Protocol Composition in Horus

Robbert van Renesse
rvr@cs.cornell.edu

Kenneth P. Birman
ken@cs.cornell.edu

Dept. of Computer Science
Cornell University *

March 29, 1995

Abstract

Horus is a communication architecture that treats a protocol as an abstract data type. Protocol layers can be stacked on top of each other in a variety of ways, at run-time. This paper starts out with describing the many classes of protocols that can be supported this way. Next, we describe 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 which implements a group membership protocol. Then, we look at a example stack of protocols, which provides fault-tolerant, totally ordered communication between a group of processes. We conclude with presenting some remaining challenges in our project.

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 most large, complex distributed systems. For example, building a large fault-tolerant multi-media system is tremendously complex, even though a variety of both fault-tolerance and multi-media protocols are readily available. The composition of subsystems that provide different protocols requires intimate knowledge of the internals of each subsystem, and considerable creativity to make them work together.

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

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

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 non-trivial challenges that the application developer must resolve. Network standardization has mostly focused on the network message formats, permitting processes running on different operating systems to communicate. However, 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, so they can be stacked on top of each other like Lego^{tm} blocks (see Figure 1). This, in turn, requires a protocol interface sufficiently strong for 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 [5] 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 twenty 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 only pays for properties it uses. Horus can support many applications concurrently, each of which can be configured individually if necessary. In addition, Horus supports non-Horus subsystems by providing a separate scheduling environment for each subsystem, and a special system-call interception technique that traps system calls made by the subsystem, giving Horus complete control over the subsystem and an inexpensive way to communicate with it.

In this paper we focus on Horus as a practical tool. Although built with the idea of protocol composition in mind, we have not yet developed the theoretical tools needed to support this in a formal sense. Instead, we developed the simplest possible 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. 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.

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 nothing but best-effort byte delivery over hardware such as an ATM network 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. This is one way of *signing* of messages; 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 on 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 any messages 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 protocols used in Horus (see figure 1).

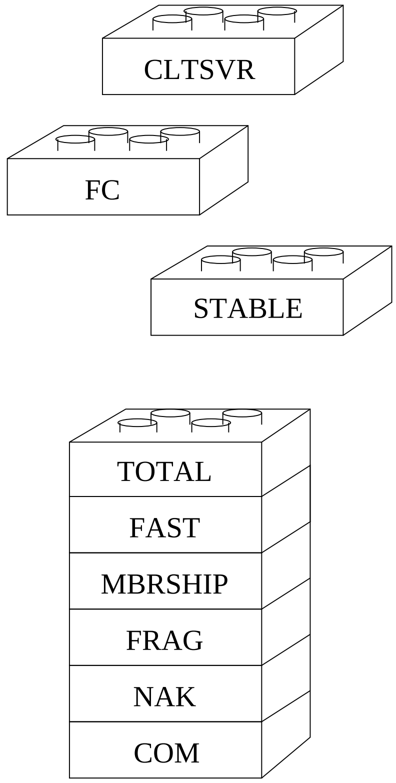
A communication system, such as the Horus embedding of virtually synchronous process-groups into UNIX sockets, is built by stacking a set of protocol modules. The top-most module is the only one to deviate from the Horus interface standard: it converts the Horus protocol abstraction into one matching 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 message-based paradigm.

3 Horus Objects

We noted the need to standardize the abstractions used by protocol modules, as well as their interfaces. Horus provides (among others outside the scope of this paper) four types 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. However, 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 stack of protocols has its own endpoint in a process.

The group object maintains the local protocol state on an endpoint (perhaps *protocol object* would have been a better name). Associated with each group object is the *group address* to which messages are sent, and a *view*: a list of endpoint addresses which represent the members of the group. Since a group object is purely local, Horus allows different



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 participance
encryption	private communication
key distribution	security
resource location	in the internetwork
synchronization	*e.g.*, of clocks
logging	tolerance of "total" crash failures
tracing	debugging, statistics
management	user administration, etc.
accounting	keeping track of usage
application-specific	*e.g.*, mail, ftp, www, NFS, ...

Figure 1: Protocol layers can be stacked at run-time like Lego[tm] blocks. The table on the right contains a list of common protocol types.

endpoints to have different views on the same group. Note that we use the term *group* to mean the set of members that communicate using a common group address, and not the *group object* that each member endpoint maintains.

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 is not surprising, as message objects travel down (in case of sending) or up (in case of delivery) the protocol stack. Message objects are data structures, which are different from the packets that travel on the network proper. Therefore, 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.

All objects discussed so far maintain state only. Horus also provides thread objects, which perform computations. Horus threads are not bound in any way to a particular endpoint, group, or message object, although a thread will often deal with 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 may block using one of the several interfaces that Horus provides for critical regions, or by calling

downcall	argument	description
endpoint	protocol stack, lower endpoint	create a comm. endpoint
join	endpoint + group address	join group and return handle
join_denied	join request	deny join request
join_granted	join request	grant join request
view	group handle, list of members	install a group view
merge	view contact	merge with other view
cast	message	multicast a message
send	message + 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, return handle
dump	group handle	dump layer information

Table 1: Horus downcalls

a blocking system kernel interface. Threads execute concurrently and pre-emptively, using mutual exclusion to protect critical regions. Thread priorities are also supported.

The threaded architecture of Horus proves beneficial from both the standpoint of performance (through increased concurrency) and simplicity (increased code modularity). The message object approach is similarly valuable, providing a "scatter-gather" representation that is extremely inexpensive, minimizes the need to copy data when a message is sent or received, and well matched to the capabilities of existing communication devices, which often implement scatter-gather in hardware.

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, among others, multicasting messages, installing views, and reporting error conditions. The HCPI is designed for multiprocessing, and is completely 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 below them, but we have tried to minimize such dependencies.

When creating an endpoint, a process describes, at run-time, 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 *cactus stack* of protocols can be built. Given an

Event 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 + source	received multicast message
SEND	message + 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

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

5 Example: A Membership Protocol

An example that shows most of the special features of the Horus Common Protocol Interface is a Horus membership protocol called MBRSHIP. 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 all or any 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 to either 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

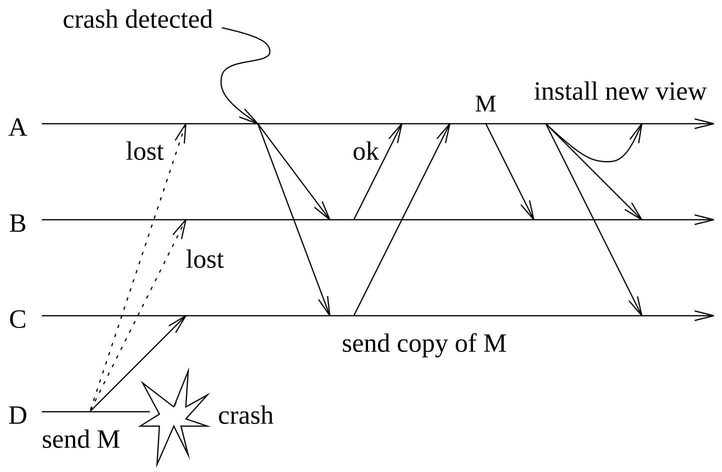


Figure 2: This picture depicts four processes: A, B, C, and D. D crashes right after sending a message M, and only C received a copy. After the crash is detected, A starts the flush protocol by multicasting to B and C. C sends a copy of M to A, which forwards it to B. After A has received replies from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

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. This significantly reduces 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, in which case a MERGE event will later occur when communication is restored. Only if MBRSHIP is used with a perfect failure detector would this simulation become "accurate."

MBRSHIP relies only on FIFO ordering of messages, that is, if some sender sends two messages m_1 and m_2, then any receiver that delivers m_1 and m_2 has to deliver m_1 first.

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. The protocol works as follows: one of the members (usually the oldest surviving member of the oldest view) is elected[1] as the *coordinator* of the flush (see figure 2). The coordinator broadcasts a FLUSH message to the (surviving) members 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

[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.

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.

6 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.

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 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 provided (*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 [1]. There are two reasons for this. First, the semantics that the TOTAL layer provides are slightly weaker, since it only guarantees timely delivery to the surviving members in the view. The second reason is that, indirectly, failure information is provided by the MBRSHIP layer in the form of view updates.

7 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 clear protocol structure. 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 and leave events to create the abstraction of a fail-stop environment.

Thus, the membership layer sees two kinds of inputs: inaccurate failure and join events. Its output are failure and join 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 join 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: the development of formal tools for describing layers and reasoning about the composition of layers. Notice in particular that the Horus membership layer implements a complex protocol that converts join and failure events with weak semantics to join 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. Thus, the whole issue of how composition leads to simplicity, in a formal sense, emerges as an extremely interesting opportunity for future study. A related problem arises in groups where the stacks used by some members differ from those used in others (e.g. if some members can send messages but others only receive). Here, checking for stack compatibility arises as a problem. The apparent similarity to type checking merits further study.

8 Related Work

We are not the first to realize that a framework for communication protocols is necessary. The most well-known framework for composing a set of protocols is the STREAMS framework [4]. In this approach, the protocols are lined up linearly, and two reliable, sequenced channels are placed between each pair of consecutive protocols. 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 [3]. 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. We note that a follow-on to the *x*-kernel project, called Consul [2], is attempting to deal with this disadvantage by supporting sophisticated micro-protocols between protocol modules.

9 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] 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.

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

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

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

[5] 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.