and Ensemble execute in separate processes which communicate via UNIX pipes. This version is used while debugging applications because it isolates the ML heap from application errors.

4.2.5 Debugging and profiling

Until recently, Ocaml did not provide a debugger, which occasionally made debugging difficult. However, we found many of the hardest problems to debug in C, such as memory errors, are prevented by the ML type checker, and so the impact of a missing debugger is somewhat reduced. A continuing problem, however, is the difficulty of profiling memory behavior of ML programs (normal C tools can be used for standard execution profiling). While predicting the operations that cause memory allocation is usually easy in Ocaml programs, it is much more difficult to get a good picture of the overall memory allocation patterns in programs. Other systems, such as Harlequin's Standard ML environment [Har96], provide support for profiling memory usage, so this kind of support is certainly possible.

4.2.6 Summary

Our experience with Ocaml has been generally positive. It has provided a stable platform for us, and when there have been bugs in the system, the implementors have been quick to respond with fixes. Our success with ML is due in no small part to the excellent job of the Ocaml developers.

4.3 Comparing C and ML implementations

The similarities of Horus and Ensemble allows us to draw a variety of comparisons between the systems in order to better understand the impact of ML. We begin by discussing why it is reasonable to compare the two systems, and then present a variety of comparisons, from more to less concrete. Throughout these comparisons, keep in mind that for all their similarities, Ensemble and Horus are different in many ways. Ensemble supports almost all the functionality that Horus does and many things Horus does not, but the functionality of Ensemble is still neither a superset nor a subset of Horus. The design of Ensemble embodies many lessons we learned from Horus. Were we to rewrite Ensemble in C, we would probably arrive at a system closer to Ensemble than Horus, even with differences in the programming languages. Also note that both Horus and Ensemble are highly modular systems and each have many different configurations. In comparing them, we have attempted to match comparable configurations where possible.

4.3.1 Development times

Both Ensemble and Horus were developed primarily by single (though different) programmers and contributions from the research group were primarily made in the form of additional protocol layers or interfaces to support additional programming languages. Horus was actively developed for 2 years. Ensemble has been under development for 1.5 years.

4.3.2 Language interfaces

Both Ensemble and Horus are intended to be able to fit a variety of interfaces and to be used from many programming languages, so a consideration in our switching to use ML was the question of how accessible the system would be to programs written in C. As described above in Section 4.2, Ocaml provides adequate support for interfacing with C. In addition to C, C++, Tcl/Tk, and CORBA [Maf95], which are supported by Horus, Ensemble also supports Smalltalk, Ada, and (of course) ML.

4.3.3 Supported platforms

Both Ensemble and Horus seek to be largely platform-independent. Horus is supported on a variety of UNIX platforms and a smattering of other operating systems such as Chorus and Mach. Supporting new platforms for Horus requires writing some low-level system calls to access platform-specific thread and messaging operations. Ensemble runs on all platforms supported by Ocaml, including practically all UNIX platforms and Windows 95 and Windows NT. The use of ML has meant for us that porting issues are largely left to the ML compiler. For instance, there are some platforms (IBM AIX and Hewlett Packard HPUX) that were too bothersome to support for Horus that are supported by Ensemble because it is no trouble to do so.

The Ensemble software distribution includes a pre-compiled ML bytecode library that can be used on all supported platforms. A user downloading Ensemble on any platform merely compiles the demonstration programs (or their own programs) and links with the platform-independent bytecode libraries we provide. The platform-dependent, native-code libraries are only compiled if bytecode execution provides insufficient performance.

4.3.4 Multi-threading

Horus is a heavily threaded system. Every message received from the network causes a new thread to be created to handle it. In addition, every time a message is passed up in a protocol stack a new thread is also forked to handle that. However, optimizations are made so that in many cases the previous thread is recycled instead of forking an entirely new thread.

The issues related to threads in Ensemble are somewhat more subtle because its architecture is more flexible than Horus'. Although Ensemble is single threaded, there are a number of ways to introduce threads to the system. For instance, the C interface to Ensemble introduces C threads for the application, while Ensemble executes on in a single thread. In addition, each Ensemble protocol stack can optionally be configured to use a threaded implementation of the layering model (similar to Horus') instead of the default event-queue implementation.

However, one issue that has caused difficulty is that ML and C programs executing in a threaded environment are not as well supported as in single-threaded environments. In the absence of threads, C code can call into ML code and vice-versa. With threads, these inter-language interactions have to be more carefully managed because, in general, C code cannot call into ML code.

4.3.5 Sizes of executables

Ensemble and Horus executable binaries are approximately $880K$ and $400K$ bytes, respectively. These numbers are of course dependent on the platform, compiler, and level of optimization. We took the size of the stripped (i.e., without debugging symbol information) binaries for the default optimization level of the systems on Sparcstations running Solaris 2.5. For Ensemble, we give the size of the native code binaries. The bytecode binaries are $700K$ bytes. We believe the reason bytecode is not much smaller than native code is that Ocaml uses a 32-bit bytecodes.

4.3.6 Memory requirements

Ensemble and Horus have roughly similar memory requirements. At initialization, Ensemble processes with no protocol stacks use $91K$ bytes on the ML heap. In a similar configuration, Horus begins with $73K$ bytes on its heap. Each group joined by a process adds $7.8K$ bytes with Ensemble and $11K$ bytes with Horus. When the stacks become active (for a non-intensive application), the heap may grow to around $.5M$ bytes for Ensemble and $1.5M$ bytes for Horus. The resident set sizes for these processes (again for a non-intensive application) are around $1.5M$ bytes for Ensemble and $2.5M$ bytes for Horus. We believe the additional space used for Horus is caused by the use of preallocated thread stacks (Horus protocols make extensive use of threads, while Ensemble is single-threaded by default).

4.3.7 Performance

Both Ensemble and Horus have very good performance. When transmitting $1K$ messages on Sparcstation 20's, both Ensemble and Horus are easily able to use the available bandwidth of a $10M$ bit Ethernet. In order to compare the efficiency of the common code paths in both systems, a good measurement is the application to application latency. On Sparcstation 20's, Ensemble (compiled as native code) achieves a one-way latency of $595\mu s$ and in a similar configuration Horus has $700\mu s$. In this configuration, the overhead of the network is $355\mu s$, so the latency induced by Horus is $345\mu s$ and that of the Ensemble is $240\mu s$. Garbage collection has a minimal impact on performance because Ensemble allocates very little memory on the heap in the normal cases for sending and receiving messages, and none of the normal-case allocation gets promoted to the major heap.

With the optimizations described in Chapter 3, the overhead for Ensemble drops to $41\mu s$. These same optimizations could be applied to Horus to achieve a similar speedup. However, many of the architectural improvements made to Ensemble would have to be replicated in Horus first, which would probably necessitate extensive rewriting.

4.3.8 Line counts

Line counts are another useful characteristic for comparison even though comparisons of sizes of source code depend a large amount on factors such as the coding style of the programmers. See Table 4.1 for a number of different line count comparisons. All line counts are raw (comments have not been stripped) and include both implementation and interface files (i.e., .c and .h files, respectively, for C).

- *Total lines* is the total number of lines in each system, including demonstration programs. This gives a sense of the overall sizes, but otherwise contains little information for useful comparison. Ensemble has considerably more demonstration programs than Horus. The bulk of the C code listed for Ensemble is for C/C++ interfaces, associated testing code, and an interface to the Electra CORBA-based replicated object system [Maf95]. No C code is actually needed to run Ensemble because the Ocaml UNIX library provides stubs for all needed system calls. However, Ensemble comes with its own set of UNIX stubs that can optionally be used to improve performance. These stubs amount to about 1000 lines of C code.

- *Core lines* is an estimate of the size of the core components in a stripped down system. This is what is needed to get standard configurations of each system running. Only minimal sets of protocol layers and none

Table 4.1: Comparison of lines of code in Ensemble and Horus. See text for explanation.

measurement	Ensemble (ML)	Horus (C)
total lines	45873 (+ 11000 C)	140000
core lines	≈ 17000	≈ 35000
protocol lines	10692	79000
average lines per protocol	198	1519
platform-dependent lines	0	27020

of the external language interfaces are included. For Horus, we only include the machine dependent code for a standard UNIX system. This measurement gives a more focussed picture of the code sizes of both systems.

- *Protocol lines* is the total number of lines of protocol layers. Both systems have around 55 layers. This measurement is very important because the protocol layers are the most complex parts of each system, and smaller layers tend to be easier to develop, debug, comprehend, verify, and maintain. Protocol layers coded in ML are significantly smaller than those coded in C. This is discussed in more detail below.

- *Platform-dependent lines* is a count of lines of code used on a subset of the supported platforms. For Horus, this consists primarily of code for accessing system-dependent threads and messaging operations. Ensemble, which is unthreaded by default, has effectively no such code (there is a little to work around idiosyncrasies of Windows 95 and NT), whereas Horus has a large amount. This is significant because platform-dependent code often creates software maintenance problems.

4.3.9 Sizes of protocol layers

While the Horus and Ensemble infrastructures have diverged a good deal, they retain the same basic layered architecture. Many layers or collections of layers have direct analogues in both systems, thus allowing comparisons

Table 4.2: Size comparisons of comparable Horus and Ensemble protocols.

C-layer	ML-layer(s)	C-lines	ML-lines	factor
Frag	Frag	900	176	5.1
Problem	Suspect	1389	128	10.8
Stable	Stable	1639	318	5.2
Credit	Credit	2367	435	5.4
Mbrshp	Inter:Intra:Leave:Merge:Elect	5134	911	5.6

of their number of lines. In general, the "important" protocol layers in Ensemble are about a factor of 5 times smaller in lines than those in Horus. See Table 4.2 for a table of sizes of comparable layers (or sets of layers) from both systems. Some of the differences in size can be attributed to differences in the language used, but some also to an overhaul in Ensemble of the general layering structure in Horus. For instance, Horus layers interact directly with thread and synchronization operations, whereas in Ensemble the infrastructure handles synchronization for all layers. If Ensemble were to be recoded in C, the sizes of the resulting layers would be significantly smaller than the Horus layers.

This said, we believe that the decrease in code size is also due in part to the use of ML as a programming language. There are several ways in which the programming language has had an impact. The first is that almost all data structures in Ensemble are managed automatically by ML. The explicit management of memory in Horus requires a large amount of code. The second is the better set of facilities in ML for abstraction, which encourages structural changes that result in decreased code size. Examples of this include the use of polymorphic abstract data types and higher order functions. The third is the ability to manipulate messages with standard language facilities such as pattern matching. Most communication systems use a special set of operators for manipulating messages because the contents must be linearized (marshalled) before transmission. Instead of treating messages as sequences of bytes, Ensemble uses ML data structures for all headers and linearizes them with an automatic marshalling facility provided by Ocaml. Not surprisingly, raising the abstraction level for Ensemble's core data type, messages, leads to significant reductions in code size.

4.3.10 Bugs

The way we use ML prevents many kinds of bugs from occurring. For instance, when programmers add fields to the headers of protocol layers in Horus, they have to go through the protocols looking for all cases where a change needs to be made. In Ensemble, the headers are normal ML data structures, and this allows the compiler to detect and signal inconsistencies through its type checker. In addition, the ML marshaller handles converting the headers into byte sequences for transmission on the network, so incompatibilities in byte ordering and word size are transparent to protocol layers in Ensemble. A user recently compiled and ran Ensemble on a machine with a 64-bit word size for the first time we are aware and encountered no problems. These kinds of problems were a constant concern in Horus. Bugs in Ensemble are generally bugs in the protocol logic and not memory management or message formatting errors.

4.3.11 Evolution

Even though difficult to quantify, an important system characteristic is the ability to evolve. Both Horus and Ensemble are research systems that were intended in part to be toolkits to facilitate research in new protocol architectures. Our experience with Horus was that it did evolve a great deal for some time . However, it became increasingly difficult to make changes to the system. We believe this is because Horus became over-engineered generating a web of interdependencies. Often, these dependencies had to do with details of memory management or other issues that do not arise in ML. Ensemble has continued to evolve, often in dramatic ways. A detailed description of the evolution of one part is in Section 4.5, but there are many other similar examples.

4.4 Messages

As we pointed out in Chapter 2, messages are a central data structure of Ensemble and their implementation is an important practical concern. The approach we have taken in Ensemble is to use a low-level byte representation for the message payload and to use normal ML data structures for protocol headers. Thus, messages have two parts: payload and headers. The payload consists of sequence of bytes, but the headers are regular ML data structures.

```
/* total.c Message Headers */
struct to_header {
    uint8_t type;
    uint8_t flags;
    uint16_t dest;
    uint32_t token;
};

enum to_message_type {
    TO_TOKEN_REQUEST,
    TO_TOKEN,
    TO_DATA,
    TO_UNORDERED
};
```

Figure 4.1: Example of headers in Horus: type definitions.

The use of ML objects for headers is an important design feature of Ensemble and a departure from many previous communication systems. This design means that the payloads (which most affect performance but which protocols typically do not access) have an efficient implementation, while the protocol headers (which protocol layers manipulate a great deal) benefit from features of ML.

4.4.1 Protocol Headers

In Ensemble, each protocol layer has a data type for the headers it puts on messages. A layer pushes a header onto a message by tupling its header with the headers of the layers above it. At the destination, the headers are unmarshalled from the messages and passed to the layers. Each layer in turn extracts its header from a tuple and passes the remaining headers up to the layer above. At the application, the last header is removed and all that remains is the message body, which is passed to the application.

This still leaves the question of how the header object is linearized into a sequence of bytes at the bottom of the protocol stack so that it can be transmitted over the network along with the payload. Ensemble uses the ML

```
/* Send a message. */
...
msg = horus_message_alloc(to_memory,
                          "TO_TOKEN_REQUEST");
horus_message_add(msg, 0, sizeof(*hdr),
                (void **) &hdr);
hdr->type = TO_TOKEN_REQUEST;
hdr->flags = 0;
hdr->dest = 0;
hdr->token = htonl(group->seqno);
err = horus_cast(group->below, 0, msg);
...

/* Receive a message. */
void handler(event *ev, horus_message *msg) {
  enum to_message_type type ;
  unsigned seqno ;
  ...
  switch (event.type) {
  case HORUS_CAST:
    err = horus_message_read_byte(msg,&type) ;
    if (!horus_err_ok(err)) ...
    switch (type) {
      case TO_TOKEN_REQUEST:
        err = horus_message_read_nlong(msg,&seqno) ;
        if (!horus_err_ok(err)) ...
        ...
        break ;
        ...
    }  ...
  } ...
}
```

Figure 4.2: Example of headers in Horus: message handling code.

```
(* total.ml Message Headers *)
type header =
 | TokenRequest of int
 | Token of (int * int)
 | Data of int
 | Unordered

(* Sending a message. *)
...
down (castEv name) (TokenRequest token) ;
...

(* Receiving a message. *)
let up_handler event msg = match (getType event), msg with
| Cast, TokenRequest token ->
    (* Code to handle a token request *)
| Cast, Unordered ->
    (* Code to handle a non-token ordered message *)
    ...
| ...
```

Figure 4.3: Example of headers in Ensemble.

marshaller for this purpose. A *marshaller* is a function that takes a concrete data structure (embedded functions are not allowed) and linearizes it into a sequence of bytes from which a corresponding function can reconstruct a copy of the object. Marshallers typically transparently handle incompatibilities in byte ordering and word size. There are numerous standard marshalling formats such as XDR and ASN.1 [X.287]. Ensemble uses the general-purpose marshaller in Ocaml, although it can easily support other marshallers.

By representing headers as regular ML data structures, protocols can leverage the same powerful language features, such as pattern matching and type checking, that are used for other data structures. This greatly simplifies the construction of protocols and eliminates a great number of programming errors. Not only does the programmer not have to handle complications from low-level details such as incompatible machine byte ordering and word sizes, but compilers can detect problems such as mismatched header types and cases where not all header combinations are handled. Using normal ML data structures gives the protocols a higher level of abstraction because many implementation details are hidden.

The use of a marshaller has the potential to add significant overheads which do not exist in an architecture where protocol headers are created using low-level operations. The marshaller provided by Ocaml does introduce a small but significant amount of overhead, both in the size of messages and in processing costs. However, the optimizations presented in Chapter 3 eliminate this overhead. As with automatic garbage collection, marshalling was useful because it let us focus on just the critical cases by automating the rest.

Special purpose marshallers

Although Ensemble currently uses the Ocaml general-purpose marshaller, we are experimenting with using special purpose marshallers compiled from type information provided by the ML compiler. The normal ML marshaller takes an arbitrary ML data structure and uses tags in the data representation to marshal it. A marshaller specialized to the actual data type of a message can achieve a more compact representation and smaller marshalling/unmarshalling times than the general purpose marshaller, but the main benefit is that it would be better able to detect malformed messages. This is important for security in settings where an intruder may attempt to crash other processes by sending so-called *poison-pill* messages that are designed to violate the typing expectations of the protocols and cause run time type errors (which usually crash the process).

4.4.2 Message Payloads

Whereas protocol headers provide many opportunities to make use of features of ML, the application payload portion of messages raises a series of issues. This is largely because the nature of message payloads requires that they be represented as low-level sequences of bytes. One normally thinks of ML as a language best suited for manipulating high-level objects, and sequences of bytes fall outside of the domain where many features of ML can help. For instance, sequences of bytes in the payload often represent some high-level object, but type checkers are usually not able to capture this structure in useful ways. Thus, the question arises of whether ML is a good language for doing systems development where low-level objects often occur and where language support for them is important. Indeed, a major reason Ensemble benefits from the use of ML is that we have succeeded in abstracting much of the system at a high enough level that features of ML pay off. It is only in message payloads that Ensemble confronts issues associated with low-level objects. However, we feel that it is typical of many domains that most of the problem can be abstracted above low-level issues.

A message payload implementation must support a variety of operations, including allocation, release, subset (creating a new message from a subsequence of the bytes in another), and catenation. The subset and catenation operations are needed mostly for fragmentation-reassembly and message packing protocols. For instance, a fragmentation-reassembly protocol needs to be able to break a large message into smaller messages that fit the maximum message size supported by the network and reassemble it at the destination. All these operations should be efficient for messages of sizes ranging from 0 bytes to at least $10K$ bytes. They should not cause additional allocation for the body of the message nor should they copy the contents.

Message payloads in Ensemble are represented as arrays of records called **iovec**s (based on the UNIX data structure of this name). An **iovec** contains a pointer to a string, an integer offset, and an integer length. The offset specifies where the **iovec**'s body begins in the string and the length gives the number of bytes of data. Both the string and the other fields of the **iovec** are treated as read-only. **Iovec**s are exported to the rest of the system as an opaque, abstract data type. A subset of an **iovec** is created by allocating a new **iovec** record with the same string as the original but with different offset and length. Catenation is done by catenating arrays of **iovec**s. Thus, neither subset nor catenation copy the contents.

Some recent work has focused on introducing support in ML for low-level data structures such as untagged word arrays [TMC+96]. Such support is justified in part by the claim that it is needed in order to do real low-level systems work in ML. While this may be true for programs that interface directly with device drivers, Ensemble interacts with the network through system call stubs written in C, and it has not suffered from the absence of untagged word arrays. It would have been nice to have been able to implement Ensemble entirely in ML without these stubs, but the addition of less than 1000 lines of simple C code is a relatively insignificant portion of the system. Our experience with Ensemble has shown that the **string** core data type wrapped in **iovec**s with support from C system call stubs is sufficient at least for the needs of communication systems development. Of course, this not to say that untagged word arrays would not improve performance and/or memory usage, only that they are not a prerequisite to doing high-performance systems work.

4.5 Buffer management

Although ML strings wrapped with **iovec** records are sufficient for efficiently manipulating message payloads, a variety of memory management issues arise regarding how **iovec** strings are allocated and managed. The exact issues are involved with details of the Ocaml garbage collector, but the general lesson we learned was that garbage collectors may not be the best mechanism for managing data structures with a major impact on performance, such as messages. Some garbage collection strategies can cause unnecessary copying of data, bad fragmentation of memory, and slow recovery of memory. In the end, these problems drove us to explicitly manage message buffers, even though the rest of the system still benefits from automatic memory management.

4.5.1 First implementation

In our initial design, Ensemble allocated a new string prior to receiving a message from the network. This caused a variety of problems. First, because the length of a message received from the network is not known in advance, a string of the maximum transmission length had be allocated. For instance, the `recv()` system call in the UNIX BSD socket interface must be passed a buffer with sufficient space to contain the largest expected message size. This size can be up to $64K$ bytes, but for a number of reasons Ensemble typically uses messages of at most $10K$ bytes. Allocating a $10K$ byte block every time a message is received wastes a great deal of memory when the actual size turns out to be much smaller. In the case of a 100 byte message, 99% of the $10K$ byte space is wasted from internal fragmentation. Internal fragmentation refers to unused memory space allocated within an object and can result in a large waste of memory, as in our case. There are a variety of potential solutions to this problem. One option is to copy the message out of the buffer into a new string of the appropriate size. However, this causes a copy for each message, and external fragmentation (unused space outside of objects) is still a problem because the Ocaml garbage collector (as with many non-copying collectors) does a poor job of managing large blocks of varying sizes [WJNB95].

4.5.2 Using large message buffers

Both sorts of fragmentation are avoided in Ensemble by using very large strings for allocating **iovec**s. These strings are called *segments* and are typically $256K$ bytes long. Segments are managed by **msgbuf**s, which consist of a current segment and offset. Allocation from **msgbuf**s is done by creating an **iovec** record with the **msgbuf**'s segment and offset, and the desired length. The offset of the **msgbuf** is then advanced. If there is no longer enough space left to allocate the maximum size block from the **msgbuf**, a new segment is allocated and the offset is reset to zero. Allocation and release of **iovec**s from **msgbuf**s are inexpensive operations. Allocation usually consists of just advancing the **msgbuf** offset. Deallocation is done by the garbage collector once for each segment when there are no more references to a segment. The use of large segments has the potential drawback that a segment can only be released after the release of the last message using it. However, practice has found that this is not a problem as messages tend to have similar expected lifetimes.

The problem that arises with this design is that under even moderate loads much of the execution time (more than 25%) is used by the garbage collector in order to recover segments. Because we do a good job of avoiding other allocation, there is very little dead data to collect other than the (albeit large) segments, so these collections are inefficient. On each collection, the garbage collector scans the entire heap to collect a relatively small number of segments. Ocaml could be configured to wait longer between garbage collections, but this causes a lot of memory to be wasted. So we changed our approach again.

4.5.3 Reference counted management

We decided to add explicit reference counting to buffers. Each segment has an associated reference count that keeps track of the number of references to the buffer. When the reference count drops to zero, the segment is returned to a free list maintained by Ensemble. This form of reference counting is simple to implement because the objects being managed do not contain references to other objects. It eliminates our problems with the garbage collector because memory allocated for message payloads is rapidly recovered without requiring a garbage collection. The Ocaml garbage collector is only triggered by allocation on its heap, so when the apparent allocation rate decreases, the rate of garbage collections does also.

The use of reference counting adds some programming cost because protocols must correctly update the reference counts. We found that this was easy to do because the reference count operations are only needed when a protocol layer releases or buffers a message, and these operations are simple to recognize. The computational overhead of maintaining the reference counts is quite small because the compiler inlines the reference count operations at the call-site (we describe this in detail in Section 4.6). In addition, operations for managing the segment free list are efficient because the cost for each segment is amortized over all of the messages allocated from it.

The average time to allocate messages is graphed in Figure 4.4 for message sizes ranging from 4 bytes to $8K$. The measurements were taken on a 200 Mhz Intel Pentium Pro processor and were made by first growing the heap to a typical size for Ensemble and then allocating and releasing 50000 objects. The x-axis denotes the size of messages being allocated. The y-axis denotes the average time (in microseconds) to allocate and release one message. Note that both axis have logarithmic scales. The four lines correspond to (a) allocating 10K byte strings for every message, (b) allocating exact sized string for each message (and copying), (c) **msgbuf**s without reference counts, and (d) **msgbuf**s with reference counts. Option (a) is almost uniformly the worst. For 20 byte or smaller messages, option (b) performs best. For larger sized messages, (c) begins to perform better than (b) because the cost of copying starts to dominate the cost of garbage collection. (b) and (c) both climb significantly after 256 bytes because this is the lower threshold for allocating objects on the major heap (allocation/freeing on the Ocaml major heap is significantly more expensive than the minor heap). Option (d), however, is always close to the others and maintains low latencies throughout the range of message sizes (although in this test (b) has better performance than (d) for messages of less than 20 bytes, in the actual use in Ensemble (b) and (d) exhibit equivalent performance for these message sizes). This is because the garbage collector is rarely being activated.

Reference counts introduce the concern that they can cause both memory faults and memory leaks due to programmer errors. However, these problems can be prevented in Ensemble by enabling a debugging flag that causes reference counts for segments to be checked prior to allowing access to the segment. This slows execution somewhat, but prevents memory errors. The opposite problem, memory leakage, can occur if reference counts are occasionally not decremented to zero, causing segments to never be released. We addressed this problem by using weak pointers [Hay92] to detect when the reference count object (which wraps the segment) has no further references. When this happens, the problem is signaled to the user and the segment is recovered to prevent a memory leak (see Figure 4.5).

Both ISIS and Horus had similar problems. Messages are no less crucial data structures there than they are in Ensemble. The use of the system-provided memory allocation and release operations (**malloc()** and **free()**, respectively) were insufficient for managing messages, and both ISIS and Horus ended up developing their own sub-systems for managing memory associated with messages. These message sub-systems involved complex, multi-level, reference-counted data structures with special-purpose free lists containing preallocated and pre-formatted objects. The result was that message management was at least as complex as in Ensemble, and did not provide provide as good performance, even with all the optimizations done in C.

In summary, the memory management facilities for ML turned out to be insufficient for Ensemble and we had to add our own support to the system.

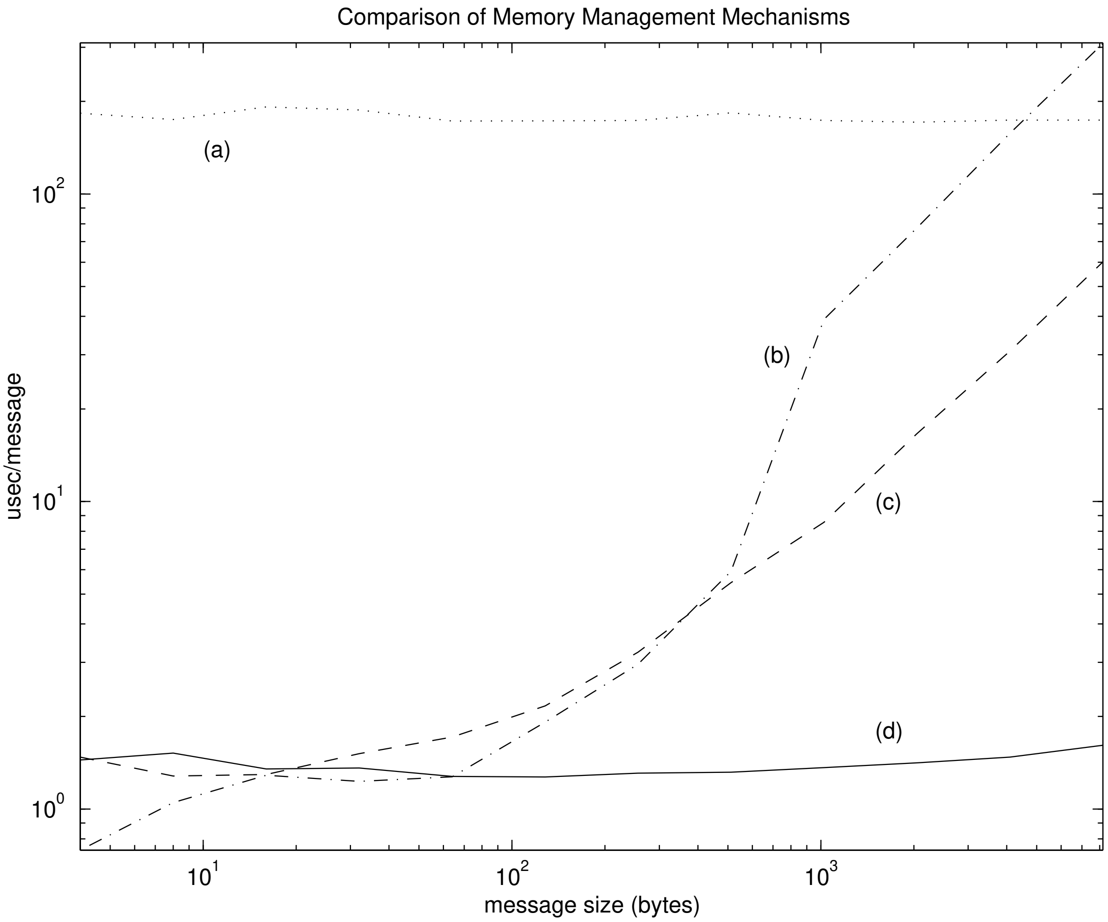

Figure 4.4: Comparison of the performance of management mechanisms used for **iovec**s. See text for an explanation

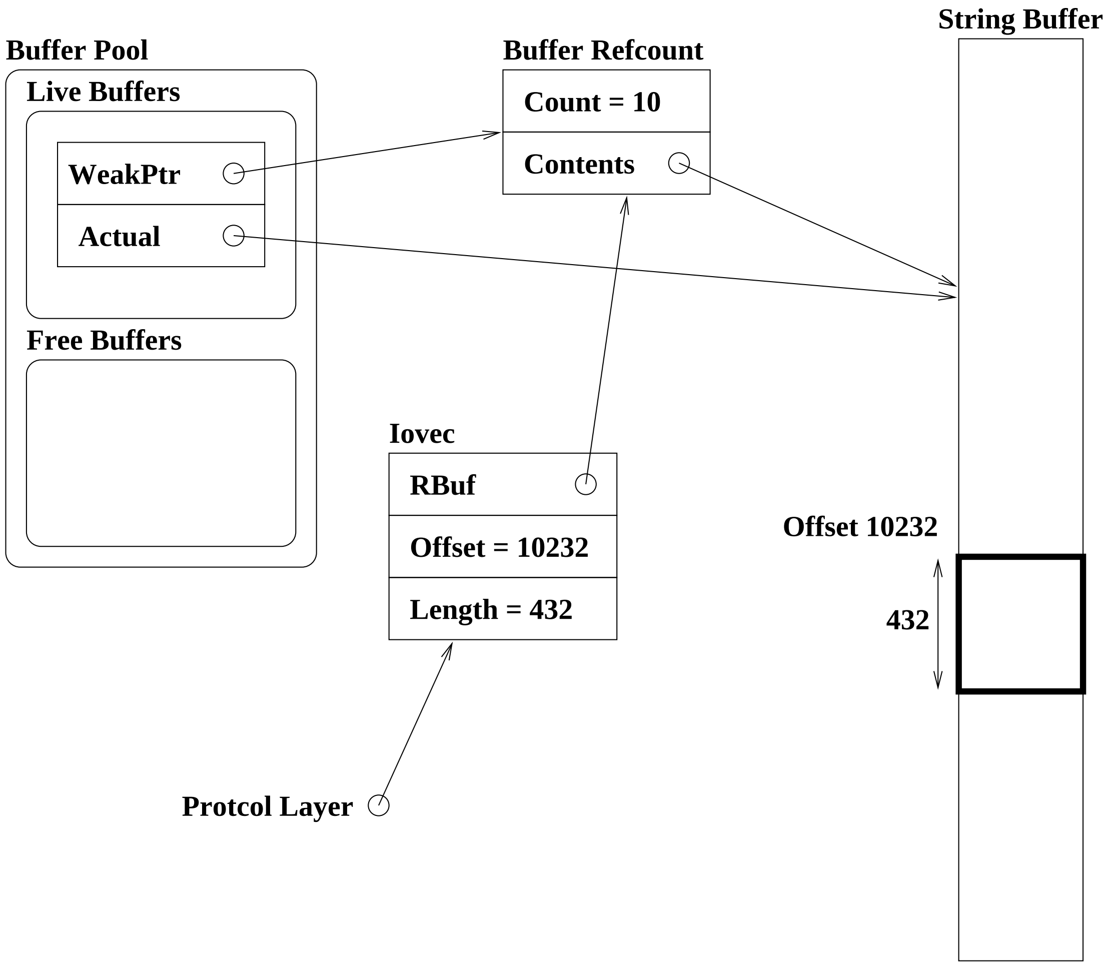

Figure 4.5: Depiction of the revised **iovec** structure. A protocol layer is given a pointer to an **iovec** record. The **iovec** contains a pointer to a **refcount** record. The **refcount** contains a reference count and a pointer to the string buffer. The **iovec** record contains the integer offset and length of data in the **Refcount** string. The buffer pool contains a weak reference to the **Refcount** record and a pointer to the string buffer.

It is important to be careful in how one views this. One could say that this is a failure of a garbage collected language because the garbage collector was not powerful enough to handle everything. However, it is a characteristic of systems style work that there are often a small set of data structures which require very careful management to achieve high performance. This was the case in Horus and ISIS, and it turns out that Ensemble is not any different in this respect. Ensemble's special management of messages highlights the usefulness of automatic garbage collection: the garbage collector handles the vast majority of memory management, allowing us to focus on the cases where specialized management is required. Whereas all data structures are explicitly managed in Horus and ISIS, in Ensemble only one is treated in this manner.

4.6 Inlining

Communications systems and other programs with a strong systems flavor often have multiple levels of abstraction barriers that must be crossed to manipulate data structures, even though the abstraction barriers often hide relatively small pieces of code. Ensemble exhibits this structure because of its layering and extensive use of modules. The abstraction barriers are useful because they increase modularity. The problem, of course, is that when there are lots of abstraction barriers to be crossed in doing inexpensive manipulations, the cost of the abstraction barriers (in the form of function call overhead) can be larger than the cost of the manipulations themselves.

Programmers in other languages such as C are familiar with this problem. They solve it by using macros or an inlining mechanism provided by the compiler. However, this often requires that the programmer annotate which functions are to be inlined. Ocaml also provides inlining support that eliminates the cost of these abstraction barriers, but without requiring annotation by the programmer. Inlining involves copying the body of a function to the sites at which the function is called, thereby eliminating the overhead of a function call. This is done at the potential cost of causing growth in the size of the compiled code, but we have not had any problems from the size exploding because the Ocaml compiler has a variety of heuristics to prevent excessive growth.

To see how all of this works, we present an operation in Ensemble, that of releasing the contents of an Ensemble event data structure (done at least twice for every message), and show how inlining eliminates the potentially costly abstraction barriers. In addition to the call site, there are four modules in this example. The call to the function **Event.free** results in turn to calls to **Iovec_array.free**, **Iovec.free**, and **Refcount.decr**. Each step adds one small part to the overall operation, such as dereferencing a record's field and calling another function on that field.

Through all the modules, string arguments are passed for use in debugging. These function arguments are called **debug** and the string passed in this example is **"FIFO"**. With this debugging information, all of the modules described here can be compiled to emit detailed traces. However, these debugging arguments are ignored in the normal code for the modules presented below.

```
(* call site *)
 ...
 Event.free "FIFO" ev
 ...
```

```
(* Event module *)

type t = {
  ty            : typ ;
  origin        : rank ;
  ranks         : rank list ;
  ack           : acknowledgement ;
  iov           : Iovec_array.t ;
  extend        : field list
}

(* Call the Iovec_array.free function on the iov
 * field.  Pass on the debug string unchanged.
 *)
let free debug ev =
  Iovec_array.free debug ev.iov
```

```
(* Iovec_array module *)

type t = Iovec.t array

(* Call the Iovec.free function on each entry in
 * the array.  Pass on the debug string unchanged.
 *)
let free debug ia =
  for i = 0 to (Array.length ia) - 1 do
    Iovec.free debug ia.(i)
  done
```

```
(* Iovec module *)

type t = {
  rbuf : string Refcount.t ;
  ofs : ofs ;
  len : len
}

(* Call the Refcount.decr function on the rbuf
 * field of the record.  Pass on the debug string
 * unchanged.
 *)
let free debug i =
  Refcount.decr debug i.rbuf
```

```
(* Refcount module *)

type 'a t = {
  mutable count : int ;
  obj : 'a ;
  mutable debug : (string * string) list
}

(* Decrement the reference count.  This version
 * ignores the debug argument.
 *)
let decr debug r =
  r.count <- r.count - 1
```

When compiled, all four levels of function calls are inlined at the caller. This happens even though all the module interfaces export abstract data types. Thus, the cost of crossing all of the abstraction barriers has been eliminated by inlining across modules. The call to **Event.free** is inlined by Ocaml like this:

```
let iova = ev.iov in
let lo = 0 in
let len = Array.length iova in
let hi = len - 1 in
for i = lo to hi do
  let iov = iova.(i) in
  let refcount = iov.rbuf in
  refcount.count <- refcount.count - 1
done
```

Note that because of abstract module interfaces, it is not possible for a programmer to write code that directly accesses the data structures as in the code generated after the inlining. Also, there was no cost at run time for passing debugging strings to the functions: the inlining exposed to the compiler the fact that the debugging string was not being used in the modules and so it was eliminated. This is the resulting assembly code (for the Intel 386 instruction set) that is generated for the original call to **Event.free** (annotated with the corresponding ML code from above):

```
    movl  16(%eax), %ecx      # let iova = ev.iova
    movl  $1, %eax            # let lo = 0
    movl  -4(%ecx), %ebx      # let len = Array.length iova
    shrl  $9, %ebx            #   contd.
    orl   $1, %ebx            #   contd.
    addl  $-2, %ebx           # let hi = len - 1
.L105:
    cmpl  %ebx, %eax          # for-loop termination
    jg    .L104               # escape if done
    movl  -2(%ecx, %eax, 2), %edi # let iov = iova.(i)
    movl  %edx, %esi          #   contd.
    movl  (%edi), %esi        # let rc = iov.rbuf
    addl  $-2, (%esi)         # rc.count <- rc.count - 1
    addl  $2, %eax            # i <- i + 1
    jmp   .L105
```

There is some room for further minor optimizations in the resulting code (mainly restructuring to eliminate one of the branch instructions in the loop), but the compiler has done a very good job of eliminating the abstraction barriers from the resulting code. Achieving similarly optimized code in C while maintaining opaque abstraction barriers would not be easy; in ML it takes no additional work on the programmer's part.

4.7 Conclusion

We began this chapter by asking the the question of what role can advanced programming languages such as ML play in the design and implementation of systems-style applications. We showed that ML allowed us to achieve high level of abstraction in the system, and that the language support allows system designer to focus on important parts of the system to achieve very good performance. We described how ML aided in significantly reducing the size of protocol layers, which are the most complex part of the system. We also showed in detail how the Objective Caml compiler compiles efficient code, and makes efficient use of memory.

Chapter 5

Formalization of layer optimizations

This chapter describes how optimizations in Chapter 3 can be formalized in type theory and implemented with the use of a theorem prover. This step is important for a number of reasons. First, carrying out the optimizations in this fashion gives strong guarantees regarding their correctness. Second, it demonstrates a general methodology for manipulating layered systems in a formal context. Third, we believe that formalization provides insight into the structure of the protocols and their optimization. As an example, we show how an actual Ensemble protocol stack is optimized using these techniques. The major contribution of this chapter is to demonstrate how the layering architecture from Chapter 2 and the optimizations from Chapter 3 mesh to enable the optimizations to be carried out in a formal fashion.

For these optimizations, we use the Nuprl theorem prover [C^{+}86, Jac94, Kre97, Con96]. Nuprl is an interactive theorem prover with support for automation. Proofs typically require human interaction with the theorem prover to direct the method of proof. However, the user can apply *tactics* [GMW79, CKB84] which are programs that encode various proof techniques, thereby automating those techniques. Nuprl's underlying semantics are based on a very expressive type theory. The Nuprl term language is similar to the subset of ML we use to implement protocol layers. This makes it straightforward to model protocol layers in Nuprl.

The formalization is described in several steps. First, we show how to translate imperative protocol layers into functional ones. Eliminating imperative operations is important because it is easier to reason about functional programs. Second, we give a functional composition operator for combining functional layers (following the layering model). The functional composition operator combines a stack of layers into a single layer. This step effectively makes reasoning about stack of layers the same as for a single layer, except that the resulting term object is larger than that for an individual layer. It is then straightforward to import the protocol stack into a theorem prover. Third, we show how the optimizations on protocol stacks presented in Chapter 3 can be carried out in a theorem prover. Fourth, as an example we present the application of these techniques to an actual Ensemble protocol stack with 4 protocol layers.

The process of implementing the optimizations in Nuprl is work in progress, and some of the description is of optimization steps that have not yet been implemented. The work with the Nuprl theorem prover described here was done primarily by Christoph Kreitz with help from Jason Hickey, Bob Constable, and Mark Hayden. Jason Hickey is developing a new version of Nuprl, called Nuprl-light, that addresses programming environment issues, such as those that have arisen in this work. The parts that have been implemented are the following: the automatic transformation of imperative layers into a functional form, the functional layer composition operator, trace conditions for the initial protocol stacks that are being optimized, extraction of the initial trace handler, and message compression. Clearly, there remains a good deal of work left in the implementation of these optimizations, although we believe there are no major technical barriers.

5.1 Functional layers

We begin by showing how to transform imperative protocol layers into purely functional layers. This step is described in detail in [Kre97]. The transformation to functional protocol layers is important in the later optimizations because the absence of imperative operations facilitates formal reasoning in Nuprl. We begin with protocol layers having the structure described in Section 2.1.8, and the resulting protocol layers use a subset of ML that is purely functional. The layers make no use of mutable state or exceptions (other than for aborting in the case of a protocol failure).

An imperative protocol layer has two forms of imperative operations to be eliminated: state updates and event emission. First, each layer has a state record containing mutable fields that are updated by the the layer handlers.

In the functional version, this state cannot be mutated, but layers still need to maintain some form of state. This is done by having each handler take as input the current state of the protocol layer and return as part of its output a new state record. For example, if a handler needs to update a field in its state, it creates a copy of the current state containing the modified field, and this copy is returned from the handler.

The second form of imperative operation is more subtle. It involves the way in which events are emitted from a layer. Recall that a handler takes as input one event and may emit zero or more events to pass to the handlers of the adjacent layers. In an imperative layer, event passing is implemented through the use of imperative queues into which events are deposited by calling a function from within the handler. However, a functional layer cannot use such queues. The solution to this is to emit events by returning them from the handler (along with the new state).

After eliminating the two forms of imperative operations, the layer handlers become functions that are called with the current state and a single event. When called, the handler processes the state and event and returns a pair of the new state and a list of zero or more events to pass to the adjacent protocol layers.

This transformation can be applied automatically on imperative layers through the use of a simple "compiler." This program parses the source file of an imperative protocol layer. It then identifies the handler functions in the layer (all the protocol layers follow a fixed structure). We modify the handlers so that the state and list of emitted events are "threaded" through each call. Where the imperative protocol handler would update a field in its state, the functional handler copies the state record, leaving all the fields the same except for the one to be updated. Where the imperative protocol handler would emit an event by calling a function that deposits it in an queue, the functional version adds it to the list of events to be emitted from the function.

5.2 Functional layer composition

The next step in formalizing the optimizations involves developing a functional layer composition operator that takes two functional layers as input and returns a new layer that is their composition. The difficulty in writing such an operator is that it needs to follow the layering model from Chapter 2, which basically requires that each protocol layer only be activated with one event at a time, and that events be passed between protocols layers in FIFO order.

Such an operator is straightforward to implement with imperative queues. Eliminating the imperative queues requires a moderately tricky design. Our version makes use of queues, but functional ones. A functional queue is similar to an imperative queue, except that the operations for adding elements to the tail and taking elements from the head both return a new queue instead of modifying the queue in-place (as an imperative implementation does).

The use of functional queues introduces some overhead: they are not as efficient as queues that use in-place modification. However, the optimizations applied later eliminate all of the queue operations from event traces through static analysis. So the overhead only appears outside the common case and does not significantly affect the performance.

The functional composition operator works by always executing events sequences to completion. Thus, when an event is injected into the top or bottom of a pair of composed protocol layers, the appropriate handler is called with the event. The events emitted from that layer are then split into two queues. One queue is for events that are being passed between the two layers (those that are passed up from the bottom layer and down from the top layer), and the other queue is for events emitted from the composed layer (down from the bottom layer and up from the top layer). The handler in the composed layer keeps on executing events in the former queue until there are no more left, at which point the events in the latter queue are returned.

With a composition operator, we can take any number of protocol layers and compose them into a single layer, thereby reducing the problem of reasoning about a stack of protocol layers into that of reasoning about a single (albeit large) layer. Because all of this is done functionally, it is straightforward to import the resulting layer into a theorem prover in order to carry out the optimizations.

5.3 Trace conditions

The optimizations start with two formal terms. The first is the stack of functional protocol layers composed using the operator just described. The second is the trace condition, which is a predicate on an event and the state of the protocol stack prior to injecting an event into it (see Section 3.3). The transformations we carry out on the layer term to generate the optimized layer are done with the assumption that the trace condition holds for the initial state and the event (in the actual system, the trace condition is checked prior to executing the optimized trace handler). In the description here, the trace condition is assumed to have been provided by the protocol designer, although it is conceivable that it could be generated through automated techniques that determine the normal-case conditions that apply to messages in a particular protocol stack. Two problems can arise if the trace condition is incorrectly specified. If it is too weak, some optimizations can not be made, limiting the effectiveness of the work. If the condition is too strong, some cases do not meet the condition, preventing the use of the optimized handler. Of course, a particular stack may have several trace conditions and trace handlers.

5.4 The optimizations in a functional context

We now describe how the optimizations presented earlier in Chapter 3 correspond to formal manipulations in a theorem prover. All the optimizations there have analogs in the context of this method. For some, the formalization is straightforward, while others are more challenging. The presentation here parallels that in the earlier chapter. For each optimization, we briefly review its purpose and then proceed by describing how it is carried out in our formal framework. These optimizations can be viewed as a sequence of simplifications or transformations, where on each pass we are able to prove that a modified protocol (with fewer operations) has equivalent behavior to the original if the trace condition is enabled.

5.4.1 Extracting the source code

The first pass extracts the relevant source code from each layer in the stack in order to generate the initial version of the handler. After this pass, all of the event queue operations for communicating between layers have been eliminated. These manipulations appear in the formalization through sequence of reductions on the code. For instance, when we are reasoning about the handler for particular layer, there may be different cases in the handler for the different types of events. Typically, the trace condition specifies the type of the event being injected. Using this information we can perform dead-code elimination on the handler to eliminate the dispatching operation on the event type and the cases for the other event types. By repeating these sorts of reductions, we can progressively eliminate code outside of the event trace until all that is left is the relevant code for the handler.

Note that in the description of the optimizations, we referred to the use of ad-hoc annotations in the protocol layers that allow the optimized handlers to be extracted from the constituent layers. In general, these annotations are required. However, annotations are not needed when we use a theorem prover. The trace condition replaces them: we can use the theorem prover to reason about which portion of each layer occurs in the optimized trace handler without the use of annotations. This is a benefit of our approach.

5.4.2 Intermediate data structures

This pass eliminates intermediate data structures, such as event records, that are allocated and used within a single call to a handler. By eliminating these data structures and shifting their contents to local variables, we can prevent unnecessary memory allocation and improve the time to access the information (local variables can be more easily assigned to registers where access times are much faster than data structures in memory).

In the formalization, this simply involves "flattening" data structures that are allocated in the trace handler but do not escape. For instance, assume that at one point in the handler a pair is allocated and later one of the elements is accessed:

```
let e = (a,b) in
  ...
  let (c,d) = e in
    if d then ...
```

The allocation of the pair can be eliminated and the contents can be substituted throughout the rest of the term, arriving at the following code:

```
    if b then ...
```

5.4.3 Inlining of functions

Because the trace handlers are relatively small and have a significant impact on the performance of the system, they are good candidates for inlining. With functional languages, inlining is a straightforward operation. It is done by substituting the definition of a function for one of its references, and then carrying out beta-reduction.

5.4.4 Traditional optimizations

Traditional optimizations are carried out by the Ocaml compiler after the code from the theorem prover is emitted.

5.4.5 Delayed processing

Another optimization involves splitting the trace handler operations into those that must be done before delivering a message to the application or transmitting it on the network, and those that can be done after. In this formalization, this optimization is simple because these two sets of operations are clearly divided by the return value of the handler. For instance, after optimization, the handler for a stack with two protocols has this form:

```
let handler (s1,s2) ev msg =
  ((update1(s1,ev,msg),update2(s2,ev,msg)),emit(s1,s2,ev,msg))
```

The state of the stack is a pair, and **update1**, **update2**, and **emit** correspond to the optimized terms for updating the state of the first protocol layer, the update for the second layer, and the function for calculating the events and messages to emit.

Delayed operations are moved outside the fast path by dividing the handler into two functions, **emit** and **update**. The **emit** function is computed and the resulting messages are emitted before the **update** function.

5.4.6 Compressing protocol headers

The final optimization involves compressing headers in normal case messages. This optimization both improves computational performance (smaller messages are often faster to manipulate) and the utilization of the network (a smaller portion of messages on the network consists of headers). First, the programmer identifies the common message formats and generates an optimized message data type where the common messages have a compact form. Second, the protocol stack is wrapped with conversion functions that translate between the optimized and unoptimized messages (messages being sent on the network are compressed into the optimized form and messages received from the network are expanded back into normal form). Third, the resulting stack is manipulated to eliminate the compression and expansion operations in the normal cases.

The first step, identifying the common message formats, is straightforward. The common messages are the ones that appear in the **emit** portion of the trace handler. For instance, one of the messages emitted may be:

```
Push(Hdr1(seqno),Hdr2)
```

What we want to do is eliminate the constant content and just send the non-constant portion (in this case, the integer **seqno**). Determining which parts of the message is constant and is currently done by the programmer, but could probably be automated for most cases. This can be formalized through a kind of representation analysis on the messages. We take the type of messages as they are emitted onto the network and wrap this type in another type. For instance, if the type of the messages is **msg**, then this wrapper type, **msg_opt**, would take this form:

```
type msg_opt =
| Normal of msg
| Opt0 of seqno
| Opt1 of ...
...
| Optn of ...
```

The **msg_opt** type can represent an arbitrary message by wrapping it as **Normal(msg)**. However, there are now message formats that contain just the information needed in the common cases. For instance the **Opt0(seqno)** would correspond to **Push(Hdr1(seqno),Hdr2)** (the **seqno** type is an alias for the core type for integers, **int**). Note that the **msg_opt** type corresponds to a particular protocol stack and way in which it is being optimized.

The new message type can be used with the original protocol stack by wrapping the stack with functions that compress the normal messages cases and expand the optimized ones. A handler function, called **handler**, is wrapped as follows:

```
let handler_opt states ev msg =
  (* Expand the input message from optimized to normal.
   *)
  let msg = expand msg in

    (* Run normal handler on the event and message.
     *)
    let (states,events) = handler states ev msg in

      (* Compress the output messages from normal to optimized
       *)
      let events =
        map (fun (ev,msg) ->
          (ev, (compress msg))
        ) events
      in

        (* Return new states and compressed messages.
         *)
        (states,events)
```

When an optimized message is received by this handler, it first expands it into a normal message. Then the unoptimized handler is executed on the event and message and it returns a list of events and messages to be emitted. It then scans through these (the **map** iterator takes the list of events and applies the given function to each item in the list to create a new list). For each message emitted, the **compress** function is called, which pattern matches for the optimized cases and, when it finds them, returns the optimized form of the message. If it does not find a match, the message is wrapped as **Normal(msg)**.

All the optimized message formats are now compressed before transmission on the network. However, when transmitting messages, the optimized handlers still are checking each of their emitted messages to see if they match the normal cases. Also, on receipt, the optimized messages are expanded unnecessarily. Both these operations can be eliminated by further transforming the optimized handlers so that the compression and expansion operations are moved to where the messages are actually being constructed or accessed and then eliminating the construction and compression operation. For instance, after the transformation, we may have this code segment:

```
(* Construct the message.
 *)
let msg = Push(Hdr1(seqno),Hdr2) in

  (* Compress the message.
   *)
  let msg_opt = match msg with
    (* Match for case 0.
     *)
  | Push(Hdr1(seqno),Hdr2) -> Opt0 seqno

    (* Match for other cases.
     *)
  | ...

    (* If all else fails, then wrap with Normal.
     *)
  | msg -> Normal(msg)
  in

    (* Return the event and optimized message.
     *)
    [(ev,msg_opt)]
```

We can statically reason that the message being constructed is always going to match the first pattern, and therefore reduce accordingly:

```
(* Construct the message directly.
 *)
let msg_opt = Opt0 seqno in

  (* Return the event and optimized message.
   *)
  [(ev,msg_opt)]
```

Through this static analysis, we have eliminated the operations for constructing the full message and the comparisons for detecting the optimized cases. An analogous transformation can be used at the receiver to leave the message in the optimized format in the call to the handler, thereby yielding similar improvements.

5.5 Reintroducing imperative operations

While a functional protocol stack is easier to manipulate formally, the functional implementation is likely to have worse performance than a corresponding imperative implementation. In particular, state updates with imperative operations are usually more efficient than the functional equivalent because they can be done in-place. It may therefore be advantageous to transform the trace handlers back into an imperative representation after completing the optimizations.

While we have not worked out the details of this reverse transformation, the general idea is straightforward. We take the functional representation of the new state returned by the handler and use it as a prescription for corresponding imperative updates to the imperative layer state.

5.6 Optimization of an actual protocol stack

To make the above presentation concrete, we present the transformation of an actual Ensemble protocol stack following this methodology. The optimized protocol stack we present is the default Ensemble protocol stack, without flow control protocols. We plan to add these protocols in the near future, however even without them the stack provides sufficient functionality for many applications.

The stack we optimize here includes four protocol layers. Our optimization approach involves composing these layers into a single layer and then optimizing the important event traces for that stack. However, in practice the "stack" we optimize is really a sub-stack of the stacks that are used in practice. Only the layers below the application representative layer affect the normal performance of the stack, so it is sufficient to optimize just these and use unoptimized layers for everything above the application. In a sense, we are optimizing a wide range of protocol stacks because there are a large number (more than 1000) combinations of existing layers in Ensemble that can be usefully added above the application.

The four layers being optimized are (starting closest to the application) *Frag*, *Pt2pt*, *Mnak*, and *Bottom*. We briefly describe these along with the common cases that we optimize for them.

- The *Frag* layer implements fragmentation and reassembly for messages that are larger than the maximum size transmitted on the network. For this layer, we optimize the case where the message is small enough that fragmentation is not required. In this case, the layer adds a trivial header and passes the message directly through.

- The *Pt2pt* layer implements reliable point-to-point message transmission. The protocol (which is similar to the retransmission protocol for TCP) adds a sequence number to each message's header and buffers a copy until it is acknowledged by the destination. If an acknowledgement is not received after a certain amount of time (this timeout is a parameter of the protocol), the protocol layer begins retransmitting the message at regular intervals. However, the normal behavior is for the initial transmission to be successful and for messages to arrive in order. This is the case that we optimize. In addition, we optimize the treatment of multicast messages which pass untouched directly through this layer.

- The *Mnak* layer implements reliable multicast message transmission. In a similar fashion to the *Pt2pt* protocol, this protocol adds sequence numbers to all messages and buffers them until they are acknowledged. However, retransmission is carried out through NAKs, where each destination explicitly requests retransmissions when it detects missing messages. Again, the normal case is for the network to behave reliably and deliver multicast messages in the order they were sent. In addition, this layer ignores point-to-point messages, which are handled by the *Pt2pt* layer.

- The *Bottom* layer serves a number of special purposes in Ensemble. However its behavior in the normal case is quite simple. It keeps track of which endpoints in the group have been marked as failed by the membership protocols and drops messages that arrive from those endpoints after they have "failed"[1]. This is important because it provides the illusion to the rest of the stack that failed endpoints have really failed (they are not heard from again), and considerably simplifies other protocols. Thus, this layer only puts trivial headers on multicast and point-to-point messages and the only operation it does in the normal case is on receipt to check if the origin of a message has failed. The case that is optimized is that the origin has not failed.

For this protocol stack, there are four event traces to be optimized (send and receive for both point-to-point and multicast messages). However, in the discussion that follows, we focus on only one case, the receipt of a multicast message. We first show for two of the layers (*Bottom* and *Mnak*) the trace condition and resulting optimized trace handler that are used for this event trace. We present a theorem that governs how layers are composed under certain kinds of traces. We then present the resulting trace handler that we extracted from the four layer protocol stack for this condition (represented as a Nuprl theorem). Finally, we show the application of the message compression optimization to this handler.

5.6.1 Independent optimization of layers

The first step in carrying out the optimization is individually to prove theorems for the behavior of the layers under the trace condition. We later assemble the individual theorems and use them to establish further theorems about the composition of the layers. This bottom-up, divide-and-conquer approach allows the problem to be sub-divided, which substantially improves the performance of the theorem prover.

All of the Ensemble protocols are divided into separate handlers for up and down events and each of these handlers typically do an immediate case

[1]The process may not have actually failed, but may merely be unresponsive or unreachable via the network.

```
match getType ev, hdr with

  (* For normal Cast and Send messages, just check
   * that the origin has not failed.
   *)
| (ECast|ESend), NoHdr ->
    if not s.failed.(getPeer ev) then
      (* Common case: origin was alive.
       *)
      up ev abv
    else
      free name ev
| ...
```

Figure 5.1: Portion of handler for normal case messages in the *Bottom* layer.

split on the type of the event and the header of the message. For the *Mnak* and *Bottom* layers, the relevant portions for the event traces that we are concerned with are in Figures 5.1 and 5.2.

The purpose of this first step is to extract the code from these arms of the case split that apply in the event trace we are optimizing. To do this, we determine the trace condition to use for the event trace. In this event trace, the event type is expected to be **Cast** (a multicast event), the header for *Bottom* is **NoHdr** ("no header"), and the header for *Mnak* is **Data(seqno)** (where **seqno** is the sequence number of the message).

Each of the layers adds an additional condition, as described in the protocol descriptions above. For *Bottom*, the condition is that the origin of the message is not considered to be failed. The origin of the message is in the **peer** field of the event, and this condition is checked by examining the corresponding array entry of the **failed** field of the *Bottom* layer's state. With this condition enabled, we can then prove that a simpler handler suffices for this layer.

```
match getType ev, hdr with

  (* ECast:Data: Got a data message from other
   * member.  Check for fast path or call recv_cast.
   *)
| ECast, Data(seqno) ->
    let origin = getPeer ev in
    let buf = s.buf.(origin) in
    let iov = getIov ev in

    (* Check for fast-path.
     *)
    if Iq.opt_insert_check buf seqno then (
      (* Fast-path.
       *)
      s.buf.(origin) <- Iq.opt_insert_doread buf seqno iov abv ;
      up ev abv
    ) else (
      recv_cast origin seqno abv (getIov ev) ;
      free name ev
    )

| ...
```

Figure 5.2: Portion of handler for normal case messages in the *Mnak* layer.

```
∀vs:View.state.
∀hdlr:Layer.handler.
∀s:Bottom.state.
∀ev:Event.t.
∀hdr:Bottom.header.
∀msg:Message.TYPES.
∀peer:Trans.rank.
  hdlr = snd (convert Bottom.l vs)
  ⇒
    getPeer ev = peer
    ∧ getType ev = Cast
    ∧ hdr = NoHdr
    ∧ s.failed.(peer) = false
    ⇒
         hdlr (s, UpM(ev, Full(hdr, msg)))
       = (s, [=>UpM(ev, msg)<=])
```

Figure 5.3: Optimization theorem for the *Bottom* layer.

The corresponding theorem appears in Figure 5.3[2]. The first portion of the theorem is a series of universal quantifiers that specify the types over which the variables later in the theorem range. For instance, the variable **s** ranges over the type **Bottom.state**, the type of the state records for the *Bottom* layer. Next come two sets of assumptions. The first set of assumptions describe how the protocol layer has been initialized. In this case, there is one assumption: that the handler was generated by applying the **convert** function to **Bottom.l** (the implementation of the *Bottom* layer) and a view state, **vs**. The reasons for the initialization being done in this fashion are beyond the scope of this description. Suffice it to say that the resulting handler is bound to the **hdlr** variable and is a function that takes a pair of a state record and an event and returns pair of a new state and queue of events. The next set of assumptions are the trace condition for this layer. For instance, in this case the "type" of the event is **Cast**:

```
getType ev = Cast
```

The last part of the theorem is a statement about the behavior of the handler under both of the above sets of assumptions. In this case, the state is returned unmodified and the event and message are passed up (after the header for this layer has been removed). The syntax, [=> **x** <=], represents a functional queue with a single item in it (this is analogous to the notation commonly used for instantiating lists, [**x**]). The theorem can be summarized as follows: if certain conditions hold on the state of the *Bottom* layer and an event and message to be injected into it (together, these form the trace condition), then the normal execution of the layer is equivalent to a simplified handler that does not update the state, and returns just the event that was injected into the layer (this handler is the trace handler).

We give the corresponding theorem for the *Mnak* layer in Figure 5.4. The condition for *Mnak* is that the sequence number in the header should be the next expected sequence number from the endpoint. This is checked by calling

[2]As the point of this description is to demonstrate an application of the methodology and not provide a tutorial for Nuprl, the theorems presented here have been modified from those generated using Nuprl. The changes include additional formatting, renaming of variables, elimination of some extraneous information (such as some of the type information), and elimination of some details of Ensemble that would otherwise obscure the presentation.

```
∀vs:View.state.
∀hdlr:Layer.handler.
∀s:Mnak.state.
∀ev:Event.t.
∀hdr:Mnak.header.
∀msg:Message.TYPES.
∀seqno:Trans.seqno.
∀peer:Trans.rank.
  hdlr = snd (convert Mnak.l vs)
  ⇒
    getPeer ev = peer
    ∧ getType ev = Cast
    ∧ hdr = Data(seqno)
    ∧ Iq.opt_insert_check s.buf.(peer) seqno = true
    ⇒
         hdlr (s, UpM(ev, Full(hdr, msg)))
       = ((s[.buf←s.buf[.(peer)←
            Iq.opt_insert_doread s.buf.(peer)
              seqno
              (getIov ev)
              msg]]),
          [=>UpM(ev, msg)<=])
```

Figure 5.4: Optimization theorem for the *Mnak* layer.

```
∀TopHdlrs,BotHdlrs,CpsHdlrs,TopState,BotState:TYPES.
∀Top:View.state -> TopState * TopHdlrs.
∀Bot:View.state -> BotState * BotHdlrs.
∀vs:View.state.
∀ev:Event.t.
∀top_hdlr:TopHdlrs.
∀bot_hdlr:BotHdlrs.
∀cps_hdlr:CpsHdlrs.
∀ev:Event.t.
∀msg0,msg1,msg2:Message.TYPES.
∀s1,s1a:TopState.
∀s2,s2a:BotState.
  top_hdlr = snd (Top vs)
  ∧ bot_hdlr = snd (Bot vs)
  ∧ cps_hdlr = snd (compose Top Bot vs)
  ⇒
    top_hdlr (s1, UpM(ev, msg1)) = (s1a, [=>UpM(ev,msg2)<=])
    ∧ bot_hdlr (s2, UpM(ev, msg0)) = (s2a, [=>UpM(ev,msg1)<=])
    ⇒
      (cps_hdlr ((s1, s2), UpM(ev, msg0))
    = ((s1a, s2a), [=>UpM(ev, msg2)<=]))
```

Figure 5.5: The upward linear layer composition theorem.

the **Iq.opt_insert_check** function on the message buffer corresponding to the origin of the message (the **Iq** module implements "infinite queues," the abstraction we use for managing message buffers). The handler is similar to that of *Bottom* in that the event is then emitted unmodified. However, the state is updated by adding the message into the buffer.

5.6.2 Composing the handlers

With the above theorems for the individual layers in hand, the next step is to prove the corresponding theorem for the stack. Before we move on to this, we first introduce a theorem called the *upward linear layer composition theorem* (ULLC). ULLC simplifies the composition of the trace conditions and handlers of the individual layers. It states that when two layers pass an event directly up, their composition also passes that event directly up. An analogous theorem exists for the downward case. ULLC is used as a building block for constructing trace conditions and handlers for linear event traces. We envision constructing a library of such theorems to form an "algebra" of protocol layers and optimizations. The theorem appears in Figure 5.5.

With the layer composition theorem, we can now compose the conditions and handlers for the event trace we are using as an example. This appears in Figure 5.6. The theorem is different from the theorems for the *Bottom* and *Mnak* layers only in that the "layer" being reasoned about is the composition of four layers. Otherwise, it is similar in that it states that if certain conditions hold on the state of the layer (here the state is really the states of the four constituent layers), then the normal execution of the layer is equivalent to a simplified handler. Note that the intermediate event queue operations introduced by the **compose** function have been entirely eliminated, leaving only the creation of a singleton event queue.

5.6.3 Message compression

After extracting the trace condition, there are a number of further optimizations to apply to the protocol layer. An important one that we present here is the message compression optimization. This follows directly along the lines described above in Section 5.6.3. The result of the optimization is a transformed protocol stack that uses an alternate message format that is more efficient for the common cases. The corresponding theorem appears in Figure 5.7. Note that where the full-sized message **Full(NoHdr,Full(...))** appeared in Figure 5.6, these have been replaced with the optimized form, **Opt0(seqno,msg)**.

5.7 Automation

People often have misconceptions regarding the degree of automation in theorem provers such as Nuprl. While very simple proofs can sometimes be tackled automatically by Nuprl, proofs in general require human guidance. For instance, the optimizations in this chapter required a significant amount

```
∀vs:View.state.
∀hdlr:Layer.handler.
∀s_frag:Frag.state.∀s_pt2pt:Pt2pt.state.
∀s_mnak:Mnak.state.∀s_bottom:Bottom.state.
∀ev:Event.t.∀peer:Trans.rank.
∀msg:Message.TYPES.∀seqno:Trans.seqno.
  hdlr = snd
           ((compose (convert Frag.l) (compose (convert Pt2pt.l)
            (compose (convert Mnak.l) (convert Bottom.l)))) vs)
  ⇒
    getPeer ev = peer
    getType ev = Cast ∈ Event.typ
    ∧ Iq.opt_insert_check s_mnak.buf.(peer) seqno = true
    ∧ s_bottom.failed.(peer) = false
    ⇒
         hdlr ((s_frag,s_pt2pt,s_mnak,s_bottom),
               UpM(ev,
                Full(Bottom.NoHdr,
                 Full(Mnak.Data(seqno),
                  Full(Pt2pt.NoHdr,
                   Full(Frag.NoHdr,msg))))))
       = ((s_frag
         , s_pt2pt
         , s_mnak[.buf←s_mnak.buf[.(peer)
            ←Iq.opt_insert_doread s_mnak.buf.(peer)
              seqno
              (getIov ev)
              Full(Pt2pt.NoHdr, Full(Frag.NoHdr, msg))]]
         , s_bottom)
         , [=>UpM(ev, msg)<=])
```

Figure 5.6: The optimized stack theorem.

```
∀vs:View.state.
∀hdlr:Layer.handler.
∀s_frag:Frag.state.∀s_pt2pt:Pt2pt.state.
∀s_mnak:Mnak.state.∀s_bottom:Bottom.state.
∀ev:Event.t.∀peer:Trans.rank.
∀msg:Message.TYPES.∀seqno:Trans.seqno.
  hdlr = snd
           ((compose (convert Frag.l) (compose (convert Pt2pt.l)
            (compose (convert Mnak.l) (convert Bottom.l)))) vs)
  ⇒
    getPeer ev = peer
    ∧ getType ev = Cast
    ∧ Iq.opt_insert_check s_mnak.buf.(peer) seqno = true
    ∧ s_bottom.failed.(peer) = false
    ⇒
         Comp.handler_opt hdlr
         ((s_frag,s_pt2pt,s_mnak,s_bottom),
          UpM(ev,Opt1(seqno,msg)))
       = ((s_frag
         , s_pt2pt
         , s_mnak[.buf←s_mnak.buf[.(peer)
            ←Iq.opt_insert_doread s_mnak.buf.(peer)
              seqno
              (getIov ev)
              Full(Pt2pt.NoHdr, Full(Frag.NoHdr, msg))]]
         , s_bottom)
         , [=>UpM(ev, msg)<=])
```

Figure 5.7: The optimized stack theorem with message compression.

of human interaction. However, proof tactics that encode various strategies can decrease the amount of interaction as tactic "libraries" are developed and refined. One area for future work is the further development of these tactics to automate the optimizations.

The ultimate goal of complete automation involves support that would allow developers of a new protocol stack to have an optimizer automatically detect trace conditions, extract trace handlers, determine the optimized message representation, ... Some stacks would undoubtedly still require human interaction at various steps, but the hope would be that an optimizer could capture most protocol stacks without this interaction.

Totally automated optimization is still a long way off. Currently, these portions of the optimization process have been automated to a large degree: importing protocol layers into Nuprl, extraction of trace handlers from individual layers (given the trace condition), and application of the ULLC theorem to compose trace handlers together. The rest of the optimizations either require a good deal of human interaction or still have not been implemented in the context of Nuprl.

A question raised by this work is whether or not the optimization framework will eventually be subsumed by optimizations implemented in standard compilers. We believe this will not happen because many of the reasoning steps involved in the optimizations are quite complex and specific to the problem domain. While we believe most or all of the steps could be automated for most protocol stacks, this will require a large degree of domain-specific knowledge about protocol optimization. Indeed, the optimizations are generated through tactic programs that encode complex proof techniques. It is unlikely that compilers would be able to replicate these kinds of manipulations without also requiring the protocol designer to "program" the optimizations, at which point the compiler would begin to appear very much like a theorem prover.

5.8 Conclusion

This chapter demonstrated how the protocol layer optimizations can be implemented using a theorem prover. We began by introducing the Nuprl theorem prover and providing a brief background. Then we showed how imperative protocol layers can be transformed into a functional representation and we gave a composition operator for such layers. We then described how each of the optimizations from Chapter 3 can be made on the functional layers through the use of a theorem prover. Finally, we presented an actual Ensemble protocol stack and demonstrated several of the optimizations on it as a proof of concept.

Chapter 6

Conclusion

This thesis has addressed issues in building communication systems with very high levels of performance and flexibility. We began by presenting the Ensemble architecture, describing the components, their interactions, and how this architecture compares with other communication systems.

We then examined the performance problems that arise in highly layered communication systems. We showed where these performance inefficiencies arose and showed how to structure the system in order to eliminate these inefficiencies through a sequence of high level optimizations.

We examined how advanced programming languages such as ML can aid in the design and implementation of such systems. We showed how ML helped in reducing the size of the protocol layers and in exposing structure of the system that allowed for further decomposing the layers and improving the performance. We showed that ML allowed us to achieve high level of abstraction in the system, and that the language support allows system designer to focus on important parts of the system have very good performance.

In the final chapter, we continued the development of the protocol optimizations, showing how to formalize them and implement them using a theorem prover.

Looking ahead, there are many directions to pursue. The formal structure constructed for carrying out the protocol optimizations in Nuprl was designed to support protocol verification as a next step. The advantage of our approach is that the use of ML in the protocol layers means that verification is done on the actual, executable protocol layers instead of just specifications or abstracted versions of the executable protocols, as is commonly done.

One question that remains to be answered is whether the protocol optimizations in Chapter 5 can be automated to the degree that they can be used by protocol designers who themselves do not have much experience with theorem provers. Bringing the optimization technology to this level would have repercussions in the areas of distributed systems and theorem proving. Protocol designers would be given a powerful tool for further developing extremely flexible protocol suites without having to pay a price in performance. Theorem proving researchers would be given an active set of users applying their tools to real world problems on a daily basis. Whether or not this goal can be achieved remains to be seen, however.

One problem in automating the optimizations involves automatically determining the trace conditions for a protocol stack. An interesting solution to this would be to use profiling data to determine the common execution traces for a protocol stack and from this information derive the trace conditions. One can imagine using this approach to adaptively optimize protocol stacks in which the trace handlers that are important to optimize depend on the behavior of the application. The idea is to profile protocol stacks running under an application and regularly generate new trace conditions based on recent profiling data. When the set of trace conditions changes, the optimizer could be executed in the background to generate optimized trace handlers for the new trace conditions. When the new handlers are ready, they can be dynamically linked to the running process and the protocol stack can switch to them[1]. Thus the optimized protocol stack would be regularly adapting to the application workload.

As described in Chapter 4 the use of ML in Ensemble raises a number of issues regarding the usefulness of advanced programming languages such as ML in systems style settings. Experience gained with Ensemble has helped us to understand many of these issues, and this experience has been fed back into the Ocaml compiler, resulting in improvements in garbage collection, marshalling facility, interoperability with C, ... Other programming language projects are now using Ensemble as a example systems style application for use in programming language experiments. We hope that Ensemble will continue to serve as a vehicle for breaking down barriers to using ML in systems style projects.

[1] As mentioned in Chapter 2, Ensemble supports both dynamic linking of new protocols and switching protocol stacks on-the-fly.

BIBLIOGRAPHY

[AWWV96] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang.* Prentice Hall, 1996.

[Bas97] Anindya Basu. *A Language-based Approach to Protocol Construction.* PhD thesis, Cornell University, Cornell University, August 1997.

[BBVvE95] Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-Net: A user-level network interface for parallel and distributed computing. In *Proc. of the Fifteenth ACM Symp. on Operating Systems Principles*, pages 40–53, Copper Mountain Resort, CO, December 1995.

[BHLM94] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. Signatures for a network protocol stack: A systems application for Standard ML. In *Proc. of the ACM Conf. on Lisp and Functional Programming*, Orlando, Florida, June 1994.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of the Eleventh ACM Symp. on Operating Systems Principles*, pages 123–138, Austin, TX, November 1987.

[BvR94] Kenneth P. Birman and Robbert van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, Los Alamitos, CA, 1994.

[C+86] Robert L. Constable et al. *Implementing Mathematics in the NuPRL Proof Development System.* Prentice–Hall, 1986.

[CHTCB95] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. Technical Report TR95-1548, Cornell University, October 1995.

[CJRS89] David Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.

[CKB84] Robert L. Constable, T. Knoblock, and J. L. Bates. Writing programs that construct proofs. *J. Automated Reasoning*, 1(3):285–326, 1984.

[Con96] Robert L. Constable. *The Structure of Nuprl's Type Theory in Logic and Computation.* NATO ASI Series. Springer Verlag, 1996.

[CT90] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. of the 1990 ACM Symp. on Communications Architectures & Protocols*, pages 200–208, September 1990.

[CZ85] David Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[FLS97] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. In *Proc. of the Sixteenth ACM Symp. on Principles of Distributed Computing*, pages 53–62, Santa Barbara, CA, August 1997.

[GMW79] Michael Gordon, Robin Milner, and Christoph Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 73 of *Lecture Notes on Computer Science.* Springer-Verlag, 1979.

[Har96] The Harlequin Group, Cambridge. *The MLWorks User Guide*, November 1996.

[Hau94] Bogumil Hausman. Turbo Erlang: Approaching the speed of C. In Evan Tick and Giancarlo Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, 1994.

[Hay92] Barry Hayes. Finalization in the garbage collector interface. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, volume 637, pages 277–298. Springer Verlag LNCS, St. Malo, France, September 1992.

[Jac90] Van Jacobson. Compressing TCP/IP headers for low-speed serial links. RFC 1144, Network Working Group, February 1990.

[Jac94] Paul B. Jackson. *The Nuprl Proof Development System, Version 4.1 Reference and User's Guide*. Cornell University, Ithaca, NY, February 1994.

[Kar97] David A. Karr. *Specification, Composition, and Automated Verification of Layered Communication Protocols*. PhD thesis, Cornell University, Cornell University, March 1997.

[Kay95] Jonathan Kay. *Path IDS: A Mechanism for Reducing Network Software Latency*. PhD thesis, University of California, San Diego, 1995.

[Kre97] Christoph Kreitz. Formal reasoning about communication systems I: Embedding ML into type theory. Technical Report TR97-1637, Cornell University, July 1997.

[Kru93] Clifford Dale Krumvieda. *Distributed ML: Abstractions for Efficient and Fault-Tolerant Programming*. PhD thesis, Cornell University, Cornell University, August 1993.

[Ler97] Xavier Leroy. *The Objective Caml system release 1.05*. INRIA, France, May 1997.

[Mac93] David MacQueen. Reflections on Standard ML. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693, pages 32–46. Springer Verlag LNCS, 1993.

[Maf95] Silvano Maffeis. Adding group communication and fault-tolerance to CORBA. In *Proc. of the 1995 USENIX Conference on Object-Oriented Technologies*, Monterey, CA, June 1995. USENIX.

[MOR$^+$96] Keith Marzullo, Michael Ogg, Aleta Ricciardi, Alessandro Amoroso, F. Andrew Calkins, and Eric Rothfus. NILE: Wide-area computing for high energy physics. In *Proc. of the of the 7th European SIGOPS Workshop*, Connemara, Ireland, September 1996.

[MP96] David Mosberger and Larry Peterson. Making paths explicit in the Scout operating system. In *Proc. of the of the 1996Symp. on Operating Systems Design and Implementation*, pages 153–168, Seattle, Washington, October 1996.

[MPBO96] David Mosberger, Larry L. Peterson, Patrick G. Bridges, and Sean O'Malley. Analysis of techniques to improve protocol processing latency. In *Proc. of the 1996 ACM Symp. on Communications Architectures & Protocols*, pages 73–84, Stanford, September 1996.

[PHOA93] Larry L. Peterson, Norm Hutchinson, Sean O'Malley, and Mark Abbott. RPC in the x-Kernel: Evaluating new design techniques. In *Proc. of the Fourteenth ACM Symp. on Operating Systems Principles*, pages 91–101, Asheville, NC, December 1993.

[Pos81] Jon Postel. Transmission Control Protocol. RFC 793, September 1981.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *Proc. of the ACM SIGPLANN '91 Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.

[Rit84] Dennis M. Ritchie. A stream input-output system. *Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[TMC$^+$96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. of the 1996 SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

[TVEB95] Anindya Basu Thorsten Von Eicken, Veena Avula and Vineet Buch. Low-latency communication over atm networks using active messages. Technical Report TR94-1456, Cornell University, March 1995.

[vR96] Robbert van Renesse. Masking the overhead of protocol layering. In *Proc. of the 1996 ACM Symp. on Communications Architectures & Protocols*, Stanford, September 1996.

[vRBM96] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. of the International Workshop on Memory Management*, Kinross, Scotland, UK, September 1995.

[WL93] Pierre Weis and Xavier Leroy. *Le Language Caml*. InterEditions, Paris, 1993.

[X.287] CCITT Recommendation X.208. Specification of Abstract Syntax Notation One (ASN.1), 1987.