Design and Performance of Horus: A Lightweight Group Communications System

Robbert van Renesse
Takako M. Hickey
Kenneth P. Birman*

TR 94-1442
August 1994

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

---

*The work reported was supported by ARPA/ONR grants N00014-92-J-1866 and N00014-91-J-1219, the NSF grant CCR-9003440, the DARPA/NSF grant CCR-9014363, the NASA/DARPA grant NAG-2-893, and the AFOSR grant F49620-94-1-0198. E-mail addresses for authors: rvr@cs.cornell.edu, takako@cs.cornell.edu, ken@cs.cornell.edu. The authors are just a subset of the full membership in the Horus project; other participants are cited in the acknowledgements.

Design and Performance of Horus: A Lightweight Group Communications System

Robbert van Renesse, Takako M. Hickey, and Kenneth P. Birman*

Dept. of Computer Science
Cornell University

Abstract

The Horus project seeks to develop a communication system addressing the requirements of a wide variety of distributed applications. Horus implements the *group communications* model providing (among others) unreliable or reliable FIFO, causal, or total group multicasts. It is extensively layered and highly reconfigurable allowing applications to only pay for services they use. This architecture enables groups with different communication needs to coexist in a single system. The approach permits experimentation with new communication properties and incremental extension of the system, and enables us to support a variety of application-oriented interfaces. Our initial experiments show good performance.

1 Introduction

In the last several years, we have seen a growing use of group communication primitives in distributed and/or parallel applications. Physically distributed systems (such as stock markets and factories) employ group communications to disseminate information to large numbers of clients. Parallel systems use group communications to allocate jobs among slave worker processes and to exchange intermediate results. Fault-tolerant systems use group communications to propagate updates to replicas and to collect vote quorums for distributed decision making. Standards for group communication are under study in the X/Open and IEEE communities, and a group-oriented parallel communication standard (MPI) was introduced in 1993.

Not surprisingly, many distributed operating systems have offered group communication mechanisms in addition to more conventional RPC and message streams mechanisms

*The work reported was supported by ARPA/ONR grants N00014-92-J-1866 and N00014-91-J-1219, the NSF grant CCR-9003440, the DARPA/NSF grant CCR-9014363, the NASA/DARPA grant NAG-2-893, and the AFOSR grant F49620-94-1-0198. E-mail addresses for authors: rvr@cs.cornell.edu, takako@cs.cornell.edu, ken@cs.cornell.edu. The authors are just a subset of the full membership in the Horus project; other participants are cited in the acknowledgements.

[4, 6, 13]. Unfortunately, with the exception of Amoeba [6], these systems provide limited support for reliability and consistency which are crucial to long running or life critical applications. (Amoeba has a different disadvantage of lack of portability to other vendor-developed systems.) At the same time, however, these distributed operating systems have demonstrated benefits of microkernel software architectures. The principle of microkernel architectures is to provide a small number of basic primitives at a low (kernel) level and to leave sophisticated services to higher levels (in user space). The microkernel approach results in flexible and extensible system functionality, such as customizable memory management and multiprocessor scheduling.

Although group protocols can be implemented in user space, as was done in Isis [3], such a configuration results in suboptimal performance. There are three reasons for this. First, the communication systems implemented at the non-privileged user level cannot always exploit multicast primitive available at underlying network level (such as Ethernet, FDDI, and some ATM switches). Second, many operating systems implement resource management and communication buffering policies that were not designed for group communication protocols and perform poorly when used by such protocols. Third, having a system at in user space results in more context switches and cross-address space references (see [15]).

The challenge, then, is to develop a group communication system that can run either in user space or in a microkernel. To accomplish this goal, a group communication system must minimize complexity, but still maximize both performance and flexibility. This is not straightforward: protocols to support large numbers of groups, dynamic group membership, message ordering, synchronization and failure handling can be complex.

We met this challenge by the design of a portable group communication subsystem called *Horus*. Horus has few system dependencies, and can be incorporated in modern distributed operating systems as either a user-level service or kernel-level subsystem, or both. Compared to its parent system Isis, Horus is smaller, provides more flexibility, and performs better. It also offers security features, and is able to deal with network partitioning. The system design integrates ideas developed in Isis, Transis [2], and the x-kernel [9].

This paper discusses the architecture and implementation of Horus, reviews the interfaces supported (notably, an interface in which the cost of the protocols supporting a communication group can be varied depending on the properties desired by the user), and presents performance figures for a version of the system running in user-space over UNIX^{tm}.

2 The Horus Group Model

A *group* is an addressing abstraction used to refer to a collection of group members. A group member is a communications endpoint which can originate and deliver messages. Processes (or more precisely, communications endpoints owned by processes) can join or be added to a group, leave a group, or be dropped from a group because of failure. These operations cause the membership of the group to change.

The *view* of the group is a snapshot of the group membership at a specified point in the execution of a process. As execution proceeds, a member will see group membership changes as a succession of views. Views are reported to the group members concurrently and asynchronously. Thus, at any instant in real time the group members could have different views of the group. The Horus protocols attempt to deliver the same sequence of views to each member, and, if successful, guarantee that each member see the same set of messages between views. Horus can thus implement a variety of process-group execution models, including the *virtual synchrony* model first introduced by the Isis toolkit. This model was also adopted by [2] and [16].

Horus can be configured to allow progress during transient failure and network partitions, using a variation of protocols proposed by Transis [1]. When a network partition failure occurs, a single group may split into multiple subgroups: one primary and others non-primary subgroups. Group members in different subgroups will then observe different sequences of views. When the partition is repaired a non-primary subgroup can heal itself by merging with the primary subgroup. This contrasts with Isis which only allows the primary partition to continue execution.

The primary partition is usually the majority partition, and is typically defined at the machine level and not at member or process levels. "Primaryness" is detected for sets of machines and all the groups in a given partition inherit the primaryness attribute of that partition. Horus tracks primaryness and reports the value to members through a *primary bit* associated with the group view. If an application is programmed to shut down whenever a group view is delivered with the primary bit clear, the behavior is as that of Isis.

On the other hand, if an application wishes to tolerate partition failures, it can continue execution in groups for which the bit is cleared. Horus continuously seeks out and attempts to merge partitions. When partition merge occurs, there are three possible cases:

- The primary partition may be merged with a non-primary partition, resulting in a larger primary partition.
- Two (or more) non-primary partitions may be merged to obtain a larger non-primary partition.
- Upon merging non-primary partitions, Horus may be able to deduce that the merged partitions create a new primary partition.

Each of these types of events are reported to group members through *partition merge* events, and members of each group then run an algorithm to compute a merged group state. The partition merge protocol is also used for joining new processes to the group (an idea borrowed from Transis [2]).

Communication to a group is by *group multicast.* In the absence of failure, a group multicast is delivered to all group members in the view of the sender. When failures occur, a modification of this rule applies: if a message is delivered to one *reachable* member, it will be delivered to all *reachable* members. Specifically, when a process has problems communicating with another process in its view, Horus will attempt to install a new view excluding this member. Horus synchronizes with the other reachable members in the view so that all these members install the same new view. Horus guarantees that if two processes were in one view, and agree on installing a new second view, that those two processes will deliver exactly the same set of messages. This is a type of message atomicity called *virtually synchronous group addressing* in the Isis model. When Horus is configured to support network partitioning, the execution model that results is the *extended virtual synchrony model* [7, 8].

3 Design

Horus implements the group model discussed in the previous section in an extensively layered and highly reconfigurable manner. This design allows applications to pay only for those aspects of the group model they need. Much of Horus design has been inspired by concepts from various modern systems such as microkernel operating systems, the x-kernel, and object-oriented systems.

Microkernel operating systems support a limited number of basic primitives at a kernel level and more sophisticated services on higher levels allowing flexibility. In Horus, a small number of basic primitives have been identified and provided by its microkernel, called −−MUTS. We will describe −−MUTS briefly in section 5.

Our overall approach resembles the x-kernel, which is a framework for implementing network protocols. In the x-kernel, each protocol implements a simple feature, and the protocols can be tied together in a graph to support the needs of the application. Horus adopts this idea, but its interface is more suitable to multicast protocols. Horus can be integrated into the x-kernel, if desired, in which case it is best viewed as an extention of x-kernel specialized for the case of process groups and group multicast, but can also be run inside of, or over, other operating systems.

In object-oriented systems, sophisticated objects are derived from basic objects. In Horus, a simple "basic group" can be extended with features such as message ordering or flow control. The basic group does not provide virtually synchronous views, or even reliable message passing. Instead, each basic group has an identifier and a current membership view. Each group member can have its own view of the group, and is responsible for maintaining and installing new membership views, for example when other members join, leave or fail. In the basic group multicast protocol, messages are delivered on a best effort basis. This type of basic group is supported transparently over a wide variety of different networks, such as Ethernet and ATM, and is optimized for maximum performance.

Over the basic group there are some fifteen features that can be selectively added to change the semantics of group view reporting and group communication. Each feature is coded as a light-weight software layer that can be added dynamically (*i.e.,* at run-time). Normally, one sets up a set of layers for a given group and then leaves them unchanged, and all the group members use the same layers. (To a limited degree, features can be selectively enabled or disabled on a per-multicast basis, but this requires special knowledge of how the layers work.) The intent is that, by stacking a particular set of layers, a group can be tailored to the needs of the application (see figure 1). This flexibility can then be hidden behind simple user-oriented interfaces that require no special knowledge of how Horus is really configured.

At the time of this writing, the major Horus layers implement FIFO message passing, fragmentation/reassembly, virtual synchronous membership and communication, flow control, causal order, total order, and primary bit maintenance. All these, and future layers support the same interface, called the *Uniform Group Interface* (UGI). The UGI is well-defined, and consists of a set of downcalls and upcalls, and has some support for extension of the interface. The interface provides for, among others, multicasting messages, installing views, and reporting error conditions. The UGI is designed for multiprocessing, and is com-

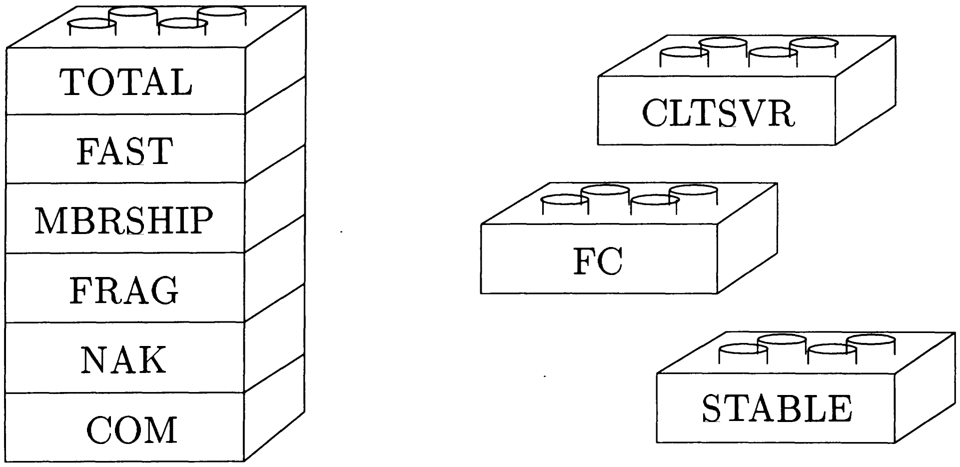


Figure 1: Layers can be stacked at run-time like Lego[tm] blocks.

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

pletely asynchronous and reentrant. See tables 1 and 2 for a complete list of upcalls and downcalls. The UGI is fundamental, allowing users total flexibility in stacking the layers.

The topmost layer will typically offer an application-dependent interface rather than the UGI (the UGI is the most primitive interface to Horus, and is used primarily by protocol developers.) The usual topmost interface is the standard BSD socket interface—an interface

Event Type	Information	Description
JOIN_REQUEST	source	request to join
JOIN_FAILED		request failed
JOIN_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

that is well known to most application programmers. To join a group, a user creates a socket in the Horus addressing domain, and binds it to a Horus group address. Sending and receiving messages can be done using the normal *read* and *write* system calls. It is possible to mix UNIX file descriptors, TCP socket descriptors, and Horus socket descriptors and apply a BSD *select* call. We also provide a reliable multi-threaded UNIX system call library with this interface, allowing multiple *reads* to execute in parallel. The UNIX *ioctl* system call provides control over group properties, such as message ordering and the degree to which and how events such as new group views will be reported to the application program.

Other topmost interfaces include the Transis interface, and the (ORCA) Panda interface (allowing parallel ORCA programs to run over Horus). We are currently developing an interface to support the Message Passing Interface standard that has been developed as a follow-on to PVM for parallel computing, and a Tcl/TK interface. Real-time and object-oriented language interfaces are planned for the future.

4 Layers and Protocols

While supporting the same UGI interface, each Horus layer runs a different protocol. Applications may specify at run-time what stack (or stacks) of layers they wish to use (see figure 1). If an application wishes to use unreliable ATM communication, it specifies the COM:atm stack. If instead it wishes to use totally ordered UDP communication, it would specify the TOTAL:MBRSHIP:FRAG:NAK:COM:udp stack (these layers will be explained shortly).

As another example, a single test program can be used to test the functioning and performance of any set of layers, without recompilation or linking, but by changing a run-time argument. To test the performance of the FC:STABLE:MBRSHIP:FRAG:NAK:COM:atm stack it could be invoked as:

horus_test -s FC:STABLE:MBRSHIP:FRAG:NAK:COM:atm

Such a test program would create a Horus socket and use the *ioctl* system call to push the desired layering onto the socket (much like protocol modules on a System V *stream*). Horus allows parameters to be passed to layers. For example, if messages have to be fragmented to 1024 byte packets, an application specifies MBRSHIP:FRAG(size=1024):NAK:COM:atm.

Although Horus allows layers to be stacked in any order (and even multiple times), most layers require certain semantics from layers below it, imposing a partial order on the stacking. We will now describe some of the most important Horus layers roughly in the partial order from the lowest to the highest.

4.1 COM (basic communication)

This layer does not actually run any protocol. Its purpose is to provide the UGI interface over other low-level communication interfaces; in the example above, atm is a parameter to COM causing COM to connect itself to an ATM device interface. COM currently supports interfaces to IP, UDP (with or without broadcast), the Deering multicast extensions of IP and UDP, ATM, MACH messages, the x-kernel interfaces, and a network simulator which we developed. The default configuration runs over UDP. The COM layer also keeps track of byte-ordering and maintains some low-level message logging event-charts depicting which events happened where and in what order, both of which are useful for debugging and reproducing events that lead to a software failure. Note that COM may also run over reliable communication layers, in which case no additional protocols need be run.

4.2 NAK (FIFO communication)

The NAK layer provides reliable FIFO multicast and point-to-point communication over unreliable communication mechanisms. It does not use a window-based protocol (in fact, no acknowledgements are sent). Instead, it only sends negative acknowledgements when message omission errors are detected, and occasional protocol status update reports (much like XTP [14]). The NAK layer does not provide fragmentation of messages (this is done by the FRAG layer, which is not discussed in this paper). Other layers can implement flow control (see, for example, the FC layer below). The NAK layer reports potential communication problems when it has not received a protocol status update for some time from a member.

If the underlying communication system already provides reliable communication (such as in the case of TCP), the NAK and FRAG layers can be omitted.

4.3 MBRSHIP (membership and atomicity)

Membership and message atomicity is implemented at this layers, the heart of which is the *flush* protocol. The flush protocol is run when a member crash is detected, or when views

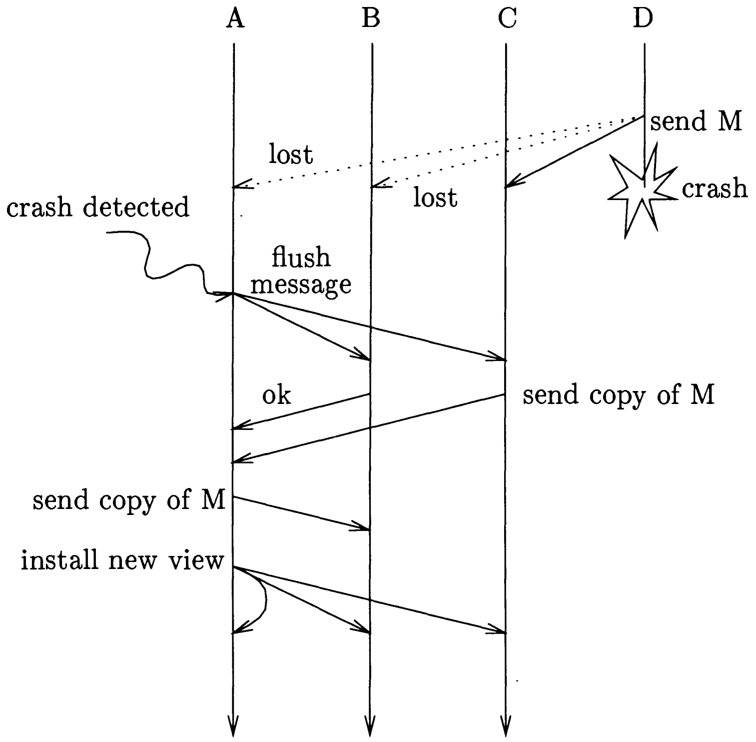


Figure 2: This picture depict four processes: A, B, C, and D. D crashes right after sending a message M, and only C received a copy. After the crash is detected, A starts the flush protocol by multicasting to B and C. C sends a copy of M to A, which forwards it to B. After A has received replies from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

merge. Its intention is to *finalize* the view, so that all surviving members have seen exactly the same set of messages (virtual synchrony). The protocol works as follows: one of the members (usually the oldest surviving member) 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 unstable (see next section) messages from failed members, followed by a FLUSH_OK reply message (note that for this it is necessary that all members log all unstable messages). 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.

[1]By picking the oldest group member, 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.

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 the MBRSHIP layers of any set of groups, so that they have the same (consistent) view of the environment.

4.4 STABLE (message stability)

A message is called *stable* when all members have seen that message. Each member can decide what it means to "have seen" a message. This may be immediately after message delivery, after the message has been written to disk, or after some external action. At that point the application invokes the UGI *ack* downcall. The UGI downcall is an internal acknowledgement of the member to the protocol layer, and does not necessarily result in an immediate message back to the sender of the message.

Stability information is useful to several other layers and applications. The membership layer uses stability information for garbage collection of messages. The current flow control layer uses it to control the flow of messages through Horus. The causal delivery layer uses it occasionally when communicating across different groups. Some fault-tolerant applications delay external action after receiving a message until that message is stable (named *safe* delivery in the Transis system).

To track message stability, the STABLE layer periodically broadcasts a report that includes the sequence numbers of the last messages that have been acknowledged. It also piggybacks some information on existing traffic. The layer collects the stability information in what is called the *stability matrix*. When multicasts are sent, some of our protocols need to know about k-stability (when k members have seen the message, where k may be less than the full set of group members). In support of this, the stability matrix indicates not only full, but also partial stability. The stability matrix is local and its ij entry stores the number of messages sent by i which have been acknowledged by j. The stability matrix is shared between the users and the stability layer itself, and updates to the stability matrix are reported using STABLE event upcalls.

4.5 FC (flow control)

The FC layer is currently the only mechanism that provides flow control in Horus. It uses a credit (window) based mechanism. Currently, credit is based on the stability information provided by the STABLE layer. When the number of unstable messages in a group exceeds some maximum, the FC layer starts delaying messages. When the number of unstable messages goes below this threshold, the buffered messages are packed together into a single message subject to some maximum number of bytes and sent in a single send operation. Both the maximum number of unstable messages and the maximum number of bytes are adjustable at run-time. (We envision that in future these values will be determined dynamically based on run-time statistics.) Manual adjustment of these parameters shows that this scheme is quite effective, increasing the performance in several important situations.

Note that the flow control layer does not provide any *back pressure* (it does not *block* senders, but instead *buffers* the messages that cannot be sent right away). Instead, back pressure is accomplished by −−MUTS, which slows down processes that allocate more memory than they release (we have started working on a paper that describes this idea in more detail).

4.6 CAUSAL (causally ordered delivery)

This layer tracks causal relationships between messages using vector-clocks. Causal delivery is provided by a separate layer (ORDER) that has to be layered over the CAUSAL layer. The ORDER layer can also be used to provide *safe* delivery of messages (messages that are known to be fully stable), if run over the STABLE layer). For brevity, the present paper did not experiment with the CAUSAL and ORDER layers, but their overhead relative to that of the NAK layer is known to be very small. The ORDER layer is largely inactive, just passing messages through, unless group members consume messages at very different rates (which is unusual in our experience).

4.7 TOTAL (totally ordered delivery)

The TOTAL layer uses a token to implement totally-ordered delivery. Before a message can be sent, a member has to acquire the token. The member then places a sequence number on each message it sends, and messages that arrive out of sequence are delayed until they can be delivered in order. Token requests and token passing is done as much as possible on existing traffic. In fact, if the traffic load is high, *all* requests and token passes are piggybacked, resulting in extremely efficient totally ordered communication. The token is not fixed in one spot (sequencer), nor is it cycled through all members. Instead, it cycles through the current set of senders, an approach that works well if a set of senders present a uniform, high load. If, on the other hand, the traffic load is low, the latency and overhead of our protocol becomes relatively high. For these applications we are developing a second totally ordered layer that will use a different strategy. As with other layers, applications will be able to choose at run-time which layer they require.

4.8 XFER (state transfer)

When a new process joins a group, it needs to be updated with the current state of the application. Similarly, when two partitions of the same group merge, they often need to agree on a common state before normal operation can proceed. The XFER layer provides for this *state merge* or *state transfer*. It adds a phase to a membership change, in which processes can exchange state information. The XFER layer detects and notifies termination of this transfer. In the simplest and most common case, this consists of a contact member of the primary partition of the group transfering all its state to the set of new or non-primary members. In more complex cases a simple termination detection algorithm is necessary to detect completion of the merge.

4.9 CLTSVR (client-server membership)

The flush protocol used by the MBRSHIP layer scales to perhaps a hundred members (if the membership is fairly static), but will then start showing signs of severe performance degradation. To allow scaling to groups with thousands of members (or even more), we designed a second membership protocol in which a tree-structure is superimposed on large groups.

In this layer, the members are divided into two sets: *servers* and *clients*. The servers run over the MBRSHIP layer. Each server is responsible for a set of clients, and forwards any events that it receives. By default, a client communicates through its responsible server. If a server fails, another server will adopt the orphaned clients. This way, the protocol simulates exactly the same group model as the MBRSHIP layer, with better scalability but at the cost of a higher average latency for clients. Scalability and latency can be traded off further by stacking multiple CLTSVR layers on top of each other (creating multiple levels in the hierarchy). Ordering properties used by the server members are inherited by clients. For example, if the servers use the TOTAL layer, messages will be received in a total order at the clients as well as the servers.

We anticipate two primary uses for the CLTSVR layer. One of these is in support of Replicated Remote Procedure Call (RRPC), where the set of replicas (or the primary and its backups) coincides with the set of servers. RRPC can be used to provide fault-tolerance, load balancing, and parallel request processing in servers shared by large sets of clients. The second expected use is for dissemination of data from a small set of sources to a large set of processes (e.g., for use in a brokerage or factory automation system).

Although our current CLTSVR layer provides the full semantics of the group mechanisms to the clients, we also intend to support optionally weakened semantics for greater scalability in these applications. For example, clients may not need to know who the other clients are (or even the membership of the set of servers), and by not reporting this information, overhead can be reduced. Moreover, if some small risk of message atomicity violations can be tolerated by clients, latency of client-server communication can be greatly reduced. Finally, we are exploring options for integrating security features with the CLTSVR layer [11].

4.10 LWG (light-weight groups)

Another important scalability issue is scalability in the number of groups. When Isis was designed, it was envisioned that some tens, maybe hundreds of groups would suffice for all fault-tolerance and parallel execution needs. When Isis was distributed among users, it soon became clear that groups were used quite differently. Rather than using a group per fault-tolerant service, programmers created a group per fault-tolerant object. That is, they used the group paradigm for implementing individual objects. This way they did not have to deal with multiplexing requests for different objects over a single group, and could use the group to address the complete set of replicas and cached copies. This led to a serious problem of scale: if a server crashed, all objects that had a member on that server would start their own flush protocol, leading to a storm of redundant messages on the network.

To address this, Horus includes a protocol layer that multiplexes groups over a small set of *core* groups (much like light-weight threads in a small set of heavy weight processes).

The approach yields a significant amortization of costs of failure recovery, and also improves other aspects of group communication [5]. For example, the cost of joining a new member to a light-weight group is very low, allowing for short-lived membership. Also, the cost of ordering protocols (such as causal or total ordering) underneath the light-weight groups is cheaper than if each group runs its own protocol.

4.11 FAST (message acceleration)

A problem with stacking many layers for some set of features is that most layers add their own header to each message, hence overhead can be considerable. For normal data communication it is important to make the header overhead as small as possible. The FAST layer uses the same protocol as the NAK (FIFO) layer during normal operation, but the message path bypasses most layers. When a view change starts, or a message is lost, the FAST layer switches back to the original message path and retransmits unstable messages. When the abnormal condition disappears, it switches back to the accelerated path. The FAST layer provides only FIFO communication currently, so that the CAUSAL and TOTAL layers need be run over the FAST layer if such ordering is required. In addition, it is important to run the flow control layer over the FAST layer if traffic is high, since high rates of loss cause frequent switching between the FAST path and the normal one, and consequent high overhead. We are considering ways of modifying this to automatically use the flow control layer if the rate of message loss exceeds some threshold.

4.12 Other layers

The Horus project has several other layers under development. These include a rate-based flow control layer, a clock synchronization and timestamping layer, a layer that support remote procedure call, a message logging layer, a message signing and a message encryption layer, and a layer that simplifies recovery from network partitioning problems (particularly for replicated objects).

5 Implementation

The Horus system is implemented over the −−MUTS (Multi-threading and Unreliable Transport Service) kernel (a scaled down version of an earlier system we developed called MUTS). −−MUTS provides prioritized multi-threading and a collection of drivers for different network interfaces such as UDP sockets, the MACH ATM interface, and the x-kernel interface. −−MUTS contains a sophisticated message interface that avoids copying, even when crossing address spaces. −−MUTS also manages how memory is divided among the modules of an application, in an attempt to deal with scarce memory situations while avoiding deadlock.

Using the services of −−MUTS, each Horus layer implements the interface of tables 1 and 2. The interface names are made unique for linking purposes. A separate dispatch table per layer is used to implement interface overloading. Although Horus supports C++ programs, it is implemented in C for maximum portability and performance.

A process can specify at run-time, at each layer, how it wants upcalls to be invoked. This allows for maximum performance and ease of use. We currently support three types of upcall dispatch: thread creation, procedure invocation, and message enqueuing. Horus numbers all events, hence in the threaded case, concurrently active threads can synchronize themselves in event delivery order by using the Horus event number as the input to an event count synchronization barrier [10]. Such a barrier creates a mutual exclusion region that threads can only enter in sequential order.

As noted earlier, most applications use Horus through a socket interface. This is implemented by a socket layer that intercepts UNIX system calls, reimplementing them so that Horus can run multiple threads and support group communication sockets. It also allows multiple subsystems to co-exists in the same UNIX process. For example, one thread can run X Windows client code, while another thread deals with group communication. −−MUTS supports special file descriptors that can be used for safe interaction between the threads.

6 Performance

A major concern of our architecture is the overhead of each layer. Prior work on the *x*-kernel has demonstrated that modularization and layering does not necessarily mean bad performance, and our initial experience confirms this. To get an idea of the price of layering, we stacked the fragmentation layer ten times and compared the performance to stacking it only once. We found that the cost of the fragmentation layer adds 50 μseconds to the one-way latency. We believe we can bring this down somewhat. In this section we present the overall performance of Horus on a system of SUN Sparc10 workstations running SunOS 4.1.3. The workstations communicate through a loaded Ethernet consisting of multiple segments connected by a Powerhub concentrator. When reporting performance for communicating between just two machines, the workstations reside on the same Ethernet segment.

We used two network layers: normal UDP, and UDP with the Deering hardware multicast extensions. It should be noted, however, that in the second case Horus will use UDP for point-to-point messages. In particular, if a group has only two members, only normal UDP is used. To highlight some of the performance numbers: we achieve a one-way latency of 1.2 msecs over the FAST:MBRSHIP:FRAG:NAK:COM:udp stack (we think we may be able to bring this down to less than a millisecond in the near future), and 7,500 1-byte messages per second (with the TOTAL:MBRSHIP:FRAG:NAK:COM:udp stack). With some help from the application, we can drive up the total number of messages per second to over 75,000 using the FC layer. We easily reach the Ethernet 1007 Kbytes/second maximum bandwidth with a message size smaller than 1 kilobyte.

Our performance test program has each member do exactly the same thing: send k messages and wait for $k \times (n-1)$ messages of size s, where n is the number of members (see figure 3). It runs this for r rounds and divides the results by r. In the results that we present below, we have not used the FAST (message acceleration) layer, as it does not yet provide full virtual synchrony.

We use the terms *latency* for the time between sending a message and receiving it, *bandwidth* for the total number of bytes received per second, and *throughput* for the total number of messages received per second. To measure latency, we choose $k = 1$ and $s = 1$. To

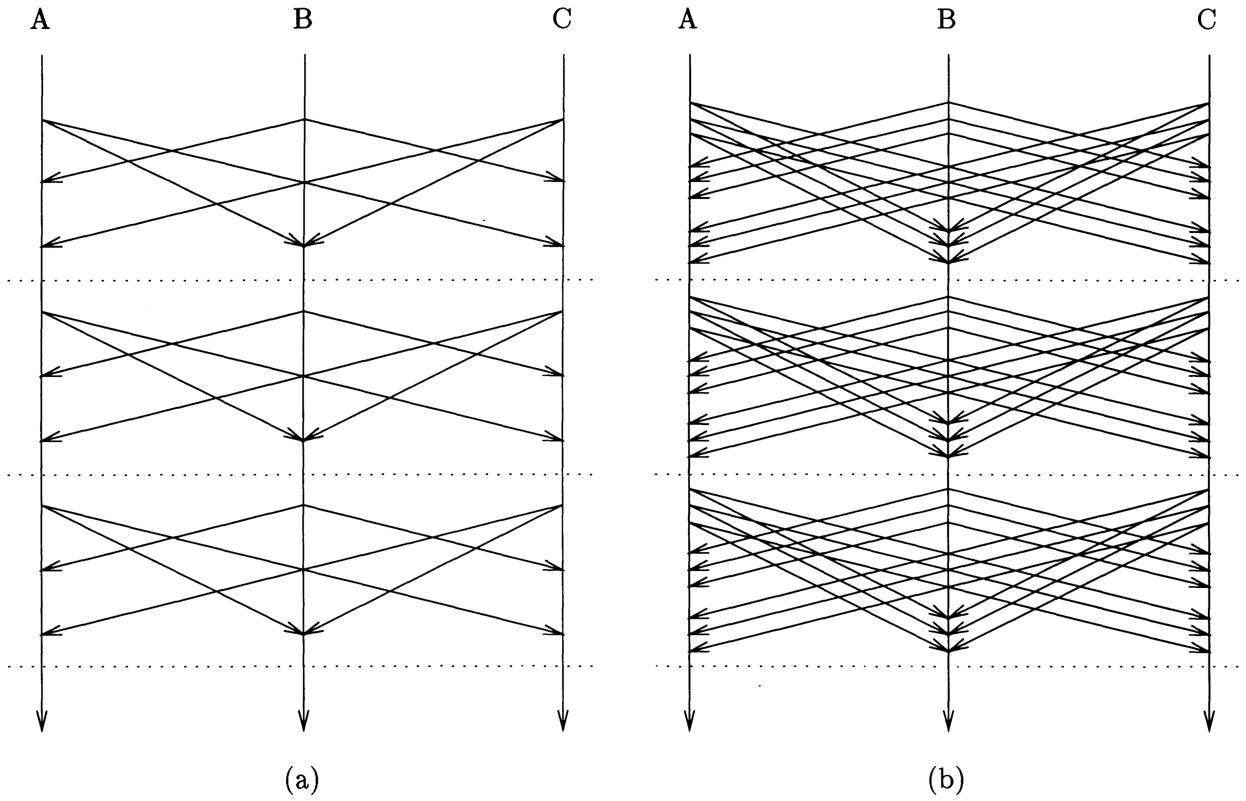
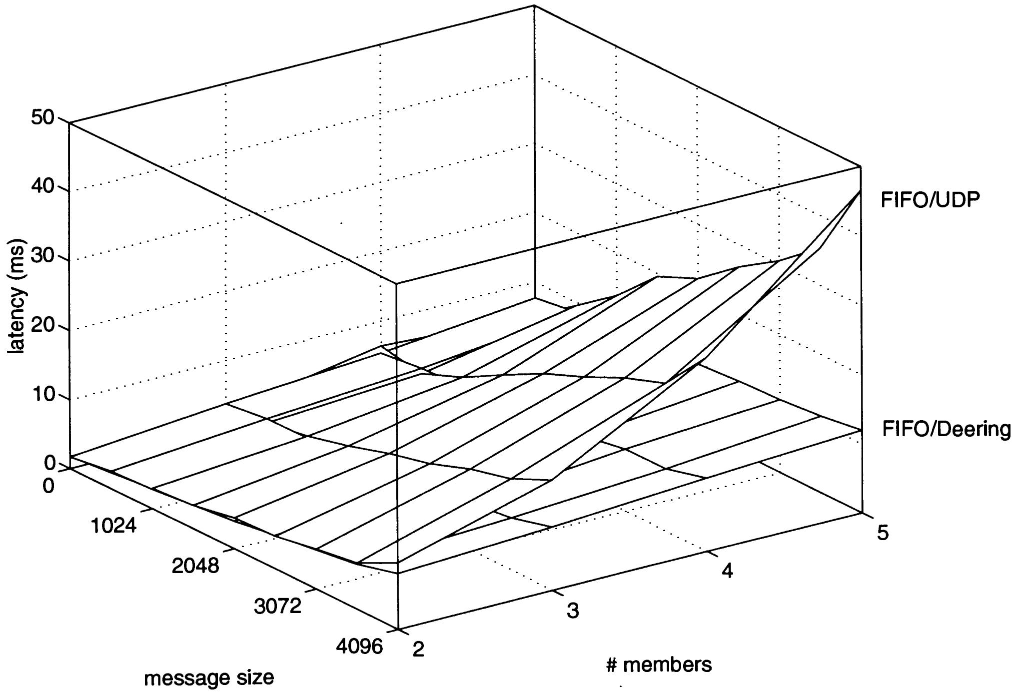


Figure 3: The performance test protocol runs rounds in which each member multicasts k messages and subsequently waits for $k \times (n-1)$ messages (where n is the number of members). In (a) $k = 1$, and in (b) $k = 3$.

measure bandwidth we choose a large s (on the order of 4 Kbytes). To measure throughput, we use $s = 1$ and a large k (on the order of 25 messages per round).

Figure 4 depicts the one-way communication latency of Horus messages. As can be seen in the top graph, hardware multicast is a big win, especially when the message size goes up. This is not surprising, since without hardware multicast $n - 1$ more messages need be sent to simulate the multicast, and hence quadratic behavior results. In the bottom graph, we compare FIFO to totally ordered communication. For small messages we get a FIFO one-way latency of about 1.5 milliseconds and a totally ordered one-way latency of about 6.7 milliseconds. As explained in section 4.7, the totally ordered layer is not particularly efficient for all senders sending at random and synchronously. In case of only one sender, the one-way latency is 1.6 milliseconds for this ordering. The next figure will show that the asynchronous message throughput of totally ordered communication is excellent.

Figure 5 shows the number of 1-byte messages per second that can be achieved for three cases. For normal UDP and Deering UDP the throughput is fairly constant, independent of the number of messages per round and going down slowly with each additional member. For totally ordered communication we see that the throughput becomes better if we send more



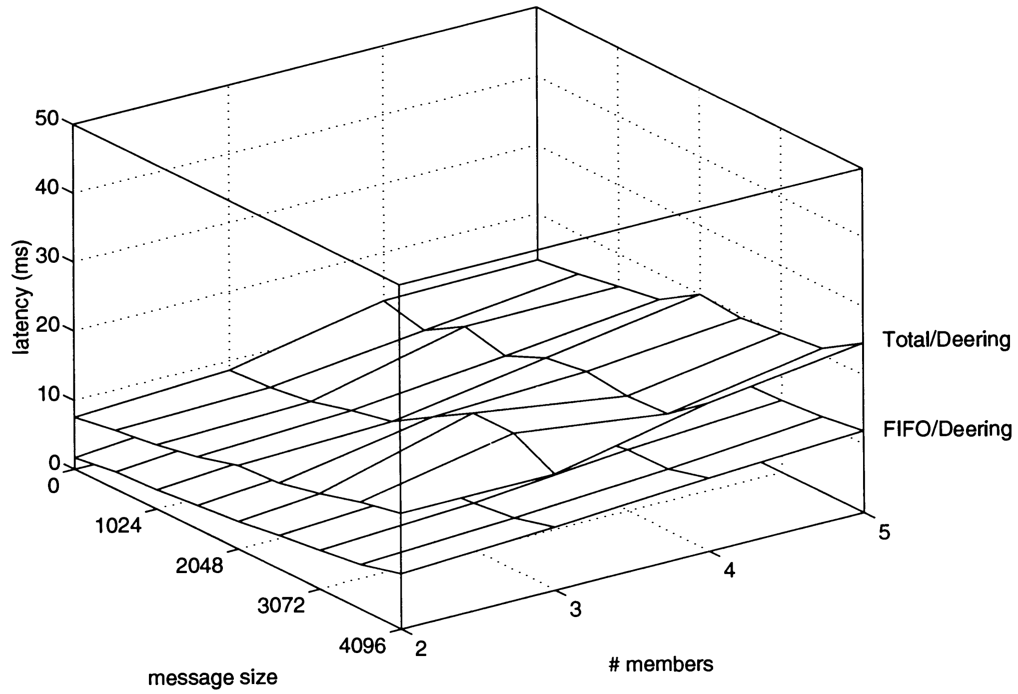


Figure 4: The top figure compares the one-way latency of FIFO Horus messages over straight UDP and UDP with the Deering hardware multicast extensions. The bottom figure compares the performance of total and FIFO order of Horus, both over UDP multicast.

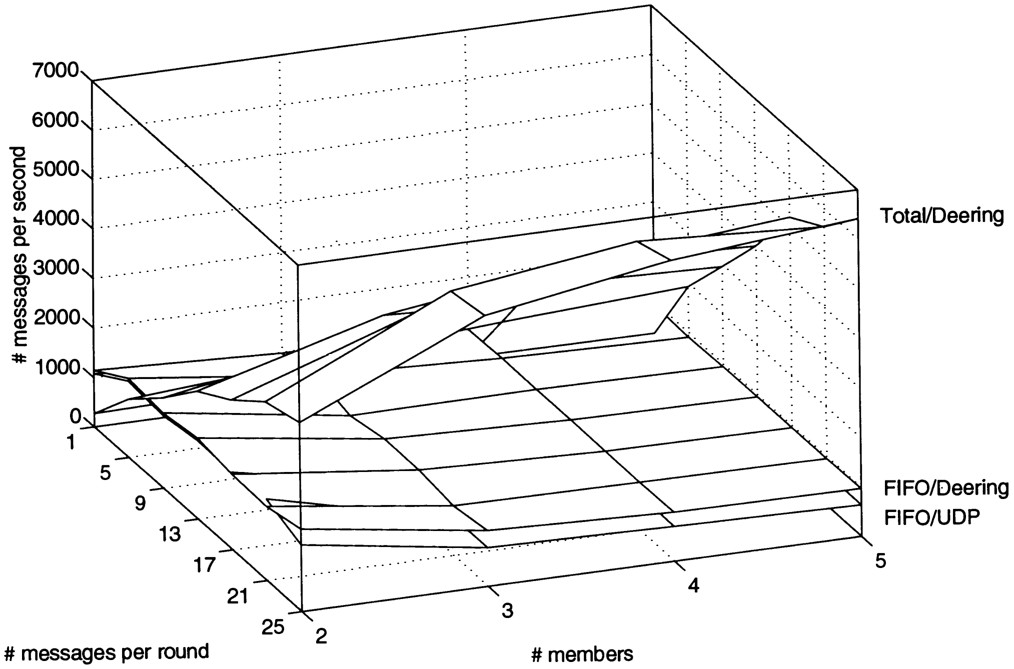


Figure 5: These graphs depict the message throughput for virtually synchronous, FIFO ordered communication over normal UDP and Deering UDP, and for totally ordering communication over Deering UDP.

messages per round (because of increased asynchrony). Perhaps surprisingly, the throughput also becomes better as the number of members in the group goes up. The reason for this is threefold. First, with more members there are more senders. Second, with more members it takes longer to order messages, and thus more messages can be packed together and sent out in single network packets. Last, our ordering protocol allows only one sender on the network at a time, thus introducing flow control and reducing collisions.

7 Ongoing work

The Horus project is an ongoing effort. Although the initial version of Horus is nearing completion, we have yet to work out some important issues. For example, the current system can support several types of RPC mechanisms, and a great number of options exist for implementing state transfer (from a group to a joining member) and state merge. Selection of a preferred strategy for solving these problems and optimization of the corresponding mechanisms will occur in the coming months. Better support for flow control and resource management is another area of intensive activity.

We are also looking at running Horus on more advanced platforms. A specific target of our current work is stripped-down compute nodes over a high performance communication switch. We anticipate that this configuration of the system will expand our application domain to parallel computing, high performance I/O servers, multi-media, computer-supported collaborative work, etc. To enable these new types of applications, we are considering extending Horus to support real-time features and an object-oriented environment.

Another area of research is security and privacy. Our earlier work on Horus included a security architecture [12]. This work needs to be adjusted to fit in the layered structure of the current system. We are also pursuing research in anonymous communication, enabling message passing without explicit knowledge of the source and the destination addresses.

8 Conclusion

Our work intends to show that a single system can solve the communication needs of a variety of different applications. The Horus system employs a layered architecture for easy customization, experimentation, and extension. Horus supports (but does not impose) the clean virtually synchrony communication model of Isis. In addition, Horus supports multi-threading, and is UNIX-independent.

Acknowledgements

A lot of people contributed to this work. In particular, we would like to thank Brad Glade and Alexey Vaysburd for valuable discussions about the architecture and interfaces, Mike Reiter for his work on security mechanisms for Horus, Katie Guo for initial work on the CLTSVR layer, and William Chan for work on monitoring and resource control. Robert Cooper and Barry Gleeson provided useful suggestions early in the design of Horus. We have learned much from discussions with people from the Transis project, notably Dalia Malki, Yair Amir, and Danny Dolev. Werner Vogels, Gautam Thaker, Dalia Malki, and Mark Hayden provided useful comments on an initial draft of this paper.

References

[1] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Membership algorithms in broadcast domains. In A. Segall and S. Zaks, editors, *Proceedings of the Sixth WDAG; Israel*, pages 292–312. Springer-Verlag, 1992.

[2] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication subsystem for high availability. In *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84, Boston, MA, July 1992. IEEE.

[3] Kenneth P. Birman. The Process Group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

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

[5] Brad B. Glade, Kenneth P. Birman, Robert C. Cooper, and Robbert van Renesse. Lightweight process groups. In *Proceedings of the OpenForum '92 Technical Conference*, pages 323–336, Utrecht, The Netherlands, November 1992.

[6] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn-Hummel, and Henri E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

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

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

[9] 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.

[10] D. P. Reed and R. K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2):115–123, February 1979.

[11] M. Reiter, K. P. Birman, and R. van Renesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 1994. Accepted for publication.

[12] Michael Reiter, Kenneth P. Birman, and Li Gong. Integrating security in a group-oriented distributed system. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 18–32, Oakland, California, May 1992.

[13] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. CHORUS distributed operating systems. *Computing Systems Journal*, 1(4):305–370, December 1988.

[14] W. T. Strayer, B. J. Dempsey, , and A. C. Weaver. *XTP: The Xpress Transfer Protocol.* Addison-Wesley, Reading, MA, 1992.

[15] Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. The performance of the Amoeba distributed operating system. *Software–Practice and Experience*, 19(3):223–234, March 1989.

[16] B. Whetten. A reliable multicast protocol. Technical Report in progress, U.C. Berkeley, 1994.