BUILDING ADAPTIVE SYSTEMS USING ENSEMBLE

ROBBERT VAN RENESSE, KEN BIRMAN, MARK HAYDEN,
ALEXEY VAYSBURD, DAVID KARR

Trends in networking and distributed computing are creating a new generation of applications that must adapt as the environment within which they execute changes. Examples of adaptation include switching protocols to overcome a security exposure or failure mode seen only in certain settings, changing data rates to accommodate a slow link, or adapting the behavior of a high level application to match the set of participants using the application. We describe the Ensemble system, a tool for building adaptive distributed programs.

INTRODUCTION

Computer systems that adapt to changing environments are of increasing importance in a great many settings. Consider a teleconferencing system. Such a system may be able to run at very high video-frame rates and with little data compression when executing over the highest speed local-area networks. However, when the background load on the network or a participating host increases, it may need to reduce the video-frame rate or switch to a different compression mechanism. If one of the hosts is mobile, the application may alternate between being strongly and loosely connected—modes of connection that require different networking protocols. If that host wanders out of the confines of a security firewall, or some other host from outside the firewall joins a conference that was until then within the firewall, the system has to switch to using signing and possibly encryption mechanisms.

Traditionally, a system designer would approach this problem by viewing it as an aspect of application functionality. Solving it would require application-specific code that anticipates, detects, and adapts the system for each of some set of situations. Such code is likely to be complex and would not be perceived as adding value to the "core" functionality of the application, hence the designer will view the development of these mechanisms as a burden. The alternative we propose here involves writing many pieces of simple code, *modules*, each one dealing with a small problem in a particular situation, and to configure and reconfigure these modules together on the fly.

Our approach has potential problems. It requires a careful design of the interfaces between the modules, and has the potential to introduce significant overheads. Also, reconfiguration of an active system is somewhat analogous to changing the engines on an airplane while it is flying: doing so involves delicate synchronization which, if not handled appropriately, could easily prove so disruptive as to shut the application down. In particular, coordinating adaptations so that all programs in the application end up configured in a consistent manner, and doing this while the system continuously provides service, represent serious challenges. Nevertheless, if these problems can be overcome, reconfigurable systems are appealing. Such systems are relatively easy to implement and verify, the modules can be reused, and the system can potentially be configured in new ways (when necessary, with new modules) to deal with new and unanticipated environments. A

more traditional, monolithic, development approach lacks these properties, making such systems harder to maintain.

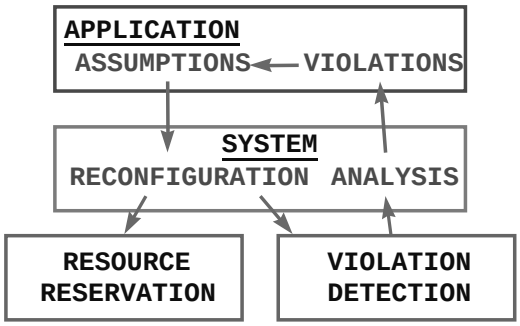

Figure 1 The process of adapting to the environment.

PROGRAMMING ADAPTIVITY

For a system to reconfigure when necessary, the application has to explicitly characterize the conditions under which adaptation will be required. The process is depicted in Figure 1. When launched, the application makes initial assumptions about the environment. For example, the application may specify that it is running within a firewall on a high-performance network. From these assumptions, the system creates resource reservation requests to try to meet the assumptions as well as possible, but also installs *sensors* that will detect potential violations of these assumptions (preferably, but not necessarily, before they happen). The output from the sensors is analyzed, and violations are reported back to the application. The application then adjusts its assumptions, leading to a reconfiguration. The Ensemble system, developed at Cornell University, represents a framework for standardizing solutions to this problem [Hayden 1997].

Adaptive systems can be understood as a generalization of other forms of coordinated distributed behaviors. Broadly, these are ones in which a system has a piece-wise stable configuration. A given configuration consists of a set of programs, their states, and the assumptions under which they are operating. While the configuration remains "valid", the system provides services in accordance with its specification for such settings. The transition to a new configuration occurs when the assumptions are violated, triggering some form of synchronization that finalizes the current configuration, after which the system can resume normal execution in some new configuration. These steps are illustrated in Figure 2.

The adaptive control problem recalls prior research on distributed application management, for example in Marzullo and Wood's Meta system [Marzullo *et. al.* 91]. However, in distinction to Meta, the concept on which we focus here involves support for shifting the application from one "configuration" to another. Here, the configuration of an application corresponds to the protocol stack it uses for communication, parameters associated with the system or its communication layer, security keys, and the membership of the system. In contrast, management and control technologies such as Meta may structure control rules according to a notion of system configuration, but they

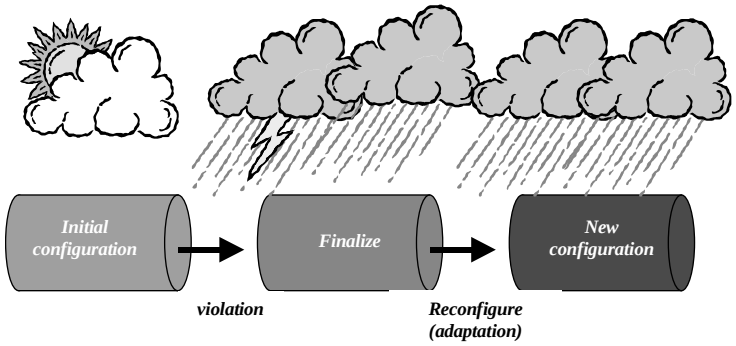

Figure 2: The sequence of events during adaptation

typically treat the application as a monolithic structure that lacks any intrinsic notion of mode of operation. The most common uses of such systems are to automate such tasks as load balancing and restarting failed components.

The structure of the remainder of this paper is as follows. We start with some concrete examples of the sorts of functionality our work is intended to support. The paper then turns to the more technical aspects of our effort. We describe the Ensemble system and discuss aspects of the system that have been important in supporting adaptation, including the "protocol switching protocol", or PSP, a novel feature of Ensemble not seen in previous modular group communication architectures. We also discuss the challenges of optimization and protocol compilation in the light of this adaptive architecture; the problem turns out to be fundamental to achieving good performance. We conclude with a section that investigates overheads associated with adaptation.

EXAMPLE 1: REPLICATION OF CRITICAL APPLICATION FUNCTIONS

Many reliable distributed systems replicate critical data or servers by playing a sequence of updates to each of a set of replicas, while load-balancing query operations [Birman and Van Renesse 96]. If a replica fails, the remaining replicas are able to assume the functions of the failed system component. The communications tool most often used in solving this problem is *ordered, reliable multicast*.

Protocols of this sort can employ any of a number of basic techniques for making sure that a group of processes receive messages in the same order. For the purposes of this section, we focus on two approaches (others are described in [Birman97]). One of these is centralized: a single participant is elected as sequencer. Messages are first sent to the sequencer, and the sequencer forwards the messages to the other participants in FIFO order. The other is distributed, and involves a token that rotates among participants. A participant is only allowed to send when it has the token.

The performance of the replication scheme will be limited by that of the multicast, which is itself limited by the degree of match between the ordering protocol selected and the environment in which it runs. For sequencer-based ordering, the delay to delivery is twice the underlying network latency if the load is sufficiently low. Under high loads (particularly when the number of active senders goes up), the sequencer gets overloaded, its input buffer overflows, and retransmissions are necessary to overcome message loss. A token-based protocol does not have this problem: there is only one sender on the network at a time. However, it has the problem that the delay is approximately half the number of participants times the network latency (the average time to wait for the token).

In Figure 3, we show typical performance curves for such protocols, given some static, fairly large group of participants. It is easy to see how switching between the protocols can optimize performance. The challenge is to detect *when* to switch.

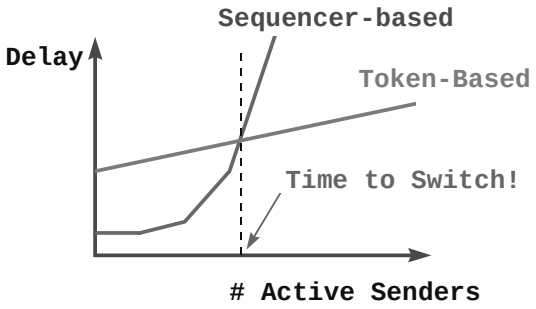

Figure 3 Typical performance curves for two well-known total ordering protocols.

Using our system, this problem is solved using a *violation detector* implemented as a micro-protocol that monitors performance of the total ordering micro-protocol. As discussed shortly, Ensemble has a layered architecture well

matched to such solutions. The detector monitors the message load and, if desired, other aspects of the runtime environment. If conditions arise that represent a potential problem, it reports a violation, triggering a reconfiguration with a new total ordering protocol. The problem of synchronizing the protocol switch with ongoing communication by the application is one of the technical issues addressed by our work.

Although we used just two ordering protocols in this example, the idea is that each of the ordering solutions available in a system supporting replication would have its own monitoring and violations detection layer. The handling of a violation is application specific and may be challenging, but is the only place where all the available ordering options might be considered as a group. While a given ordering method is in use, that method effectively monitors itself. Notice that this method permits an overlap between the configuration spaces acceptable to the various protocols: once a protocol is running, it will continue running so long as its violations detector doesn't sense an unacceptable degree of mismatch between the protocol and its environment.

This can be contrasted with the more general style of application management used in earlier work. In the traditional approach, the management system must *continuously* check the environment against a model that includes all possible ordering protocols, and in which the choice of protocols is unambiguous in any given situation. Moreover, the traditional approach requires a disjoint partitioning of the configuration space, and triggers reconfigurations each time the optimal choice changes. Such an alternative is potentially costly, runs the risk of "jitter" between similar options, and is much more complex to program. The use of a violation detector leads to much simpler code and avoids the overhead of constantly developing a distributed picture of the environment and checking it against a potentially complex model.

EXAMPLE 2: FIREWALL

Our second example works to overcome a limitation of firewalls as they are typically used. Consider an application (such as a groupware server and its clients) that sometimes runs entirely within a firewall, but sometimes spans the firewall.

When the application runs within the confines of the firewall, it is not usually necessary for messages to be signed or encrypted. However, if a participant running outside the firewall is added (see Figure 4) signing and encryption may suddenly be required. Moreover, this can affect all the clients, because many such applications use multicast communication. In the system we have developed, it is trivial to dynamically add signing and encryption layers to the protocol stack used between the server and its clients. In effect, this creates a *Virtual Private Network*.

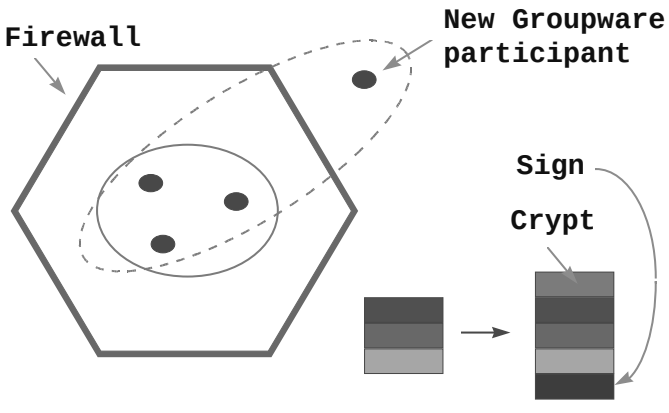

Figure 4 From a secure fire-walled environment to an insecure one.

Secure adaptation is a potentially difficult problem. Care must be taken to ensure that the adaptation protocols do not themselves introduce security breaches. Before the reconfiguration, the new participant must be authenticated, and an

access control list checked to see if it is allowed to join the application. As part of this process, the group signing and encryption keys can be exchanged, and it may be necessary to "rekey" the communication endpoints for the application as a whole if a member leaves and can no longer be trusted. Moreover, the synchronization issue mentioned earlier arises here: clearly, all members of the application need consistent expectations with respect to the encryption policy being used and the keys associated with endpoints. By packaging this complexity within our system, the developer is spared from needing to implement a subtle, special-purpose solution.

EXAMPLE 3: GROUP MEMBERSHIP TRACKING AND COMMUNICATION

Our research effort has been active in an area known as virtually synchronous group communication for many years [Birman 1993, Birman 1997]. In a virtually synchronous group, participants are provided with views—lists of participants believed to be reachable over the network. Participants may leave a group or crash, and new participants may join. Such events, as well as network partitions, lead to view changes in the other participants. The participants try to agree on the sequence of views, and the following property is guaranteed for multicasts within the group: every two participants that install the same two consecutive views, also have to deliver the same set of multicast messages between those two views. The simplicity of the model makes it much easier to keep the states of the participating application programs consistent and to coordinate their actions. Moreover, this runtime model can support a variety of useful tools for programming cooperative or coordinated behaviors.

In turns out that virtual synchrony can be described easily in the terms that we have introduced in this paper. The participants *assume* that views are fixed, and that all messages are delivered everywhere. Detectors are installed to detect violations. A failure detector detects failures, while a stability detector detects that a message has not been delivered everywhere. When the stability detector raises a signal, a process that has delivered the message will retransmit a copy to its view. When the failure detector raises a signal, a reconfiguration is initiated. A configuration is "finalized" by delivering any pending messages for the corresponding view, after which the new configuration (view) can be installed.

THE ENSEMBLE SYSTEM

We now describe the Ensemble system [Hayden 1997], a high-performance, reconfigurable Plug-and-Play network protocol architecture intended for adaptive applications. The basic functionality of Ensemble is to track the membership of groups and to provide communication support among group members, and from non-members to a group. Depending upon the nature of the application, the user may work with Ensemble directly, as a tool supporting data replication, collaboration, or coordination. However, Ensemble can also be used to *control* other technologies, such as the endpoints of standard communication sockets layered over IP-v4 or IP-v6, hidden within a set of pre-build Java applets, or integrated into a scripting language such as Tcl/Tk. In these situations, a conventional network application may be managed or controlled by Ensemble without direct visibility of Ensemble to the developer.

In the work described here, the group consists of endpoints bound to the components of the application in its current configuration. Adaptation is accomplished by changing the component set (and hence, potentially, the membership of the group), as well as by changing the communication infrastructure used by and between these components.

Ensemble's architecture revolves around the notion of a protocol stack. Such a stack is constructed from simple *micro-protocol modules,* which can be stacked and re-stacked in a variety of ways to meet the communication demands of its applications. Ensemble's micro-protocols implement, among other, basic sliding window protocols, fragmentation and re-assembly, flow control, signing and encryption, group membership, and message ordering.

Ensemble has an underlying theoretical framework for composition of layers. This framework not only makes it possible to design a micro-protocol stack for a particular application in a particular environment, but is also instrumental in providing optimal performance. We believe that such a framework is necessary for reconfigurable systems. It is described in detail later in this paper.

As illustrated in the examples, detectors initiate reconfiguration decisions. Each detector is designed to monitor its environment, sensing violations of the assumptions that were made when the current configuration was selected, and provides enough information to come to a new set of assumptions for the design of a new configuration. The detectors may be implemented as micro-protocols themselves, since that puts them in a convenient position to monitor the network traffic. The use of *failure detectors* to make reconfiguration decisions in the face of processor crashes is already well established [Chandra *et al.* 1992]. Ensemble does the same thing in a more general setting.

As much as possible, adaptation is done transparent to the application. The lowest layers try to adapt first. If they cannot respond correctly to a particular change in the environment, they pass on the notification to the layer above it. Eventually, the application may be notified. In that case, the application will have to decide how to reconfigure (see Figure 5). It may chose to adjust its quality of service expectations, adapt its interface or lower the quality of information provided to users, etc.

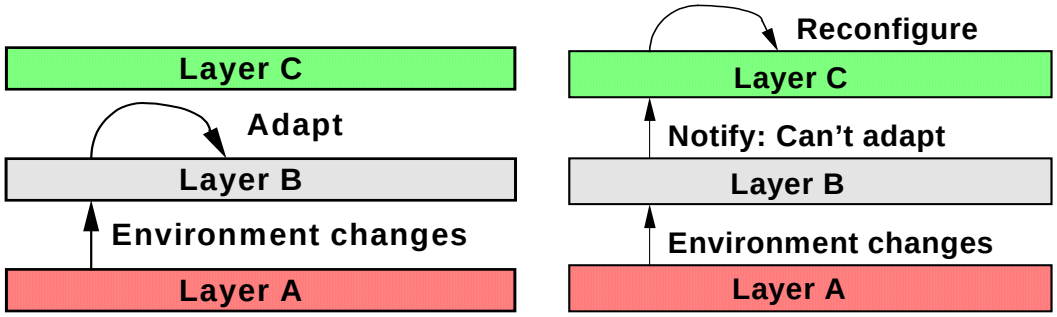

Figure 5: If a layer can't adapt to a change in the environment, it passes the responsibility up. This repeats until, eventually, the application itself has to reconfigure.

A reconfiguration in Ensemble is performed by the *Protocol Switch Protocol* (PSP). PSP is a fault-tolerant protocol that synchronizes the participants, assists them in finalizing their state and performs the necessary agreement before resuming communication. The protocol handles the synchronization issues cited in the examples., Moreover, the PSP includes mechanisms by which participants can dynamically download the code for micro-protocols, caching this code for reuse. With PSP, Ensemble can thus support adaptations that may not have been anticipated at the time the application was first launched. PSP is described in detail in a later section.

Ensemble provides a framework for composing micro-protocol modules into protocol. Each module adheres to the Ensemble micro-protocol interface. There is a top-level and a bottom-level part to this interface. The top-level part of the interface of a module communicates with the bottom-level part of the interface of the module directly on top of it.

Down Events	*Up Events*
Send-Message	Deliver-Message
Leave-Group	Comm-Problem
Request-Timeout	Timeout
Reconfigure	Cannot-Adapt

Table 1: A small selection of common events that are passed up and down the Ensemble stacks.

The interface is event-driven: modules pass events to the adjacent modules.[1] Certain types of events travel down, while others travel up the stack. They are called *down events* and *up events*, respectively. In order to support a wide range of micro-protocols, the set of events is rich: about 40. (See Table 1 for a list of the most important events—the purpose of them should become clear while reading the rest of this paper.). Most micro-protocols pass most events on unseen, and only deal with those events that interest them.

Unlike previous protocol stack models, the application does *not* live on top of the protocol stack (see Figure 6). Instead, it directly interfaces with particular micro-protocol layers. For example, the sliding window layer may provide an interface to change the retransmission timer (there is no event for this purpose). The general-purpose *Appl* layer provides interfaces to send and receive messages. It generates a Send down event in response to its send interface, and intercepts Delivery up events to implement its receive interface. The application may install more than one in a given stack. For example, it may have one underneath the encryption layer to send messages in plain text, and one above the encryption layer to send them securely. Typically, the application will try to use the bottommost *Appl* layer to send and receive messages to minimize latency. The top-level *Cork* layer discards Up events.

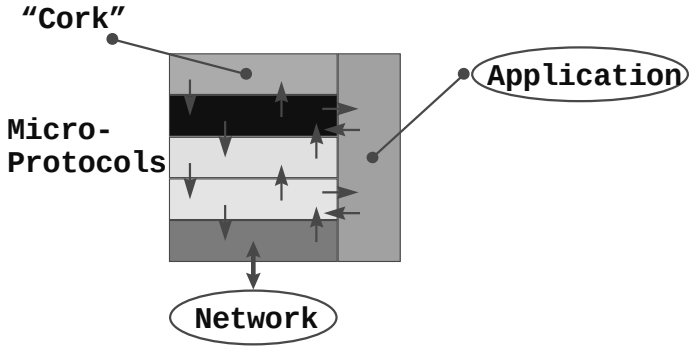

Figure 6: The Ensemble architecture.

Ensemble is written in Objective Caml, a dialect of the functional language ML. When comparing Ensemble to its predecessor, Horus [Van Renesse *et. al.* 1996], which was written in C, we find that code for Ensemble is about 7 times smaller than functionally identical code in Horus. The reasons for this include O'Caml's high-level manipulation of data structures, automated memory allocation and garbage collection, and automatic marshalling. O'Caml code is easier to develop and

[1] This is different from Ensemble's predecessor, Horus, which used procedure calls for downcalls, and thread invocations for upcalls. We find that events are less deadlock prone, easier to trace and debug, more portable, and offer better performance.

maintain, and interfaces well with C and Java. For our work on adaptation, the use of O'Caml has yielded a flexible infrastructure with which a single code base can be adapted for a target environment, and then readapted as conditions change, without requiring that the microprotocols comprising the code base be modified by the developer. Moreover, as will become clear shortly, O'Caml facilitates the use of powerful code manipulation tools, which we have used for code optimization and transformations.

ASSUME-GUARANTEE SPECIFICATIONS

Although, syntactically, micro-protocol modules may be composed in any way, most compositions will not generate a useful protocol stack [Van Renesse *et. al.* 1995]. A protocol stack is useful if it satisfies the communication needs of the application over a particular network. For optimality, we also would like it to be minimal, and not provide costly properties that the application does not require.

For this purpose, we associate one or more *Assume-Guarantee Specifications* (AGS) with each micro-protocol [Karr 1996]. Each AGS specifies that given a particular set of assumptions over the history of its input events, it guarantees something about the history of its generated output events. Examples of such specifications may be found in Table 2.

Layer	*Assumptions*	*Guarantees*
FRAGMENTATION	FIFO-delivery(below)	MaxSend(below, 1.5K)
FIFO	MaxSend(above, 1.5K)	FIFO-delivery(above)
	Eventual-delivery(below)	
ETHERNET	MaxSend(above, 1.5K)	Eventual-delivery(above)

Table 2 Assume/Guarantee Specifications of some layers.

AGSs may be composed in much the same way as micro-protocols. For a well-formed protocol stack, all layers have to satisfy the assumptions that the directly neighboring layers make. In addition, the network has to satisfy the assumptions of the bottommost layer, and the stack as a whole has to satisfy the assumptions of the application.

These conditions may be checked automatically, and this can be done on the Web (see http://www.cs.cornell.edu/Info/Projects/HORUS/hardening/demo1). Also, given the properties that an application requires, and the properties of a network, we can readily find a near-optimal protocol stack using some heuristics, as well as the set of assumption violation detectors that are required for this configuration.

PROTOCOL SWITCH PROTOCOL

When the system detects that its initial assumptions about the environment in which it runs are no longer valid, or the application requires different support than before, it will design a new protocol stack to fit the new requirements. The new protocol stack uses a different message header, so finalizing the current configuration and switching to the new stack requires careful

coordination. The problem is made worse if participants can crash or become disconnected. The protocol that installs a new stack across a set of participants is called the *Protocol Switch Protocol* (PSP). (PSP is actually just a set of micro-protocols that have to be stacked in the application's stack to enable such reconfiguration.)

Each instance of a protocol stack is uniquely identified by a *Protocol Stack Instance Identifier* (PSI-ID). The list of participants is associated with each PSI. All messages sent by a PSI start with the PSI-ID, so the network drivers of the hosts can route messages to the appropriate stacks. The task of PSP is to

1. finalize the micro-protocols in the old stack (not all micro-protocols require this).

2. distribute the new stack and assign a new PSI-ID.

3. start the new protocol stack as soon as possible.

First, PSP elects a coordinator by choosing the participant with the lowest network address (see Figure 7). The coordinator generates a new PSI-ID, and broadcasts a FINALIZE message, which includes a description of the new stack, its membership, and its PSI-ID. The message is delivered to all (reachable) participants including the coordinator itself. Upon receipt, each participant builds the new stack, and registers the PSI-ID so that messages my be delivered to the new stack. Also a FINALIZE event is delivered to the top layer (the "cork") of the old stack.

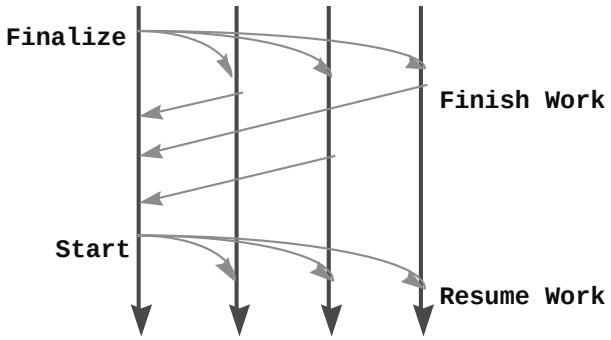

Figure 7: The Protocol Switch Protocol.

Each layer passes the FINALIZE event to the layer below when it is ready to stop sending messages. Before that, the layer may run its own internal finalization protocol. When the FINALIZE event comes out of the bottom layer, a FINALIZE-ACK message is returned to the coordinator. When the coordinator has collected all FINALIZE-ACK messages, it broadcasts a START message to the new stack (using the new PSI-ID). Upon receipt, each participant delivers a START event to the bottom-most layer of the new stack, and the old stack is discarded. The START event is passed up, and eventually discarded by the cork layer.

PSP is fault-tolerant. To recover from message loss, we note that all operations in PSP are idempotent, so a simple retransmission protocol suffices. In case of participant failure or disconnection, the coordinator broadcasts a new FINALIZE message to all participants. The FINALIZE event that is passed down the stack contains the new membership, allowing the micro-protocols to recover from waiting for messages from failed participants. The coordinator only awaits FINALIZE-ACK messages from participants in both the old and new stack. If the coordinator itself fails or becomes disconnected, the participant with the next lowest network address takes over and broadcasts a FINALIZE message. Under no circumstance does PSP wait indefinitely. If necessary, a participant installs a singleton stack and resumes operation.

Note that PSP gives few consistency guarantees. All participants that install the same PSI will only do so once, and they agree on the list of participants. However, not all participants in this

list may install that particular PSI. PSP is not a consensus protocol. Multiple PSIs for the same application may be created in case of a network partition.

It is possible for a micro-protocol to keep track of such properties of the configuration as "contains members at a majority of locations where this application can run", "the last participant to fail" [Skeen85], in order to provide strong consistency guarantees to higher layers or the application. In fact, we have developed several such protocols. For example, a minority participant may refrain from sending messages (even though it could), to prevent creating inconsistencies.

Another micro-protocol, called MERGE, has the task of locating different concurrent PSIs for the same application, and to merge them together. In such a case, PSP is run in all PSIs, but they suggest the same new stack. Participants delay delivery of the START event to their stacks until START messages have been received from all coordinators when MERGE is in use.

OPTIMIZATION STRATEGY

Maintaining good communication performance in a layered system is hard, and a variety of projects have tried to address this problem. For example, the U-Net [Von Eicken *et. al.* 1995] interface to ATM networks provides 35 μsecond one-way message latency for messages of 40 bytes or smaller. A system like Ensemble will add about 80 μs per layer to the latency. Worse yet, the message header size is usually more than 40 bytes, a size at which the U-Net latency jumps to 60 μs.

Thus, it is important to keep header size and processing overhead minimal, at least in the common cases that determine overall performance [Tennenhouse 1990]. However, many layers need to add some information to the message header, and all layers contribute to some processing overhead for every message that is sent or received. We have found that most information in headers seldom changes, allowing for significant compression of headers, typically to just 8 bytes [Van Renesse 1996, Hayden and Van Renesse 1997]. In order to minimize processing overhead is it necessary to generate special code for common cases. We believe this can be done automatically.

A protocol is a function, that takes the state of the protocol (the collected variables that that protocol maintains) and an input event (a user operation, an arriving message, an expiring timer,), and produces an updated state and a list of output events:

$$P(S, E) \rightarrow \quad (S', \{ E_1, E_2, \ldots \})$$

Protocol functions are closed under composition: when two protocols are stacked on top of each other, the result is a new protocol. To compose protocols P_1 and P_2 by stacking them on top of each other, one applies down events to P_1, and up events to P_2. The down events that come out of P_1 are applied to P_2, and the up events that come out of P_2 are applied to P_1, recursively. The up events that come out of P_1 and the down events that come out of P_2 are merged together to form the output events. The state of the composition of P_1 and P_2 is the combined

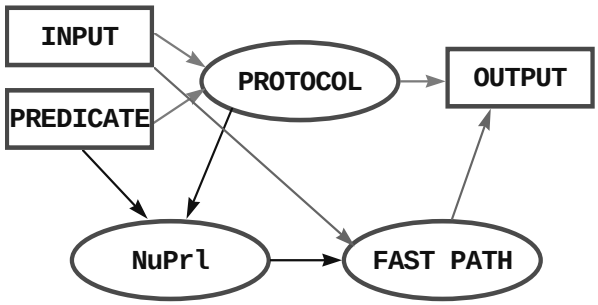

Figure 8: Partial evaluation of a protocol.

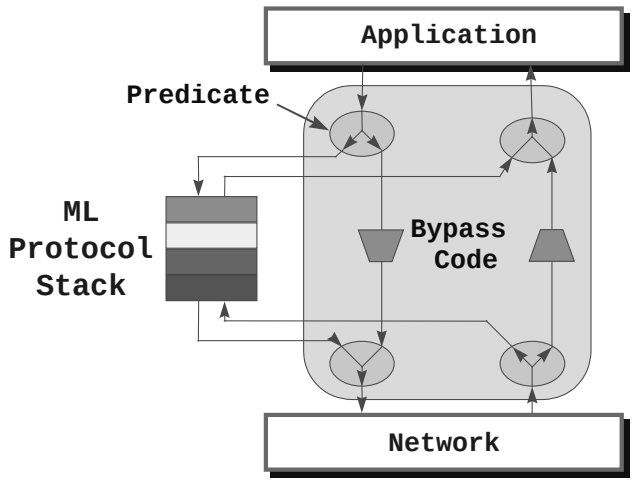

Figure 9 Optimization architecture.

states of P_1 and P_2.

This is significant, because functions may be partially evaluated. Take for example, an *interpreter*: it is a function that takes as input a program, and the input to be applied to that program. Theoretically, a *compiler* partially evaluates the interpreter with the program, to obtain an *executable* that takes just the input to the program. This results in a significant performance improvement when running the program.

The same technique may be applied to a protocol function (see Figure 8). Such a function can be partially evaluated if something is known about an input event and the state of the protocol. We express this knowledge by so-called *common case predicates* (CCP): a Boolean function on the state of a protocol and an input event. For example, a CCP may be true if the event is a Deliver event, and the low end of the receiver's sliding window is equal to the sequence number in the event (in other words, this is the next expected packet to arrive, and it was not lost or reordered). If a message is received for which the CCP predicate is satisfied, that message may be delivered and the low end of the window moved up, without a need for buffering. This technique fits our philosophy of producing modules optimized for expected common cases.

The resulting *bypass* code (see Figure 9) may be O'Caml, but can also be highly optimized C or even assembly. Currently we apply this technique by hand, but we are working with NuPrl [Constable *et. al.* 1986] to automate this. (We are already able to prove that a certain bypass code is correct for a certain stack and CCP using NuPrl.) Like any other micro-protocol, a bypass takes the protocol state and an event as input and produces the updated protocol state and a list of output events. Therefore, multiple bypass code fragments, corresponding to multiple layers, may be composed and in-lined together to produce a single bypass for the whole stack (see Figure 10).

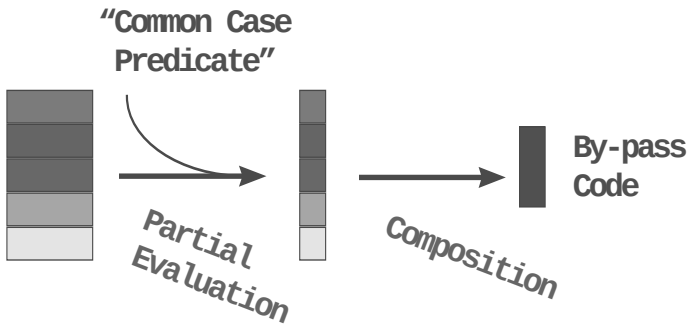

Figure 10 Bypass code fragments from a stack of layers can be composed to obtain a single bypass function guarded by a single common-case predicate.

The same CCPs that are used at compile-time to generate the bypass code, are also used at run-time to decide whether to deliver an event to the bypass, or to the normal stack. The CCP for a composed bypass is the logical disjunction of the individual CCPs for its components. It is necessary to generate efficient code for this CCP, since it will be executed for every event. This approach yields an optimized Ensemble architecture, which is the one actually employed at runtime. Typically, there may be multiple bypasses for a layer, so that many compositions are possible, resulting in many possible bypasses. Over time, we hope to elaborate this technique into one that would allow us to detect common combinations at run-time and, in an application of our adaptation methods, generate the optimized code dynamically and switch to it.

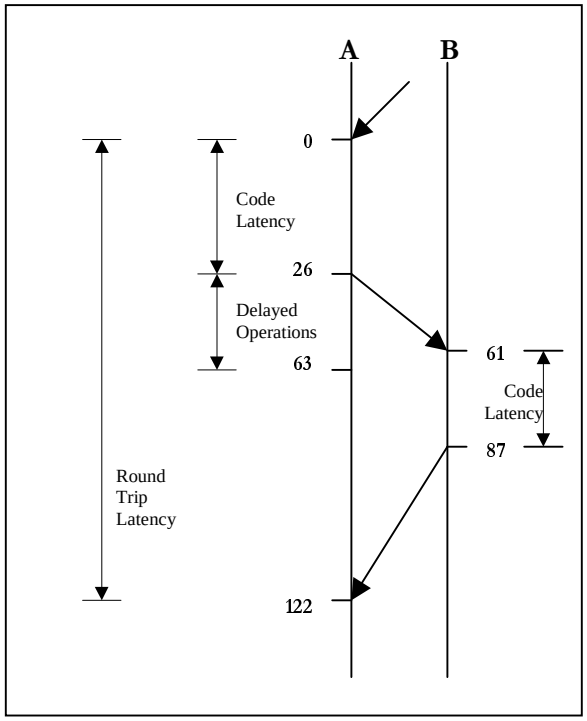

Figure 10: Performance of an optimized stack

Although the optimization architecture of Ensemble has the potential to support some very sophisticated code transformations, we are finding that even limited use of the technique can provide substantial benefits. Thus, while protocol optimization of this sort represents a substantial topic for future investigation, we are already obtaining protocols competitive with hand-coded versions.

Version	Critical Path	Delayed Operations
No bypass	1500	N/A
Bypass in O'Caml	41	28 – 63
Bypass in C	26	37

Table 3: Overhead comparisons.

In Figure 10, we show the round-trip latency in a virtually synchronous group of two processes executing on Sparc 20s running SunOS 4.1.3, and connected by an 140 Mbit/sec ATM network. The network has a 70 µs latency. The figure also shows how much time is spent doing "delayed operations" (updates of the protocol state outside the critical execution path). In this case, the bypass code has been written in C. In Table 3, we compare the overhead of the same stack to not using a bypass at all, or to using one written in O'Caml.

APPLYING ENSEMBLE TO THE EXAMPLE APPLICATIONS

The Ensemble system provides powerful, general solutions to the adaptation problems encountered in a growing class of network applications. We now briefly review the examples raised at the outset of this paper, illustrating the application of Ensemble in each case.

For example 1, where the user sought to match a multicast ordering technique with the properties of the environment, each ordering method is implemented as a separate Ensemble micro-protocol, and is paired with a detector that takes the form of a second micro-protocol, normally stacked above the ordering layer. When the detector senses that the environment has become mismatched with respect to the load requirements of the solution, a violation is triggered, and the application then instructs Ensemble to install some other protocol with better properties for this situation. In this case, finalizing the initial configuration involves delaying the protocol switch until all messages sent with the initial ordering protocol have been delivered.

For example 2, which involves reconfiguring an application so that it can run efficiently both inside and spanning a firewall, the detector check the locations of group members relative to the presence of firewalls and gateways. If encryption is being used from an application running entirely behind a firewall, or is not being used in an application that finds itself outside of the firewall, a violation would be signaled, and the PSP used to reconfigure the stack appropriately. Notice that this change of stack requires synchronization, since the members of an application must be consistent in their use of encryption policy and group key. Moreover, the example requires that the adaptation decision be made in a coordinated, distributed, manner. Again, the finalization stage of the PSP protocol solves this problem, permitting us to ensure that unencrypted communication has terminated before encrypted communication begins. (Alternatively, one could design a solution in which pending messages are retransmitted with encryption as a finalization action, in which case it suffices to ignore incoming unencrypted messages while discarding duplicates upon receipt.)

Unlike example 1, the actions required in example 2 might arise in an application that is not making explicit use of Ensemble. For example, the groupware application could be a conventional client-server system in which the server sometimes interacts with its clients as a group, perhaps by using a multicast to distribute updates to cached data or some other form of shared state. In this case, one would expect that the application operate over a standard communication substrate, such as TCP or UDP over an IP network. We have been successful in using Ensemble in such settings by grafting our solutions to "wrappers" that offer standard network functionality to the application developer, but control communication using policies mediated through Ensemble. The effect of this is to hide the adaptive mechanisms of this paper to a network that appears conventional to the user.

Finally, for example 3, the PSP provides synchronization mechanisms similar to those found at the core of typical virtual synchrony implementations [Birman97]. Ensemble includes a micro-protocol that guarantees the virtually synchronous property by making sure that all messages are delivered to everybody in the current view and the new proposed view. It prevents the PSP finalization protocol from terminating until this is the case by delaying the FINALIZE event.

Similarly, Ensemble implements its failure detector and the stability detector as micro-protocols. The stability detector adds a vector to every message that describes how many messages have been delivered from every participant. The failure detector takes its information from lower layers, like the sliding window layer and the stability detection layer, which report consistent problems. When the rate of problems reported to the failure detector overruns a configuration-defined threshold, the detector signals a violation, which leads to reconfiguration of the set of participants.

CONCLUSIONS

The emergence of a generation of network applications that must adapt to changing environmental conditions and membership poses new challenges to protocol designers. Our work illustrates an approach in which adaptation mechanisms are provided through a small set of core

functions, supported in the context of a more comprehensive architecture for supporting stackable protocols. The resulting system simplifies the design of adaptive applications. Moreover, as illustrated by the third example, the concept of adaptation lies at the core of the Ensemble system itself, in that Ensemble uses its own adaptive mechanisms to implement some of its most fundamental functionality. The design is such that a small set of micro-protocol layers can potentially be reused in a wide range of conditions.

The adaptation problem is a natural generalization of dynamic membership tracking, a function common to most group communication systems. Adaptation includes the possibility that system membership will change, but also permits change to protocol stacks and fundamental parameters of the runtime environment. New protocols can even be downloaded at runtime. The importance of the work presented here is that it offers a clean solution to such problems, in a manner optimized for high performance.

Support for adaptation is not trivial, although the networking community has not always recognized the significance of the problem. We believe that adaptive behaviors are often deferred until late in a networking project precisely because the problems look hard, and in our own experience, retrofitting such behavior can be a serious source of bugs, delays, and last-minute complexity. Good solutions to these problems require modularity, flexibility, and high performance. At the same time, because the synchronization problems that arise during adaptation are potential sources of bugs, it is important to offer a clean and easily understood runtime model. Preservation of system security during adaptation is also a challenge; for brevity we have not addressed this topic here. Our work solves these problems (including the security problem, which we will discuss elsewhere), and also enables the use of formal theorem proving tools to support provably correct protocol optimizations and code transformations. The result is that a single set of small modules can be reconfigured dynamically to match our system to the target environment, and the resulting code can be optimized for high performance.

All work described in this article has been implemented and is available for use by the public. Our software can be downloaded from http://www.cs.cornell.edu/Info/Projects/Ensemble, and is provided with no fees or restrictions upon the user.

ACKNOWLEDGMENTS

We would like to thank Anne and Mike Greene for the use of their computer, and Robert Constable for comments on an earlier draft of this paper.

REFERENCES

[Birman 93] Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. *Commun. of the ACM* Vol. 36, No. 12 (Dec. 1993).

[Birman and Van Renesse 96] Kenneth P. Birman and Robbert van Renesse. Software for Reliable Networks. *Scientific American* 274:5 (May 1996), 64-69.

[Birman 97] Kenneth P. Birman. *Building Secure and Reliable Network Applications.* Manning Publ. Company and Prentice Hall, Jan 1997.

http://www.browsebooks.com/Birman/index.html

[Chandra *et. al.* 1992] Tushar D. Chandra, Vassos Hadzilacos and Sam Toueg. The Weakest Failure Detector for Solving Consensus. In *ACM Symposium on Principles of Distributed Computing* (Aug. 1992). 147—158.

[Constable 1986]. Constable, Robert L. and Stuart F. Allen and H.M. Bromley and W.R. Cleaveland and J.F. Cremer and R.W. Harper and Douglas J. Howe and T.B. Knoblock and N.P. Mendler and P. Panangaden and James T. Sasaki and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*, Prentice-Hall, NJ, 1986

[Hayden 1997]. Mark Hayden. *The Ensemble System*. Ph.D. dissertation. Dept. of Computer Science, Cornell University. Forthcoming, Dec. 1997.

[Hayden and Van Renesse]. Mark Hayden and Robbert van Renesse, Optimizing Layered Communication Protocols, Proceedings of the 6th IEEE High Performance Distributed Computing Conference, Portland, Oregon, Aug 1997.

[Karr 1996]. David A. Karr. *Protocol Composition in Horus*. Ph.D. dissertation. Dept. of Computer Science, Cornell University. Dec. 1996.

[Marzullo *et. al.* 91] Keith Marzullo, Robert Cooper, Mark Wood and Kenneth P. Birman. Tools for Distributed Application Management. *IEEE Computer* 24:8 (August 1991), 42—51.

[Skeen 1985] Dale Skeen. Determining the Last Process to Fail. *ACM Transactions on Computer Systems,* Vol. 3 No. 1 (Feb. 1985), 15-30.

[Tennenhouse 1990]. David Tennenhouse. Intergrated Layer Processing Considered Harmful. In *Protocols for High Speed Networks*, Elsevier, 1990.

[Van Renesse *et. al. 1995]* Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr, A Framework for Protocol Composition in Horus, Proceedings of the 14th ACM Principles of Distributed Computing Conference, Ottawa, Canada, August 1995, 80—89.

[Van Renesse 96] Robbert van Renesse. Masking the Overhead of Protocol Layering. Proceedings of the 1996 ACM SIGCOMM Conference, Stanford, September 1996.

[Van Renesse *et. al.* 96] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A Flexible Group Communications System. *Commun. of the ACM,* Vol. 39, No. 4 (April 1996), 76-83.

[Von Eicken *et. al.* 1995] Thorsten von Eicken and Anindya Basu and Vineet Buch and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th Symposium on Operating Systems Principles* (Copper Mountain, Dec. 1995), 40-53.