Horus: A Flexible Group Communications System

Robbert van Renesse, Kenneth P. Birman, Brad Glade, Katie Guo, Mark Hayden, Takako M. Hickey, Dalia Malki, Alex Vaysburd, and Werner Vogels[1]

Dept. of Computer Science, Cornell University

Abstract

The Horus system offers flexible group communication support for distributed applications. It is extensively layered and highly reconfigurable, allowing applications to only pay for services they use, and for groups with different communication needs to coexist in a single system. The approach encourages experimentation with new communication properties and incremental extension of the system, and enables us to support a variety of application-oriented interfaces.

1 Introduction

Recent years have seen growing use of group communication in distributed, fault-tolerant, and/or parallel applications. Simultaneously, there have been major advances in the architectures and capabilities of operating systems, including changes in the handling of memory used for message passing, and increasingly sophisticated use of threads and upcalls as a structuring technique. This paper reports on the Horus project, which is developing a group communication system that can run either in user space or in a microkernel, and that offers great flexibility in terms of the programming model exported to application-developers, the properties provided by a protocol stack, and the overall system configuration.

Strongly motivated by the *x*-kernel [4], Horus system uses an architecture in which protocols are constructed by stacking small microprotocols, which support a common interface. Unlike *x*-kernel, however, our orientation is on group communication (*x*-kernel focuses on RPC and streams), and it was our goal to support the virtual synchrony model of the Isis system, side by side with weaker reliability models.. This is not straightforward: virtually synchronous protocols to support large numbers of groups, dynamic group membership, message ordering, synchronization and failure handling can be complex [2]. The Horus system design integrates ideas developed in Isis, Transis [1], and the *x*-kernel, while introducing novel mechanisms in support of high performance, reliable, group communication.

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.

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

[1]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 address for contact author: rvr@cs.cornell.edu.

performance figures for a version of the system running in user-space over SunOS™. For brevity, we omit discussion of the fault-tolerance protocols and execution model.

2 Architecture

Some form of process group can be found in a great number of distributed systems and protocols, although with varied semantics. Groups are used in embedded systems (for example to implement TMR voting), clock synchronization, file system caching, and are employed in operating systems like UNIX for signal delivery. A group of processes may jointly hold a reference to a port in Mach, or a file descriptor in UNIX. The V and Chorus operating systems have notions of group services: a request will be sent to some member of the group. Server groups are often found in fail-over architectures, where a request to a failed server is automatically reissued to an operational one. IP-multicast has become popular: it wraps a hardware group abstraction with a simple to use software interface and routing support. In Isis and related systems such as Transis, Psync, Totem, RMP, and Rampart, groups are used in a *virtual synchrony* architecture supporting fault-tolerant tools, such as for load-balanced request execution, fault-tolerant computation, coherently replicated data, and security. The AAS system, proposed by IBM for a next generation air traffic control application, was based on real-time process groups. Moreover, conventional peer relationships between processes, such as for stream and RPC communication, can also be viewed as group communication in a trivial case; there may be considerable benefits to doing so, for example if process migration is supported.

At the core of Horus is a group abstraction intended to be flexible enough to capture varied styles of process group, while facilitating a well-structured, modular, protocol implementation environment. We find it useful to think of this central protocol abstraction as resembling a Lego™ block; the Horus "system" is thus like a "box" of Lego blocks. Although each type of block implements a different communication feature, the blocks have standardized top and bottom interfaces that allows them to be stacked on top of each other at run time in a variety of ways (see Figure 1). As in Lego, not every building makes sense, but you have great flexibility in building exactly what you want. Horus has some forty different blocks. About ten of these blocks are "anchor blocks," that is, they can only be used at the bottom or the top of a stack of blocks. They provide either a low-level interface to the host operating system, or a high-level interface to the user.

Technically, each type of block is a software module with a set of entry points for downcall and upcall procedures. For example, there is a downcall to send a message, and an upcall to receive a message. There is a list of sixteen downcalls and fourteen upcalls that each layer can support. We will describe these later. For each layer, this list is registered with the Horus run-time system at initialization time under a certain ASCII name.

A user specifies a list of these ASCII names to instantiate the corresponding blocks, and to stack them on top of each other in the order specified. To see how this works, consider the Horus *message_send* operation. It looks up the message send entry in the topmost block, and invokes that function. This function may add a header to the message, and will then typically invoke *message_send* again. This time, it will invoke the message send function

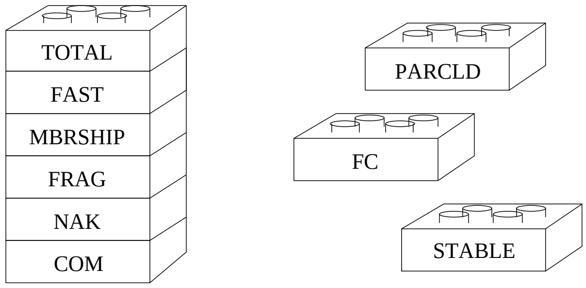


Figure 1: Layers can be stacked at run-time like Lego[tm] blocks.

in the layer below it. This repeats itself recursively until the bottommost anchor block is reached. This block invokes a driver for a particular network to actually send the message.

Each stack of blocks is carefully shielded from other stacks. Each stack has its own prioritized threads, and has controlled access to available memory through a mechanism called *memory channels* (see Figure 2). Horus has a memory scheduler that dynamically assigns the rate at which each stack can allocate memory, depending on availability and priority, so that no stack can hog the available memory. This is particularly important if running inside a kernel, or if one of the stacks has soft real-time requirements. Notice that each layer can determine its own memory policy. Thus a layer with real-time properties might use static preallocation, trusting the application to respect pre-specified data rate limitations. An asynchronous layer, on the other hand, might allocate memory dynamically, implementing flow-control to avoid excessive memory consumption.

Besides threads and memory channels, each stack deals with three other types of objects: endpoints, groups, and messages. 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 destination endpoint addresses. Since a group object is purely local, Horus allows different endpoints to have different views of the same group address. An endpoint may have multiple group objects, allowing it to communicate with different group addresses and different views.

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. Messages are typically allocated from the memory channel of the stack, but they can be allocated from other memory channels as well.

A user creates a stack by calling the *endpoint_create* function and specifying the stack that it wants by means of an ASCII string, for example:

create_endpoint("TOTAL:FAST:MBRSHIP:FRAG(size=1500):NAK:COM(protocol=atm)")

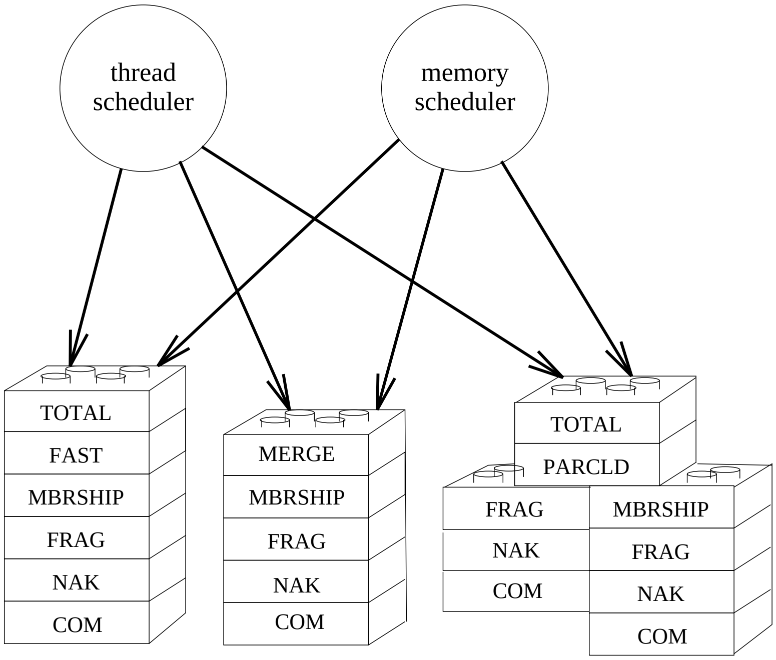


Figure 2: The Horus stacks are shielded from each other, and have their own threads and memory, each of which is provided through a scheduler.

This function calls the *endpoint_create* entry of the topmost layer (the TOTAL layer) with the remainder of the stack (FAST:MBRSHIP:...). The TOTAL layer can use this to allocate memory it needs for endpoint administration. It then calls the global *endpoint_create* function recursively with the remainder of the stack. As illustrated, the user can supply certain arguments to each layer. The structures that each layer allocates for endpoint administration are linked together in a list, and returned to the user.

Before the user can communicate, it has to allocate a group object with the *group_create*[2] function. *group_create* takes a group address argument. A group_create entry point is invoked for each protocol layer, following the linked list that was set up by the *endpoint_create* function. Furthermore, the user has to install an initial view of peer endpoints using the *view_install* function. Finally, it calls the *message_send* operation to send the message, which follows, again, the same path through the layers recursively. During normal operation, a user can install new views to extend or decrease the list of peer endpoints.

A thread at the bottommost layer waits for arriving messages. When a message arrives, the bottommost layer pops off its header, and passes the message on to the layer above it. This repeats itself recursively. Each layer has the option to drop the message, or to buffer it for later delivery. When multiple messages arrive simultaneously, it may be important to enforce an order on the delivery of the messages. However, since each message is delivered using its own thread, this ordering may be lost. Therefore, Horus numbers the messages, and uses *event count* synchronization variables [5] to reconstruct the order where necessary. In

[2]In Horus, this function is actually called *join* for historical reasons.

this approach, threads can be serialized to enter mutual exclusion regions in a predetermined order, for example given by sequence numbers in messages. A high level of concurrency is thus maintained until an order-sensitive operation is to be performed. Using multiple event counters, multiple sequenced mutex regions can be supported. Automatic synchronization is available for layers that do not require high levels of concurrency; when enabled, each (stack,layer) pair is treated as a mutual exclusion region.

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. Indeed, the segments of a single Horus message can reside in multiple address spaces. Multiple Horus messages may be packed together into a single message, which some layers (like the FC layer described below) and applications have used very effectively to amortize the per message costs. This will be illustrated in the performance section.

3 Layers and Protocols

For the layers to fit like Lego blocks, they each must provide the same downcall and upcall interfaces, even if each runs a different protocol. A lesson learned from the x-kernel is that if the interface is not rich enough, extensive use will be made of general purpose control operations (a la *ioctl*), which reduces the flexibility. (Since the control operations are unique to a layer, the Lego blocks do not "fit" as easily.) The *Horus Common Protocol Interface* (HPCI) is therefore more extensive than the corresponding interface of the x-kernel. Furthermore, the HCPI is designed for multiprocessing, and is completely asynchronous and reentrant. See tables 1 and 2 for a complete list of downcalls and upcalls.

The HPCI is currently available for the C, C++, ML, and Python languages. Layers can can be written in any of these four languages, and linked together at run-time. The choice of language reflects basic tradeoffs between performance and level of assurance. Layers coded in C or C++ perform better, but are more difficult to validate. Python is convenient for rapid prototyping, while ML layers are well suited to formal verification.

The topmost layer will typically offer an application-dependent interface rather than the HCPI (the HCPI is the most primitive interface to Horus, and is used primarily by protocol developers.) One of our topmost interfaces is the standard BSD socket interface—an interface that is well known to many 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 *read*s 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.

downcall	argument	description
endpoint_create	protocol stack, lower endpoint	create a comm. endpoint
endpoint_destroy	endpoint	clean up endpoint
group_create	endpoint + group address	create group object and return handle
group_leave	group handle	leave group
group_dump	group handle	print layer information
group_focus	identifier	focus on layer, return handle
view_install	group handle, list of peers	install a group view
view_flush	list of failed peers	remove peers and flush
view_flush_ok	group handle	go along with flush
view_merge	view contact	merge with other view
merge_denied	merge request	deny merge request
merge_granted	merge request	grant merge request
message_cast	message	multicast a message
message_send	message + subset of peers	send message to subset
message_ack	message	acknowledge a message
message_stable	message	message is stable

Table 1: Horus downcalls

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

Table 2: Horus upcalls

Other topmost interfaces include the Transis interface, the (ORCA) Panda interface (allowing parallel ORCA programs to run over Horus), and the IBM PCODE interface which is intended to support the MPI multi-processing standard. We have also developed a

Tcl/TK interface, allowing rapid construction of applications and distributed widgets (as is the case with the Python/TK interface).

While supporting the same HCPI, each Horus layer runs a different protocol. 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.

COM (basic communication). This layer does not actually run any protocol. Its purpose is to provide the HCPI 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 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.

NAK (FIFO communication). The NAK layer provides reliable FIFO multicast and point-to-point communication over unreliable communication mechanisms. By default, 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 [7]). Optionally, however, a window may be specified, in which case it uses the update reports to decide whether a message can be sent. The NAK layer reports potential communication problems when it has not received a protocol status update for some time from a process. If the underlying communication system already provides reliable communication (such as in the case of TCP), the NAK layer can be omitted.

A version of this layer, NNAK, exists that implements the cyclic UDP protocol for multimedia communication. In this case, messages are prioritized, and have an expiration time. It delivers high-priority messages with a higher probability than low-priority messages, and drops messages that expire. This is particularly useful for MPEG video streams.

FRAG (fragmentation/reassembly). This layer implements fragmentation and reassembly of messages. The protocol is sensitive to the stack of layers below it: a single bit suffices to identify the last fragment in a sequence when FRAG runs on a reliable protocol; a sequence number is needed when running on an unreliable stack.

MBRSHIP (membership and atomicity). Membership and message atomicity is implemented in this layer, 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. We are currently looking into the possibility of separating the concerns of membership and atomicity by splitting this layer into two layers.

STABLE and PINWHEEL (message stability). A message is called *stable* when all peers have seen that message. Each process 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 HCPI *ack* downcall. The HCPI downcall is an internal acknowledgement of the receiver 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).

Currently there are two different protocols that track message stability, STABLE and PINWHEEL. PINWHEEL uses a rotating token message, and works well in relatively small LANs, and gives fast stability reports. STABLE is intended for more general use, but disseminates information slowly by piggybacking it on existing traffic.

FC (flow control). Flow control is increasingly recognized as one of the most difficult challenges faced in asynchronous group communication systems. The FC layer is currently the only mechanism that provides flow control in Horus. FC uses a credit (window) based mechanism, where credit is based on the stability information provided by the STABLE or PINWHEEL 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. Manual adjustment of these parameters shows that this scheme is quite effective, increasing the performance in several important situations. Looking to the future, we envision that FC parameters can be determined dynamically based on run-time statistics. We will also introduce additional flow-control schemes based on rate control and reservations for use in real-time applications.

CAUSAL, ORDER and TOTAL (ordered message delivery). The CAUSAL layer tracks causal relationships between messages using vector-clocks. Causal delivery is then provided by a separate layer (ORDER), which must be layered over CAUSAL. 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 processes consume messages at very different rates (which is unusual in our experience).

Currently, Horus supports four TOTAL layers, each implementing a different protocol for totally-ordered message delivery. Although any one will work, each one outperforms the others for certain message traffic situations.

MERGE (network partitions). The MBRSHIP layer does not automatically locate members of the same group. There are several ways in which this can be achieved. Often, our applications use a directory server. The MERGE layer uses broadcast instead. This scales badly, but deals relatively well with temporary partitions. In each partition, a process is elected that regularly broadcasts a message listing the groups and their memberships in that partition. When a process in a group receives that message, but is not in the corresponding membership, it will start a merge phase (see the next section).

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 the oldest member transfering all its state to the set of merging members. In more complex cases, a simple termination detection protocol is used to detect completion of the merge.

PARCLD (hierarchical membership). The flush protocol used by the MBRSHIP layer scales to perhaps a hundred members (if the membership is fairly static and the load not extremely high), 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.

We anticipate two primary uses for the PARCLD 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 parents. 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 (parents) to a large set of client (child) processes (e.g., for use in a brokerage or factory automation system).

Other than most layers, the PARCLD layer is stacked on top of two protocol stacks (see Figure 2). The parent processes use virtually synchronous communication provided by the membership protocol to communicate with each other, so that they can agree on who is in charge of which child processes. However, for communication with the child processes they only need FIFO communication.

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. However, it also created a serious problem of scale: if a server crashed, all objects that had a member on that server would start their own failure recovery protocol, leading to a storm of redundant messages on the network.

In contrast, 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 [3]. For example, the cost of joining a new member to a light-weight group is relatively 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.

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 bypass functionality of FAST is interesting because it points to a contrast between Horus and *x*-kernel. In the *x*-kernel, protocols are represented as potentially complex control-flow graphs; Horus, on the other hand, presently supports a mostly pure stack model, with FAST skipping some layers but not implementing any sort of very general control flow choices, and PARCLD running over two stacks. We view this as a virtue of Horus, because it has kept the system simple, but it is also a limitation, since we lack mechanisms for linking the behavior of sets of stacks. Thus, if an application uses a real-time stack or a security stack side by side with a virtually synchronous stack or a stack with no reliability properties at a all, one might still want to loosely link the stacks, for example to relate group membership across the stacks. Horus currently lacks features to support this sort of behavior, but we may need to examine the issue in the future.

Other layers. Horus includes many other layers. Existing layers include 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 geared towards replicated objects. A display layer, DISP, can be placed in the stack multiple times; it passes upcalls and downcalls through while tracing selected events for debugging purposes.

Horus currently can run either in user space or in kernel space, but it is sometimes desirable to run the bottom of a protocol stack in kernel space, stacking other layers on it in user space. For this we are developing two anchor layers. The KERNEL layer is stacked

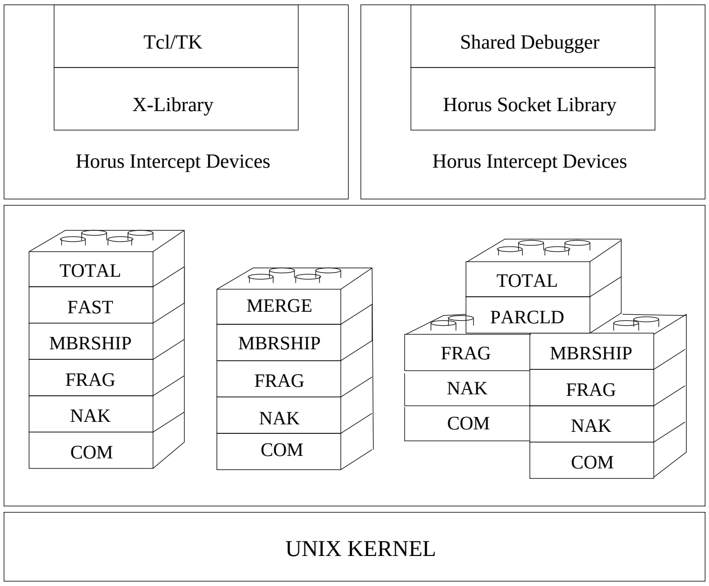


Figure 3: UNIX system calls can be intercepted by Horus using a concept called *intercept device*. These allow the implementation of new socket domains in user space, and permit us to safely link thread-unsafe applications with the Horus system.

on top of the kernel stack, and the USER layer at the bottom of the user stack; these communicate with each other using a customized system-call interface.

4 System Call Interception

Many Horus applications are built using packages such as X-windows and Tcl/TK, which were not designed with threads and Horus communication stacks in mind. Complicating matters, they each do their own event scheduling. To enable Horus to coexist with such packages, we find it convenient to run each package as a separate thread, intercepting certain of the system calls issued. Currently, we only provide this for UNIX processes (see Figure 3). We can intercept UNIX *open* and *socket* system calls, and subsequent system calls on the resulting file descriptors. The system calls are redirected to functions that can be registered at run time.

We use this in two different ways. First, as mentioned previously, many applications use Horus through a socket interface. This is done by intercepting calls to sockets in the *HORUS domain*. The socket is simulated by an intercept device. Calls to the intercept device are implemented over Horus communication stacks. Socket calls to other domains are forwarded to the UNIX kernel (via the threads module to deal with blocking system calls).

Secondly, we use this to support X-windows and Tcl/TK. These systems run their own scheduler using *select* to schedule callback procedures for input on file descriptors. Our X-windows and Tcl/TK applications run in a single thread, and open a Horus intercept device to communicate with Horus protocol stacks, and register a callback procedure with the corresponding scheduler. Subsequently, the X-windows or Tcl/TK scheduler calls UNIX *select* on a set of normal UNIX sockets and/or devices, and the Horus intercept device. The *select* call itself is intercepted as well. Since the Horus system is thread-safe, calls to it can be invoked directly from the application. Upcalls, however, are redirected through the intercept device, resulting in a call to the callback procedure when the *select* returns. Note that no change to the UNIX kernel, nor to X-windows or Tcl/TK, is necessary.

The Tcl code registers two functions with the Tcl interpreter, one to create endpoint objects, one to create group addresses. The endpoint object itself can create a group object using a group address. These objects can be used to send and receive messages. Received messages result in calls to Tcl code, which typically interpret the message as a Tcl command. This turns out to be so powerful, that a distributed, fault-tolerant, whiteboard application can be built using only eight short lines of Tcl code, over a Horus stack of seven protocols.

5 Performance

A nice property of our architecture is that we can use a single program to test the functioning and performance of any set of layers, just by changing a run-time argument which gives the stack of layers to test and any desired parameters to those layers. A major concern of our architecture is the overhead of each layer, hence we now focus on this issue.

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). Given an application that can accept lists of messages in a single receive operation, we can drive up the total number of messages per second to over 75,000 using the FC layer, which buffers heavily using the "message list" capabilities of Horus. 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 4). It runs this for r rounds and divides the results by r. In the results that we

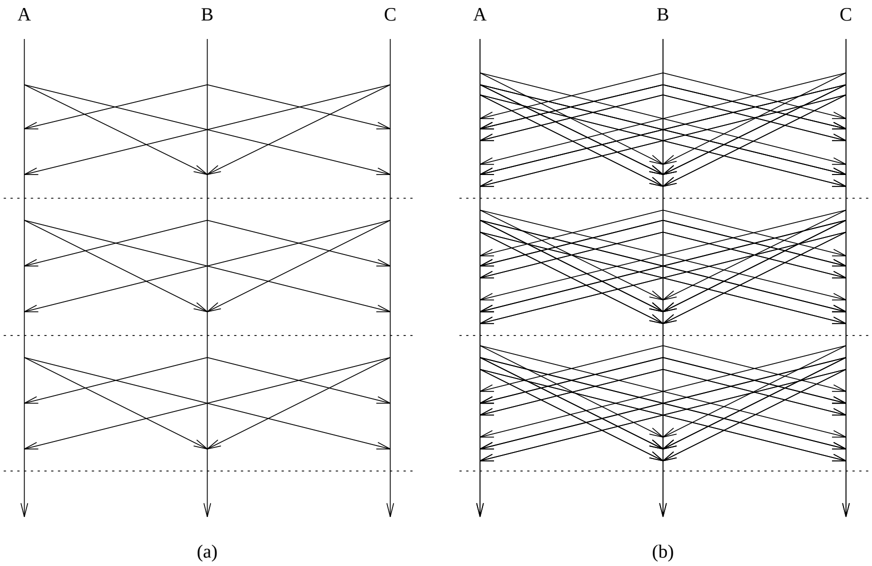


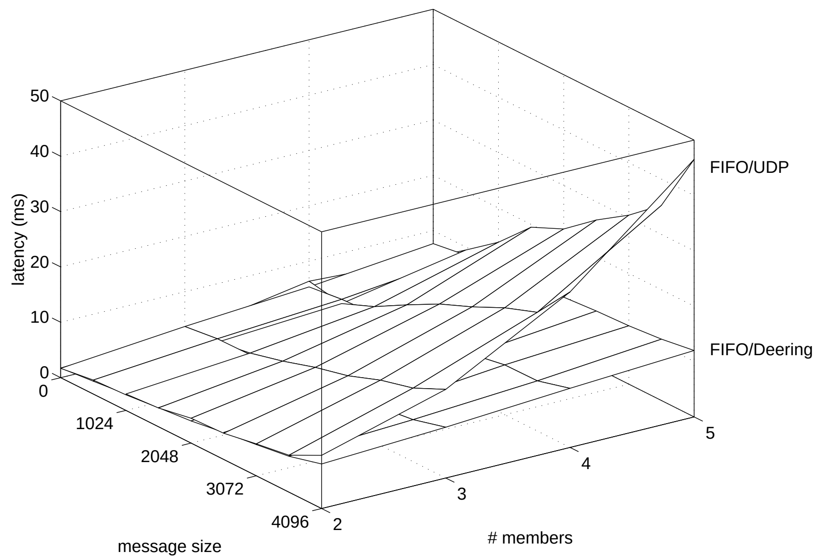
Figure 4: The performance test protocol runs rounds in which each member multicasts k messages and subsequently waits for $k \times (n-1)$ messages (where n is the number of members). In (a) $k = 1$, and in (b) $k = 3$.

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 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 5 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 3, 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 6 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 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



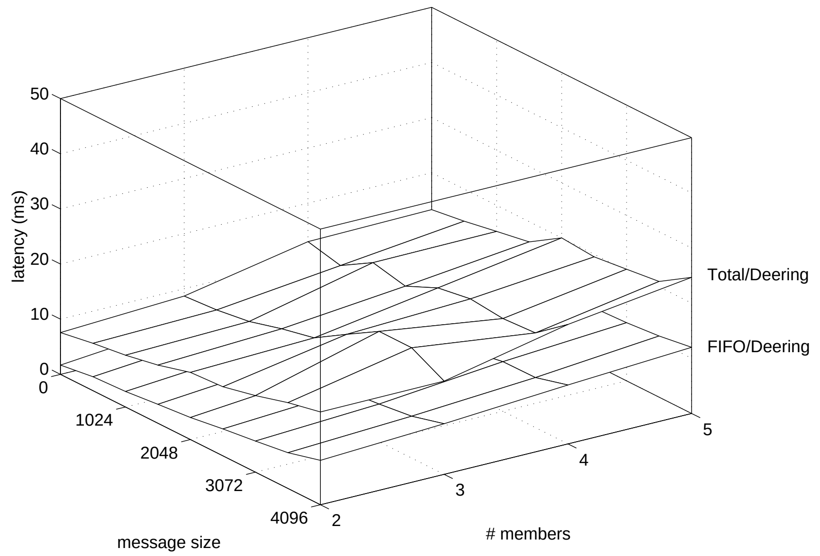


Figure 5: The top figure compares the one-way latency of FIFO Horus messages over straight UDP and UDP with the Deering hardware multicast extensions. The bottom figure compares the performance of total and FIFO order of Horus, both over UDP multicast.

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.

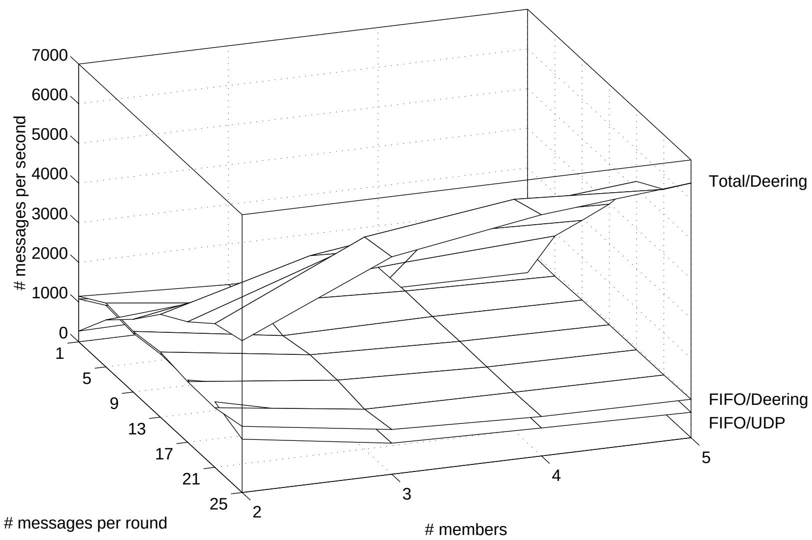


Figure 6: These graphs depict the message throughput for virtually synchronous, FIFO ordered communication over normal UDP and Deering UDP, and for totally ordering communication over Deering UDP.

6 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 ATM communication switch. For this purpose, we are running Horus in the application's address space, with I/O directly in and out of message buffers allocated by the application. Based on preliminary results (details will have to wait for a future paper), 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 extending Horus to support real-time features and an object-oriented environment.

Another area of research is security and privacy. In previous papers, we reported on a security architecture for Horus [6]. We are currently extending the architecture to address issues of privacy (anonymous communication), with the intention of eventually support general purpose tools for building robust applications that are also secure and private.

7 Conclusion

The Horus architecture supports group communication without imposing a specific semantics on groups or relying on a monolithic, all-or-nothing implementation. In this approach, it has been possible to provide a variety of reliability properties. Applications pay only for communication properties they use. Even when strong reliability properties are required, the system achieves very high performance. Looking to a future in which highly assured or "critical" applications will often have customized requirements, we anticipate growing demand for the type of flexibility and modularity stressed in the Horus design.

Acknowledgements

We gratefully acknowledge the important contributions of William Chan, Robert Cooper, Barry Gleeson, Rebecca Isaacs, Mike Reiter and Gautam Thaker. We also thank Yair Amir and Danny Dolev of the Transis project, from which we learned a great deal.

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

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