A Framework for Protocol Composition in Horus

Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr

Dept. of Computer Science
Cornell University *

Abstract

The Horus system supports a communication architecture that treats protocols as instances of an abstract data type. This approach encourages developers to partition complex protocols into simple microprotocols, each of which is implemented by a protocol layer. Protocol layers can be stacked on top of each other in a variety of ways, at run-time. First, we describe the classes of protocols that can be supported this way. Next, we present the Horus object model that we designed for this technology, and the interface between the layers that makes it all work. We then present an example layer that implements a group membership protocol. Next, we show how, given a set of required properties, an appropriate stack can be constructed. We look at an example stack of protocols, which provides fault-tolerant, totally ordered communication between a group of processes. The work contributes a standard framework for protocol development and experimentation, provides a high performance implementation of the virtual synchrony model, and introduces a methodology for increasing the robustness of the protocol development process.

1 Introduction

Even when hidden, message passing lies at the heart of any distributed system. A tremendous number of message passing interfaces and protocols have been developed both by the practical and theoretical computer science community. Efforts to bring structure to all this development have been only partially successful. Today, this lack of structure impedes the engineering of large, complex distributed systems. For example, a variety of both fault-tolerance and multi-media protocols are readily available. Yet it would be tremendously complex to implement a large fault-tolerant multi-media system. The integration of subsystems that provide different protocols into a working whole requires intimate knowledge of the internals of each subsystem, and considerable creativity to make them interplay.

Here, we adopt a perspective that treats a *protocol* as an abstract data type: a software module with standardized top and bottom interfaces. Above a protocol module are other protocols or applications that issue requests to it. The protocol itself functions by adding headers to messages, or generating new messages of its own, whereby it interacts with the corresponding module on a remote system. The lower interface permits the module to receive incoming messages, together with other sorts of events.

In most systems, this modular structure is obscured. Each subsystem may have its own top and bottom-level interfaces, its own message data structure, and its own methods of scheduling internal and external events. Interconnecting the different interfaces, converting between the different message formats, and running the different schedulers concurrently arise as challenges that the application developer must resolve. Network standardization has focused mainly on the message formats, permitting processes running on different systems to communicate. The seemingly simpler problem of composing subsystems on the same operating system has received much less attention.

The need is for a single system that has one message format, one event scheduler, and a framework allowing *protocol composition*. Composition requires that the top-level and bottom-level interfaces of the protocols be identical for each layer, so they can be stacked on top of each other like LEGO^{tm} blocks (see Figure 1). The protocol interface must be sufficiently strong to support most protocols, containing hooks with which the interface can be extended to add new features. Luckily, work on object-oriented systems has addressed exactly these requirements. If we can specify protocols in terms of objects, then we can use existing object-oriented techniques for composition of these protocols.

The Horus system provides such an object-oriented protocol composition framework. The system supports objects for communication endpoints, groups of communicating endpoints, and messages. It currently includes a library of about thirty different protocols, each providing a particular communication feature. Protocols can be composed in many ways, allowing flexibility and having the additional advantage that an application pays only for properties it uses. Horus can support many applications concurrently, each of which can be configured individually. Horus supports non-Horus subsystems by providing a separate scheduling environment for each subsystem, and a system-call interception technique that traps system calls made by the subsystem. This gives Horus complete control over each subsystem and an inexpensive way to communicate with it.

Horus arose out of our prior work on fault-tolerant process group computing in Isis system [4]. Isis supports a *virtu-*

*This work was supported by ARPA/ONR grant N00014-92-J-1866.

ally synchronous process group communication environment in which software fault-tolerance was applied to a variety of problems. Isis supported process groups with mechanisms for joining a group and obtaining its state, leaving a group (a failed process is automatically dropped from the groups to which it belonged), and communicating with groups using atomic, ordered multicasts. These primitive functions were used to support tools for locking and replicating data, load-balancing, guaranteed execution, primary-backup fault-tolerance, parallel computation, and system control and management. Horus focuses on the core of Isis, implementing a very powerful process group communication architecture which can be used in support of Isis-like tools, embedded into programming languages or parallel computing libraries, or hidden behind standard abstractions such as UNIX sockets.

In this paper we discuss Horus in relatively practical terms, omitting theory that has been explored elsewhere and pursuing new theoretical directions suggested by Horus only in a limited, preliminary fashion. The paper describes a very simple protocol architecture that is still powerful enough to support the most important styles of distributed protocols used in modern systems. It should be stressed that this layered architecture does not imply a high overhead; indeed, the cost of a layer can be as low as just a few instructions at runtime, and a few bytes (or none at all) added to a message. Our experience with Horus bears this out: with reasonable effort one can achieve performance fully comparable to the best existing systems for similar environments [15].

Although much remains to be done, we have also started to develop formal tools for specification of protocol layers, as well building reference implementations of the most critical Horus layers. In this use, a *specification* is a skeletal description of the behavior of a layer, giving the requirements the layer makes on layers above and below it, and the guarantees the layer provides in situations where the requirements hold. A *reference implementation* is an formalized version of a layer, potentially executable, but developed primarily to facilitate the use of formal proof tools for verification. Our specification and reference language is a subset of ML, while the language of preference for highly optimized versions of layers is C or C++. Demanding applications would normally use the more optimized layers, which sometimes combine the functions of several reference layers into a single high performance production version. Our contention is that by providing high level, executable, descriptions of key parts of Horus, the system can be significantly hardened. This approach is discussed in detail in Section 8.

Horus is thus a multi-faceted effort. The project seeks to contribute a powerful and flexible programming environment for distributed application development, focusing on issues of fault-tolerance, consistency and security using process groups and (if desired) virtual synchrony. We do this in a principled and modular manner that facilitates the use of our system to implement protocols with goals different from our own. Moreover, Horus creates a framework within which formal methods can be brought to bear on such problems as protocol specification and verification.

2 Classes of Protocols

The classical example of object-oriented methodology is a window system. Starting with a basic window, one can construct extended windows with concepts like a border, a title bar, or a scroll bar. These form new objects that *inherit* the basic interface from the basic window object. The basic window with its interface and semantics forms a *class*, and each specialization or extension a *subclass*. The concept is generalized by building a tree of classes, the *class hierarchy*.

Protocols also match this model. We can start out with a basic protocol class that supports best-effort byte delivery over ATM or the Internet. With this protocol, messages may be delayed, lost, or garbled. As we layer other protocols over such a layer, its properties can be enhanced. A simple protocol that adds a (large enough) checksum to each message could be used to reduce the garbling problem to a statistically insignificant rate. Such a protocol has functionality on both the sending side, where it adds the checksum, and on the receive side, where it drops the message if the checksum does not match the contents of the message. More interestingly, the checksum could be made cryptographic (*i.e.*, dependent on a secret key), making it impossible for an malignant intruder to impersonate a member process of the application. The corresponding protocol model forms a subclass of simple communication protocols.

Next in the class hierarchy could be a protocol that deals with message loss and reordering. By adding a sequence number to each message, a receiver can detect that messages have been reordered or lost. It can then request a retransmission by returning a *negative acknowledgement* message. This requires the sender to remember each message it sent until it knows that the message has arrived. To give some idea of what other protocols may be needed in a complex distributed system, we provide a list of many of the protocol types used in Horus (see Figure 1).

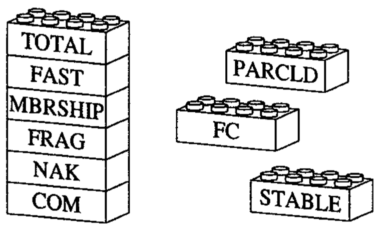
A communication system, such as the Horus embedding of virtually synchronous process-groups into UNIX sockets, is built by stacking a set of protocol modules. The top-most module is the only one to deviate from the Horus interface standard: it converts the Horus protocol abstraction into one matching the needs and expectations of a user. Thus, when Horus is used through its socket interface, the top-most module converts socket *sendto* and *recvfrom* operations into the Horus paradigm.

3 Horus Objects

We noted the need to standardize the abstractions used by protocol modules, as well as their interfaces. Among other objects outside the scope of this paper, Horus provides four classes of objects: endpoints, groups, messages, and threads. None of these objects, not even the group and message objects, are distributed objects. That is, these objects only contain state local to the process (or processor) that owns them. (Of course, they may be used to implement distributed objects.)

The endpoint object models the communicating entity. An endpoint has an address, and can send and receive messages. As we will see later, messages are not addressed to endpoints, but to groups. The endpoint address is used for membership purposes. A process may have multiple endpoints, each with its own stack of protocols.

Although a single layer may be used concurrently by many groups and many endpoints in the same process, each instance has its own state. The *group object* maintains this state on a per-endpoint basis. Associated with each group object is the *group address* to which messages are sent; a *view*, which is a list of endpoint addresses which represent



protocol type	used for
signaling	connection setup
routing	fragments through internet
fragment/assem.	large messages into fragments
compression	to improve bandwidth use
checksumming	garbling detection
signing	safe communication
flow control	preventing network congestion
retransmission	reliable message passing
rpc	client/server interactions
ordering	FIFO, causal, or total
real-time	guaranteed time bounds
membership	agreeing on participants
encryption	private communication
key distribution	security
resource location	in the internet
synchronization	*e.g.*, of clocks
logging	tolerance of total crash failures
tracing	debugging, statistics
management	user administration, etc.
accounting	keeping track of usage

Figure 1: Protocol layers can be stacked at run-time like LEGO^{tm} blocks. The table on the bottom contains a list of common protocol types.

the members of the group; and such additional information as may be needed by the layers stacked by the member that owns the endpoint. Locking mechanisms, described below, protect the group object against concurrent access, for example when threads in an application issue concurrent sends to the same group object. Since a group object is purely local, Horus allows different endpoints to have different views of the same group. Note that we use the term "group" to mean the set of members that communicate using a common group address, whereas the "group object" is a data structure local to each member, and associated with that member's communication endpoint.

The message object is a local storage structure optimized for its purpose. Its interface includes operations to push and pop protocol headers, much like a stack. This should be expected, because headers are added as message objects travel down the protocol stack in the case of sending, and are removed as they travel up in the case of delivery. The message object that is sent is different from the message object that is delivered, although, in most cases, they will contain the same data. A message object can contain pointers to data located in the address space of the application, the operating system, or even a device interface; this permits Horus to pass messages up and down a stack with no copying of the data that the message will actually transport.

All objects discussed so far maintain state only. Horus also provides thread objects, which perform computations. Horus threads are not bound to a particular endpoint, group, or message object, although a thread will often deal with at most one of each. A process typically contains multiple threads, which come into existence in a variety of ways. For example, a thread can be explicitly created by another thread, or may be created by Horus to handle an arriving message or some other event such as timer expiration. Threads execute concurrently and pre-emptively, using mutual exclusion to protect critical regions. Thread priorities are supported, but this raised many problems (starvation, priority inversion) and their use is discouraged.

The threaded architecture of Horus enhances performance (through increased concurrency) and simplicity (through increased code modularity). However, locking is also a source of bugs in layers developed by inexperienced thread users. This has led us to offer two very simple alternatives to standard critical sections. The first of these treats a layer as a monitor, allowing only one thread at a time to be active for each group object. The second is based on *event counters*, and provides a way to order threads according to an integer sequencing value: each upcall is assigned a sequence number, and threads are provided with mutual exclusion zones that will be entered in sequence order.

We have also explored a non-threaded approach based on an event queue model. This model associates queues of invocation parameters with each entry point to a layer. Rather than using a procedure call to invoke a layer, a new event is put on that layer's event queue. Each layer is then implemented with a single scheduling thread per endpoint, which is responsible for selecting (scheduling) an event to dequeue, and then for executing the required code. We find that this leads to much simplified code and reduced storage overhead (the stacks used by threads are much smaller).

4 Common Protocol Interface

For protocols to be stacked in any order, it is necessary that all protocol implementations use *and* supply the same interface. The Horus Common Protocol Interface (HCPI) is designed to be rich enough to support the features of most protocols, and has support for optional extensions. HCPI consists of a set of downcalls and upcalls. The interface provides for multicasting messages, installing views, and reporting error conditions. The HCPI is designed for multiprocessing, and is asynchronous and reentrant. See Tables 1 and 2 for a complete list of upcalls and downcalls. The HCPI allows users considerable flexibility in stacking the layers. Of course, certain protocol layers require that other layers be stacked above or below them, as described in Section 6.

When creating an endpoint, a process describes, at runtime, what stack of protocols it needs, and a base endpoint to build it on. A process is allowed to put multiple endpoints on a single base endpoint. This way, a tree or *cactus stack* of protocols can be built. Given an endpoint and a group address, a process can join a group of endpoints. Eventually, this results in a VIEW upcall which describes the set of endpoints the process can communicate with. In case a *membership* layer is part of the stack, every endpoint in the view is guaranteed to have been sent the same view.

Using the *cast* and *send* interfaces, messages may be broadcast to the view of the group, or to a subset of the view. In case of endpoints joining or crashing, a view needs

downcall	argument	description
endpoint	protocol stack and lower endpoint	create a communication endpoint
join	endpoint and group address	join group and return handle
merge	view contact	merge with other view
merge_denied	merge request	deny merge request
merge_granted	merge request	grant merge request
view	group handle, list of members	install a group view
cast	message	multicast a message
send	message and subset of members	send message to subset
ack	message	acknowledge a message
stable	message	message is stable
leave	group handle	leave group
flush	list of failed members	remove members and flush
flush_ok	group handle	go along with flush
destroy	endpoint	clean up endpoint
focus	identifier	focus on layer and return handle
dump	group handle	dump layer information

Table 1: Horus downcalls

Upcall Type	Information	Description
MERGE_REQUEST	source	request to merge
MERGE_DENIED	why	request denied
FLUSH	list of failed members	view flush started
FLUSH_OK		flush completed
VIEW	list of members	view installation
CAST	message and source	received multicast message
SEND	message and source	received subset message
LEAVE	member id	member leaves
DESTROY		endpoint destroyed
LOST_MESSAGE		message was lost
STABLE	stability matrix	stability update
PROBLEM	member id	communication problem
SYSTEM_ERROR	reason	system error report
EXIT		close down event

Table 2: Horus upcalls

to be *flushed* (see next section). This proceeds in different ways for different layers.

5 Example: A Membership Protocol

The Horus membership protocol, MBRSHIP, shows most of the special features of the Horus Common Protocol Interface. Consider a group of communicating processes. Because of various conditions, not all member processes in the group can communicate with each other at all times. Processes may crash, or the network may partition. Thus a process may not be assured that a message it sends is received by all destination members. Nor can a process be assured that a message it receives is received by other members in the destination set. This introduces a collection of failure scenarios that is difficult to deal with.

The MBRSHIP layer simulates an environment for the members of a group in which members can only fail (they cannot be slow or get disconnected) and messages do not get lost. Each member has a notion of the current view, which is an ordered list of the members. Each member in the current view is guaranteed either to accept that same view, or to be removed from that view. Messages sent in the current view are delivered to the surviving members of the current view, and messages received in the current view are received by all surviving members in the current view. This is called *virtual synchrony*, because all members that can communicate appear to see a failure at the same logical time, significantly reducing the number of failure scenarios.

Virtual synchrony is best understood as a simulation of fail-stop behavior—members excluded from the view may still be alive. When communication is restored, views may be merged using the *merge* downcall. Only if MBRSHIP were used with a perfect failure detector would this simulation be "accurate." MBRSHIP relies only on reliable, FIFO ordering of messages.

At the heart of the MBRSHIP layer is the *flush* protocol. The flush protocol is run when a member crash is detected, or when views merge. One of the members (usually the oldest surviving member of the oldest view) is elected as the *coordinator* of the flush[1] (see Figure 2). The coordinator broadcasts a FLUSH message to the (surviving) members

[1]By picking the oldest group member of the oldest view, this election can be performed without exchange of messages. Notice that the concept of "oldest" might not be meaningful in an execution model where different processes observe group views in different orders or with gaps. In Horus, the virtual synchrony model enables us to make statements like this in a way that is rigorously meaningful.

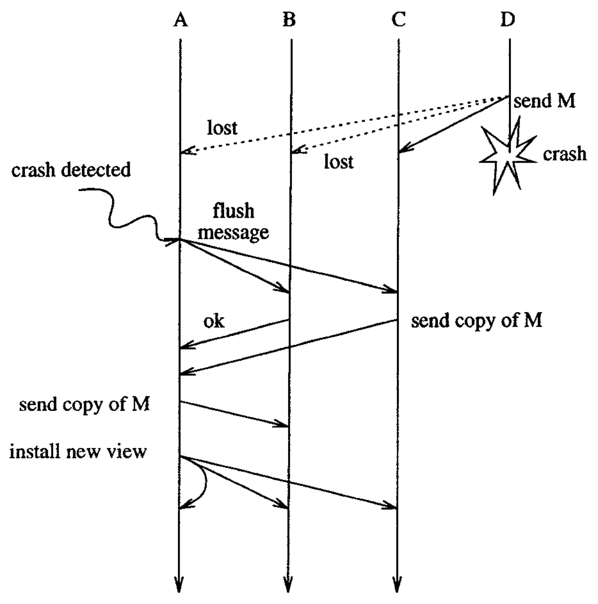


Figure 2: This picture shows four processes: A, B, C, and D. D crashes right after sending a message M, and only C received a copy. After the crash is detected, A starts the flush protocol by multicasting to B and C. C sends a copy of M to A, which forwards it to B. After A has received replies from everyone, it installs a new view by multicasting.

in its view. All members first return any messages from failed members that are not known to have been delivered everywhere. These messages are called *unstable* (note that it is necessary that all members log all unstable messages). Finally, each member returns a FLUSH_OK reply message. Subsequently, the members ignore messages that they may receive from supposedly failed members, and await another VIEW installation.

Upon receiving all FLUSH_OK replies, the coordinator broadcasts any messages from failed members that are still unstable. At this point a new view may be installed. When all messages stabilize, the flush is completed. If processes fail during the process, a new round of the flush protocol may start up immediately.

Although the MBRSHIP layer is able to do its own failure recovery, it allows for *external failure detection.* In this case, an external service picks up communication problem-reports and other failure information, and decides whether a process is to be considered faulty or not. The output of this service can be fed to all instances of the MBRSHIP layer, so that the corresponding groups have the same (consistent) view of the environment.

The MBRSHIP and MERGE layers raise an interesting issue concerning the handling of partitioning failures in Horus. We return to this question below, in Section 9.

6 Protocol Properties and Inheritance

For a given application that is to be installed over a network with a given set of properties, one needs to find a stack of protocols that will provide the properties the application requires in that environment. We need a formal way to describe what a layer requires from the layers above and below it, and what it guarantees in return. A second issue is to create a *reference implementation* of each layer to formally describe the algorithm that implements the layer's specification.

As a step towards this methodology, we have begun compiling sets of properties provided by and required by layers (see Table 4). Table 3 lists, for each of a selected set of protocol layers, which properties it requires, and which it implements. In addition, a layer may or may not pass a property through to the layer above it. We call this inheritance. Given this table, it is possible to figure out if a stack is well-formed, and what properties a well-formed stack provides. A stack is well-formed if, for each layer, all its required properties are guaranteed by the stack underneath it. The properties are either provided by the layer immediately below, or inherited from an even lower layer. Vice versa, given a set of network properties and required properties for an application, it is possible to figure out if a stack exists that can implement the requirements. If we can associate a cost with each of the properties, possibly on a per-layer basis, we can even create a minimal stack. Rather than looking at this as stacking protocols on top of each other, a different interpretation is that Horus actually builds a single protocol for the particular application on the fly.

We note the similarity between this methodology and an approach that is commonly used when developing real-time systems. In a real-time system, an application requests timing properties. The system will try to reserve the necessary resources to guarantee these properties. If successful, the application is started. If not, an error is returned to the user. Horus can generalize this idea: an application requests a set of properties first, and then Horus can figure out if it can guarantee this properties.

We are currently working on designing formal methods, so that on a per layer basis we can verify that given a set of underlying properties, it provides a new set of properties. We are also interested in verifying whether a layer leaves certain properties untouched (inheritance). We discuss our preliminary efforts in this direction in Section 8.

7 Example: A Typical Protocol Stack

In this section we look at a typical stack, namely TOTAL:MBRSHIP:FRAG:NAK:COM:ATM. In this stack, COM provides unreliable communication over a low-level network of choice; ATM was selected in the example. NAK provides FIFO ordering using a sequence number, FRAG provides fragmentation and reassembly of large messages, MBRSHIP provides virtually synchronous communication with respect to group membership, and TOTAL provides totally ordered communication within group memberships. If we know that ATM only provides property P_1 of Table 4, then we can quickly find from Table 3 that this stack results in the properties P_3, P_4, P_6, P_8, P_9, P_{10}, P_{11}, P_{12}, and P_{15}. This section will visit each of these layers in turn and clarify why these properties are obtained.

The COM, NAK, and FRAG layers do not provide consistent views. A view at these layers is nothing but the set of destination endpoints for multicast messages. The COM layer translates the low-level network interface into the Common Protocol Interface. If necessary, COM keeps track of the source of messages (by pushing the address of

Layer	(R)equires																(I)nherits/(P)rovides															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
COM	R																I	I	I	I	I	I	I			P	P	I	I	I		
NFRAG	R									R	R						I	I	I	I	I	I	I			I	I	P				
NAK	R									R	R						I	I	P	P	I	I	I			I	I	I				
NNAK	R									R	R						I	P	I	I	I	I	I			I	I	I				
FRAG			R	R						R	R						I	I	I	I	I	I	I			I	I	P				
MBRSHIP			R	R						R	R	R					I	I	I	I	I	I	I	P		I	P	I		P	P	
BMS			R	R						R	R	R					I	I	I	I	I	I	I		P	I	P	I			P	
VSS			R	R						R	R	R			R		I	I	I	I	I	I	I	P	I	I	I	I			I	I
FLUSH			R	R				R		R	R	R			R		I	I	I	I	I	I	I	I	P	I	I	I			I	I
STABLE			R	R				R	R	R	R	R			R		I	I	I	I	I	I	I	I	I	I	I	I	I	P	I	I
PINWHEEL			R	R				R	R	R	R	R			R		I	I	I	I	I	I	I	I	I	I	I	I	I	P	I	I
TOTAL			R	R				R	R	R	R	R			R		I	I	I		I	P	I	I	I	I	I	I			I	I
ORDER(causal)			R	R				R	R	R	R	R	R		R		I	I	I	I	P	I	I	I	I	I	I	I	I		I	I
ORDER(safe)			R	R				R	R	R	R	R		R	R		I	I	I	I	P	I	P	I	I	I	I	I			I	I
MERGE	R		R	R				R	R	R	R	R			R		I	I	I	I	I	I	I	I	I	I	I	I	I		I	P

Table 3: This table lists, for each of a selection of Horus protocols, the requirements on the communication underneath the protocol, the properties that are *inherited* from that communication, and the properties that are *provided* by the protocol (see Table 4 for the list of properties).

P_1	best effort delivery
P_2	prioritized effort delivery
P_3	FIFO unicast delivery
P_4	FIFO multicast delivery
P_5	causal delivery
P_6	totally ordered delivery
P_7	safe delivery
P_8	virtually semi-synchronous delivery
P_9	virtually synchronous delivery
P_{10}	byte re-ordering detection
P_{11}	source address
P_{12}	large messages
P_{13}	causal timestamps
P_{14}	stability information
P_{15}	consistent views
P_{16}	automatic view merging

Table 4: A list of protocol properties, each of which can either be a requirement on the communication guarantees provided underneath the protocol, or a guarantee that is provided by the protocol itself.

the source endpoint on each outgoing message), and filters out spurious messages from endpoints not in its view.

The NAK layer provides FIFO ordering of messages. For this it pushes a sequence number on each outgoing message, that the receiver can check. If the receiver detects message loss, it sends back a negative acknowledgement (NAK). The NAK layer buffers some messages for retransmission, and will retransmit if the message is still buffered. If not, it will send a place holder that will result in a LOST_MESSAGE event when received. Each endpoint will occasionally multicast its protocol status, so buffered messages may be flushed, and window-based flow control may be implemented. It also allows the detection of failures or disconnections (in case a status update is not received in time).

The FRAG layer provides fragmentation and reassembly of large messages. Typical networks have a limit on the size of messages they can transmit. When a user of the FRAG layer attempts to send a message that is larger than that maximum size, the FRAG layer splits the message into multiple fragments. On each fragment the FRAG layer pushes a boolean value that indicates whether it is the last one or not. The FRAG layer depends on FIFO ordering for reassembly. When the last fragment is received, it delivers the message.

The MBRSHIP layer has been discussed in the previous section. It adds strong semantics to the VIEW upcall, that is, it guarantees that all members in the view that were also in the previous view have delivered the same messages. It relies on the FIFO ordering provided by the NAK layer, and on the FRAG layer for sending large messages.

The TOTAL layer, in turn, relies on virtually synchronous communication. During normal operation, it utilizes a token. A special "oracle" at each member decides who should get the token next. The oracle cannot always make the optimal decision for minimal overhead, but the protocol that the TOTAL layer uses comes close in many cases. In case of a failure, the token may be lost. This, however, is not a problem. During the flush, all members that did not get the token in time send their messages. These messages are not delivered, but buffered. When the new view is installed, each member that remains connected to the system is guaranteed to have all messages from the previous view, and a deterministic order can easily be constructed (*e.g.*, messages are delivered in the order of the rank of the source). Another deterministic rule decides who the first token holder in this view is (*e.g.*, the lowest ranked member), and normal operation can continue.

Interestingly, the TOTAL layer does not require direct interaction with a failure detector. As providing totally ordered communication is equivalent to the consensus problem, this seems contrary to the impossibility proof of [7]. TOTAL works nevertheless, for two reasons. First, the semantics that the TOTAL layer provides are slightly weaker, since it only guarantees timely delivery to the surviving members in the view. Second, failure information is provided by the MBRSHIP layer in the form of view updates.

8 Reference Implementation Effort

As part of our effort to verify properties of the Horus communication system, we are building reference implementations of Horus protocol layers. Reference layers serve as concise specifications of the current "production" layers, but, despite the appearance of pseudo-code, are also executable. The layers are written in the ML programming language, a high level language that supports features useful for communication systems: eg., automatic marshalling and static type checking of messages. ML also has a formal definition making it amenable to analysis, including program verification in theorem proving systems such as Nuprl and PVS. Because ML is typically slower than C, the reference layers will not attain the full performance of the production layers (which are written in C). Although we are not verifying the production layers directly, we want to emphasize that we are discussing the verification of real, executable implementations of distributed protocols. Such verified reference implementations can be used primarily in two ways. First, they can be used in place of C code to save development effort but sacrifice performance. Second, the reference implementations can be translated back to C, yielding production Horus components with a considerably increased level of verification.

Viewed from a broader perspective, our reference implementation of Horus demonstrates a possible methodology for refining and verifying a class of complex but modular communication systems. Such a methodology responds to the difficulties of applying a comparable process to large systems written in C. Our approach is to translate an already existing system to a high level language (such as ML) which is amenable to the refinement and verification needed to "harden" the system. This will rarely require a complete verification of the system, but rather entails a continuing process of proving the complex and uncertain properties of the system, so that the remaining subgoals are more and more obviously true.

Once the reference implementation has been built, refined, and verified, we translate it back to C as a new production implementation to achieve the high performance of the original implementation but now with a much greater confidence in its correctness. In so doing, layers may be combined and other optimizations applied. However, we do not "throw away" the reference implementation when this process is complete because it continues to be useful as documentation and as a specification of the system, as well as a vehicle for further verification and prototyping.

The continued use of production versions of layers is a concession to the performance requirements of demanding applications. However some applications seek the utmost in reliability, at least with respect to properties such as security. For this reason, another important goal for Horus reference layers is that they be interoperable and interchangeable with the production layers. We have constructed an interface between Horus and ML so that reference and production layers can be mixed freely in a protocol stack. Such an interface is possible only because of the HCPI, to which all layers adhere.

Our approach would permit a fully verified reference layer that implements a security protocol to be inserted into a production Horus protocol stack, even though one is implemented in ML and the others are in C. Interchangeability both encourages and enforces a tight coupling between the reference and production implementations of layers. For instance, a production layer should be replaceable in a protocol stack by its corresponding reference layer and vice versa. Interoperability allows us to take advantage of the existing production version and follow an incremental approach so that "hardening" of key components can be tackled first and non-essential pieces left for later. Also, new protocol layers can be rapidly prototyped in ML, tested with a normal Horus protocol stack, and then translated to C if performance is an issue. In addition to protocol layers, distributed applications can also be written in ML using the interface.

Initial experience shows that the goals we have set for a reference implementation of Horus can be reached. We have built reference implementations for several protocol layers. These are considerably cleaner than the current production layers and are generally an order of magnitude smaller in code size. We believe we will be able to completely implement the core of Horus in a few thousand lines of ML (compared to 40-50,000 lines of C) for the purpose of verification. In addition, the implementation of reference layers has led to several improvements in the basic Horus architecture, some of which may result in improved performance when translated back to the production layers.

We have only begun to explore the issues that arise in actually proving that a layer satisfies its specification, and that a set of specifications can be combined to implement a desired property—for example, that layers can be composed, in the formal sense. Our initial work on this problem uses I/O automata (similar to the model expounded by Nancy Lynch *et al.* [6], with certain modifications suitable to the Horus architecture) to model the protocol executed by a Horus layer. Important properties provided by the layer can then be verified by combining this I/O automaton with other I/O automata representing all the layers above and below it. The composition of these automata (itself an automaton) is a closed system, which we augment with additional properties (such as fairness) expressed in simple temporal logic formulas over states and actions. We then prove that, within this system, the layer honors a specified set of user requirements. A similar technique will allow us to verify that desirable properties of a given protocol stack will be preserved by the addition of a new layer, and to help decide when the stacking order of two layers matters.

9 The End-to-End Argument

Several readers of preliminary versions of this paper raised questions about the end-to-end argument and the controversy over causal and total ordering in communication systems (catocs), asking whether our work on Horus sheds new light on these issues [5]. Before we address these issues directly, we should point out that Horus supports everything from best effort delivery to very strong semantics, and users can decide for themselves whether they need causal or total ordering, or not. Moreover, Horus (and several other catocs systems) *does* provide a true end-to-end mechanism in the form of message stability.

A message is called stable if it has been processed by all its surviving destination processes (that is, the processes that are included in the next view). The term "has been processed" is instrumental here. Horus provides a downcall, *horus_ack(m)*, with which the application process informs Horus when it has processed the message m. Eventually, this information propagates back to the sender of the message, and onwards to other receivers of the message. It is reported using a *STABLE* upcall. The upcall contains detailed information about the stability of the messages that a process sent, or received, in the form of a so-called ***stability matrix.*** Depending on the application, a message could be considered stable when it has been displayed to a user, logged to disk, when it is safe to delete, etc.

The stability matrix thus reports a property that is completely defined by the application layer. The "semantics" of stability data are exactly the semantics determined by the downcalls issued by the application to Horus. We see this as an illustration of the end-to-end paradigm as it is used within Horus: the stability layer provides a mechanism that, under control of the application, may have widely varying meaning.

Back to the concerns that were raised in [5]. Briefly, their use of the end-to-end argument has come under scrutiny from researchers, including ourselves, who favor communication systems that guarantee properties such as virtual synchrony [3], or ordering. The argument favoring "properties" is that the complexity of implementing these in the application itself can be daunting, and that, unless properties are standardized throughout a communication framework it will be impractical to extend a system with new applications that depend upon communication properties over time.

One example is an application which is designed to communicate synchronously with a service, but in which replies to the messages being sent are not needed. An application that updates a display maintained by a remote display server matches this model. Provided that the message delivery order and reliability properties are maintained, such an application could gain improved performance by using an asynchronous communication stream. Given an application consisting of a single process, one could simply use a reliable, FIFO protocol such as TCP to communicate with the server. Now, suppose that the application is composed of multiple processes that communicate among themselves— an increasingly common architecture. The FIFO ordering property now generalizes, becoming a requirement for reliable *causally ordered* message delivery [14]. Given a communication subsystem that supports causal order, the benefit of asynchronous communication can be exploited; lacking it, this performance benefit is not available.

In a superficial sense, Horus could be considered as a contribution to either side of the fence. Because Horus is often used as a library, it will often be linked directly to the application. Configured in this manner, one could argue that Horus is consistent with a philosophy in which the end application implements its own properties, as illustrated by the stability example, above.

However, Horus also employs system-wide services, and provides ordering properties and reliability. Viewed as a runtime environment or a sort of distributed operating system for robust application development, Horus takes on a role of a communication layer and associated services guaranteeing a variety of properties.

In this deeper sense, it could be argued that a system like Horus could not be implemented using an approach fully consistent with the end-to-end philosophy. Although the present paper has not focused on protocols, our previous work has discussed the Horus virtual synchrony implementation in considerable detail. One can view systems such as this as having a three-tier structure. The lowest tier simulates a fail-stop environment (consistent membership tracking with accurate notifications when membership changes occur). The second tier closely resembles a state machine, and implements higher level programming abstractions. In the case of Horus, the abstraction of choice is the virtually synchronous process group, with ordered and failure-atomic multicast (although, as we have stressed, one can easily configure Horus to have other properties, and can selectively enable or disable any of these basic properties). Finally, at the third tier, one finds applications that depend on the consistency properties of the underlying structure.

There are at least three different implementations of the first-tier that would be suitable for use in Horus. The Isis system employed a group membership protocol that provides consistent reporting of system membership changes within a primary partition [12, 8]. The Transis and Totem systems implement an extended virtually synchronous addressing model, corresponding to a partitioning model in which the primary partition is distinguished but that also allows progress in non-primary partitions [10]. The Relacs system implements a "quasi-partial" view synchrony model. In this approach, concurrent membership views will either be identical or non-overlapping [1]. Currently, Horus can be configured with an Isis-style of primary partition progress restriction, or to support the extended virtual synchrony model. A new membership layer that uses the view synchrony scheme of Relacs can easily be added.

Elimination of the membership agreement mechanism, on the other hand, introduces the risk of potentially serious inconsistencies. For example, we pointed out in Section 7 that liveness of the TOTAL ordering layer is dependent upon the membership service and that the uniqueness of the ordering token is guaranteed by exploiting consistency in the views supplied by MBRSHIP to that layer. Given inconsistent views, TOTAL might not be live, or it might give different message orderings to different endpoints. Horus is thus flexible about the specific partitioning model used, but inflexible about its need for a close approximation to fail-stop behavior.

This leads us back to the end-to-end dispute. Proponents of the end-to-end argument maintain that each application program, or each client-server pair, should cooperate to maintain the properties needed for their particular purpose. In an end-to-end mindset, none of the partitioning and membership options cited above would be acceptable. Each requires a system-wide consensus mechanism for maintaining membership views, closely integrated into all levels of the communication hierarchy. Yet, in the absence of such consensus, it appears to be impossible to provide consistent behavior at the upper tiers of the hierarchy!

We would argue that the onus falls on the end-to-end community to demonstrate meaningful ways to achieve consistency within their paradigm. For example, it is straightforward to implement replicated data, fault-tolerant synchronization, or high availability of critical servers in Horus. Horus achieves the necessary consistency guarantees through ordering and atomicity properties provided by its process group and communication protocols. These, in turn, depend upon the most basic membership agreement mechanisms. We conjecture that such a dependency structure is necessary, and that in its absence, non-trivial consistency guarantees cannot be provided. If we are correct, this would support the conclusion that end-to-end architectures are inherently less powerful than architectures based on a rigorous system membership service.

10 Performance and Overhead

The extensive use of layering raises important performance issues in Horus. On the one hand, the layering *improves* performance, since applications can choose the minimal stack for their requirements. For example, an application can decide whether or not it needs end-to-end guarantees, and, if so, whether STABLE or PINWHEEL will be optimal. Also, because each layer is small and simple, they can easily and effectively be optimized individually. Although the performance of Horus currently compares very favorably to other systems (see [15]), performance could still be improved. The performance of the current system suffers for the following reasons:

1. There is an indirect procedure call each time a layer boundary is crossed.
2. Since Horus is thread-safe, multiple procedure calls into the same layer often have to be synchronized by a lock. To avoid deadlock, it is sometimes necessary to invoke an upcall as a thread.
3. Layers push their own header onto the message. For convenience, this header is aligned to a word boundary. This leads to a considerable overhead of unused bits on messages that need be transfered. Also, each pop and push operation has an associated overhead.

We have no detailed overhead measurement, but can report that on a Sparc 10 the overhead of the fragmentation/reassembly layer FRAG (which only needs one bit of header space) adds about 50 μsecs to the one-way latency, which is considerable. We believe we could bring this down somewhat by more careful coding, but we are working on more rigorous solutions to each of these problems.

For the first problem, we will avoid unnecessary invocations of a layer, skipping layers that take no action on the way down or up. We also envision that it will be possible to take common substacks of protocols, and (from the reference implementation) create one single production layer. Ideally, a compiler might implement optimizations such as these.

To address the second problem, we are eliminating intra-stack threading, having discovered that concurrency within a stack does not lead to significant gains. This way we can reduce the use of locks and the frequency of thread creation, except when entering a stack from the top or bottom. Since synchronization between stacks is seldom necessary, we can still run each stack within its own thread.

For the last problem, we are changing the protocol implementations. A protocol will specify, instead of the layout of their header, the fields that it needs (in terms of size and alignment, both specified in bits). When building a stack, Horus will precompute a single header in which the necessary fields are compacted. This should reduce wasted space on a message to a minimum, and eliminate the header push and pop operations currently used by most layers.

11 Status and Challenges

The Horus system is fully operational, although we are continuing to extend it with new kinds of protocol stacks. The current system can be accessed through a variety of user-level interfaces. In addition to the HCPI, Horus can present a process group through a standard UNIX sockets interface (e.g. a UNIX *sendto* operation will be mapped to a multicast, and a *recvfrom* will receive the next incoming message). A similar approach could be used to hide Horus beneath a file system interface, much as in the operating system called Plan/9. Horus has also been embedded into object oriented languages, such as ORCA and the Electra version of C++.

Horus stacks exist to support the virtual synchrony model, as well as weaker, less ordered, or less reliable communication models. Horus can thus emulate our older Isis Toolkit, but can also be presented through interfaces matching those of the Hebrew University's Transis system. Very lightweight protocol stacks permit Horus users to obtain the performance of an ATM network with almost no overhead at all. A security architecture for Horus provides for authentication and encryption of messages, using a novel approach that combines security features with fault-tolerance. Looking to the future, we will be adding protocol layers to support guarantees of throughput and low latency, which require resource allocation and scheduling mechanisms within the system. Coordination of behaviors between stacks, in systems that use several stacks simultaneously, has emerged as a topic for future study.

Finally, as noted earlier, the Horus architecture promotes the decomposition of protocols into independent layered modules with clear structure and standard interfaces. For example, in the past, our work on Isis was clouded by an architecture in which protocols for group communication were "mixed" with protocols for membership agreement. In Horus, the system membership service is supported as a layer which uses potentially inaccurate failure suspicions as well as member join (actually, view merge) and leave events to create the abstraction of a fail-stop environment.

Thus, the membership layer sees two kinds of inputs: inaccurate failure and merge events. Its output are failure and merge events that have been filtered by a membership appropriate agreement protocol—a dynamically sequenced uniform agreement in our case, although other protocols could also be used here. A protocol operating over this layer will also see merge and leave input events, but these membership change events will be indistinguishable from fail-stop events. Such an approach is not just easier to implement or extend, but also to understand and reason about. The modular framework thus encourages a theoretical perspective in which it can be made precise what the semantics of a composed set of protocols and a failure detector are, layer by layer.

This leads to another major challenge for future work. Notice that the membership services discussed above each implement a complex protocol that converts merge and failure events with weak semantics to merge and failure events with much stronger (simpler) semantics. This process of filtering a complex environment to create a simpler one is not what one would intuitively expect from the composition of protocol layers. After all, composition should in the general case yield layers with semantics much *more* complex than those of any of the constituent layers. We believe that the issue of how composition leads to simplicity, in a formal sense, emerges as an extremely interesting opportunity for future study. We would argue that, in showing how complex protocols can be simplified using modular techniques, Horus takes a very positive step in the desired direction.

12 Related Work

We are not the first to realize that a framework for communication protocols is necessary. The best-known framework for composing a set of protocols is the STREAMS framework [13]. In this approach, the protocols are lined up linearly, and two reliable, sequenced channels are placed between each pair of consecutive protocols. One of these channels is for transporting user data, while the other channel carries protocol control messages. STREAMS, however, does not support group communication and has limited opportunities for concurrency. A related but more sophisticated approach is used in the *x*-kernel [11]. In this system, protocol objects can be linked together in acyclic graphs. Horus was motivated by ideas from *x*-kernel, but with group communication as the fundamental abstraction. *x*-kernel was mainly designed for point-to-point communication, and even simple request-response style communication is not always easy to map down to this interface. Also, in the x-kernel, configuration is done at compile-time, not at run-time.

Horus improves on this work by providing full thread-safety, and supporting messages that may span multiple address spaces. Since Horus does not provide control operations, and has one single address format, layers can be mixed and matched. In both STREAMS and the x-kernel, the different protocol modules supply many different control operations, and design their own address format, both severely limiting such configuration flexibility. We note that a follow-on to the *x*-kernel project, called Consul [9], is attempting to deal with some of these disadvantages by supporting sophisticated micro-protocols between protocol modules.

Our work with ML parallels the FOX project [2], which is investigating the implementation of system services such as TCP/IP over Standard ML. While we use ML as a tool for implementing prototypes and towards verification of our protocols, FOX takes it a step further by building production systems this way. Their research involves overcoming the performance problems that are incurred because of this approach, while we are interested in the protocols themselves.

13 Conclusion

The development of critical reliability distributed systems has emerged as an important challenge, and demands new tools for distributed software development. The modular, layered architecture of Horus encourages simplicity and rigor in the development process. At the same time, applications pay only for protocol properties they need, leading to extremely high performance and flexibility.

Acknowledgements

Many people have been involved with this work, a list too long to include here. We particularly thank Dalia Malki, Robert Cooper and Barry Gleeson.

References

[1] Ö. Babaoğlu, R. Davoli, L. A. Giachini, and M. G. Baker. Relacs: A communication infrastructure for constructing reliable applications in large-scale distributed systems. In *Proc. of the 28th Hawaii Int. Conf. on System Sciences*, pages 612–621. IEEE, January 1995.

[2] Edoardo Biagioni. A structured TCP in Standard ML. Technical Report CMU-CS-FOX-94-05, Carnegie Mellon University, Pittsburgh, PA, 1994. Also appeared in SIGCOMM '94.

[3] Kenneth P. Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *Operating Systems Review*, 28(1):11–21, January 1994.

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

[5] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communications. In *Proc. of the Fourteenth ACM Symp. on Operating Systems Principles*, Asheville, NC, December 1993. An earlier version appeared as Stanford CS Research Report STAN-CS-93-1485, Sept. 1993.

[6] Alan Fekete. Formal models of communication services: A case study. *Computer*, 26(8):37–47, August 1993.

[7] Michael J. Fischer, Nancy A. Lynch, and Michael S. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[8] Dalia Malki, Ken Birman, Andre Schiper, and Aleta Ricciardi. Uniform Actions in Asynchronous Distributed Systems. In *Proc. of the Fourteenth ACM Symp. on Principles of Distributed Computing*, San Diego, CA, August 1994. ACM SIGOPS-SIGACT.

[9] Shivakan Mishra, Larry L. Peterson, and Richard D. Schlichting. Experience with modularity in Consul. *Software—Practice and Experience*, 23(10):1050–1075, October 1993.

[10] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proc. of the Fourteenth Int. Conf. on Distributed Computing Systems*, pages 56–65, Poznan, Poland, June 1994. IEEE.

[11] Larry L. Peterson, Norm Hutchinson, Sean O'Malley, and Mark Abbott. RPC in the x-Kernel: Evaluating new design techniques. In *Proc. of the Twelfth ACM Symp. on Operating Systems Principles*, pages 91–101, Litchfield Park, AZ, November 1989.

[12] Aleta Ricciardi, Andre Schiper, and Kenneth P. Birman. Understanding partitions and the "no partition" assumption. In *Proc. of the Fourth IEEE Workshop on Future Trends of Distributed Systems*, Lisboa, Portugal, September 1993.

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

[14] Robbert van Renesse. Causal controversy at Le Mont St.-Michel. *Operating Systems Review*, 27(2):44–53, April 1993.

[15] Robbert van Renesse, Takako M. Hickey, and Kenneth P. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report 94-1442, Cornell University, Dept. of Computer Science, August 1994.