Horus: A Flexible Group Communications System

Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis[1]

Dept. of Computer Science, Cornell University

Submitted for publication in the Communication of the ACM

The emergence of process-group environments for distributed computing represents a promising step towards robustness for mission-critical distributed applications. Process groups have a "natural" correspondence with data or services that have been replicated for availability, or as part of a coherent cache. They can been used to support highly available security domains. And, group mechanisms fit well with an emerging generation of intelligent network and collaborative work applications.

Yet there is little agreement concerning how process groups should look or behave. The requirements that applications place on a group infrastructure can vary tremendously, and there may be fundamental tradeoffs between semantics and performance. Even the most appropriate way to present the group abstraction to the application depends on the setting.

This paper reports on the Horus system, which provides an unusually flexible group communication model to application-developers. This flexibility extends to system interfaces, the properties provided by a protocol stack, and even the configuration of Horus itself, which can run in user space, in an operating system kernel or microkernel, or be split between them.

Horus can be used through any of several application interfaces. These include toolkit-styled interfaces, but also interfaces that hide group functionality behind Unix communication system-calls, the Tk/Tcl programming language, and other distributed computing constructs. The intent is that it be possible to slide Horus beneath an existing system as transparently as possible, for example to introduce fault-tolerance or security without requiring substantial changes to the system being hardened [3].

Horus provides efficient support for the *virtually synchronous* execution model. This model was introduced by the Isis Toolkit [2], and has been adopted with some changes by such systems as Transis [1], Psync [11], Trans/Total [9], RMP [19], and Rampart [14]. The model is based on group membership and communication primitives, and can support a variety of fault-tolerant tools, such as for load-balanced request execution, fault-tolerant computation, coherently replicated data, and security.

Although often desirable, properties like virtual synchrony may sometimes be unwanted, introduce unnecessary overheads, or conflict with other objectives such as real-time guarantees. Moreover, the optimal implementation of a desired group communication property sometimes depends on the runtime environment. In an insecure environment, one might accept the overhead of data encryption, but wish to avoid this cost when running inside a firewall. On a platform like the IBM SP2, which has reliable message transmission, protocols for message retransmission would be superfluous.

Accordingly, Horus provides an architecture whereby the protocol supporting a group can be varied, at runtime, to match the specific requirements of its application and environment.

[1]This work was supported by grants from ARPA/ONR (N00014-92-J-1866), GTE, IBM, Schweizer Nationalfonds, and Siemens-AG. The Horus system is available at no fee for research use; see the Horus project page at **http://www.cs.cornell.edu/Info/Projects/HORUS** for details.

It does this using a structured framework for protocol composition, which incorporates ideas from systems such as the Unix "streams" framework [16] and the x-kernel [12], but replaces point-to-point communication with group communication as the fundamental abstraction. In Horus, group communication support is provided by stacking protocol modules that have a regular architecture, and in which each module has a separate responsibility. A process group can be optimized by dynamically including or excluding particular modules from its protocol stack.

Horus also innovates by introducing run-time configuration, group communication interfaces, full thread-safety, and supporting messages that may span multiple address spaces. Since Horus does not provide control operations, and has one single address format, protocol layers can be mixed and matched freely. In both streams and the x-kernel, the different protocol modules supply many different control operations, and design their own address format, both severely limiting such configuration flexibility[2] (see also [4]).

1 A layered process group architecture

We find it useful to think of Horus central protocol abstraction as resembling a Lego^{tm} block; the Horus "system" is thus like a "box" of Lego blocks. Each type of block implements a microprotocol that provides a different communication feature. To promote the combination of these blocks into macroprotocols with desired properties, 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). Obviously, not every sort of protocol block makes sense above or below every other sort. But the conceptual value of the architecture is that where it makes sense to create a new protocol by restacking existing blocks in a new way, doing so is straightforward [18].

Technically, each Horus protocol 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. Each layer is identified by an ASCII name, and registers its upcall and downcall-handlers at initialization time. There is a strong similarity between Horus protocol blocks and object classes in an object-oriented inheritance scheme, and readers may wish to think of protocol blocks as members of a class hierarchy.

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, control passes to the message send function in the layer below it. This repeats itself recursively until the bottommost block is reached and invokes a driver to actually send the message.

The specific layers currently supported by Horus solve such problems as interfacing the system to varied communication transport mechanisms, overcoming lost packets, encryption and decryption, maintaining group membership, helping a process that joins a group obtain

[2]We note that a follow-on to the x-kernel project, called Consul [10], has overcome some of these disadvantages by supporting sophisticated micro-protocols between protocol modules and providing extra support for group communication.

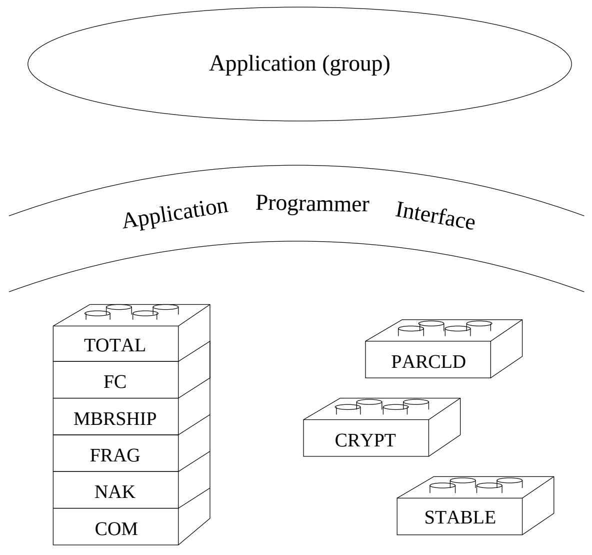


Figure 1: Group protocol layers can be stacked at run-time like Lego^{tm} blocks, and support applications through one of several application programmer interfaces.

the state of the group, merging a group that has partitioned, flow control, etc. (see sidebar). Horus also includes tools to assist in the development and debugging of new layers.

Each stack of blocks is carefully shielded from other stacks. It 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 monopolize the available memory. This is particularly important inside a kernel, or if one of the stacks has soft real-time requirements.

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. Depending on the application, it may correspond to a machine, a process, a thread, a socket, a port, and so forth. 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 *group object* is used to maintain the local protocol state on an endpoint. Associated with each group object is the *group address* to which messages are sent, and a *view*: a list of destination endpoint addresses that are believed to be accessible group members. Since a group object is purely local, Horus technically allows different endpoints to have different views of the same group. An endpoint may have multiple group objects, allowing it to communicate with different groups and views. A user can install new views when processes crash or recover, and can use one of several membership protocols to reach some form of agreement on views between multiple group objects in the same group.

Horus provides a large collection of microprotocols. Some of the most important ones are

COM – The COM layer provides the Horus group interface to such low-level protocols as IP, UDP, and some ATM interfaces.
NAK – This layer implements a negative acknowledgement based message retransmission protocol.
CYCLE – Multi-media message dissemination.
PARCLD – Hierarchical message dissemination.
FRAG – Fragmentation/reassembly.
MBRSHIP – This layer provides each member with a list of endpoints that are believed to be accessible. It runs a consensus protocol to provide its users with a *virtually synchronous* execution model.
FC – Flow control.
TOTAL – Totally ordered message delivery.
STABLE – This layer detects when a message has been delivered to all destination endpoints, and can be garbage collected.
CRYPT – Encryption/decryption.
MERGE – Location and merging of multiple group instances.

Table 1: PROPOSED SIDEBAR

The message object is a local storage structure. Its interface includes operations to push and pop protocol headers. Messages are passed from layer to layer by passing a pointer, and never need be copied.

A thread at the bottommost layer waits for messages arriving on the network interface. When a message arrives, the bottommost layer (typically COM) pops off its header, and passes the message on to the layer above it. This repeats itself recursively. If necessary, a layer may drop a message, or buffer it for delayed 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 depending on the scheduling policies used by the thread scheduler. Therefore, Horus numbers the messages, and uses *event count* synchronization variables [13] to reconstruct the order where necessary.

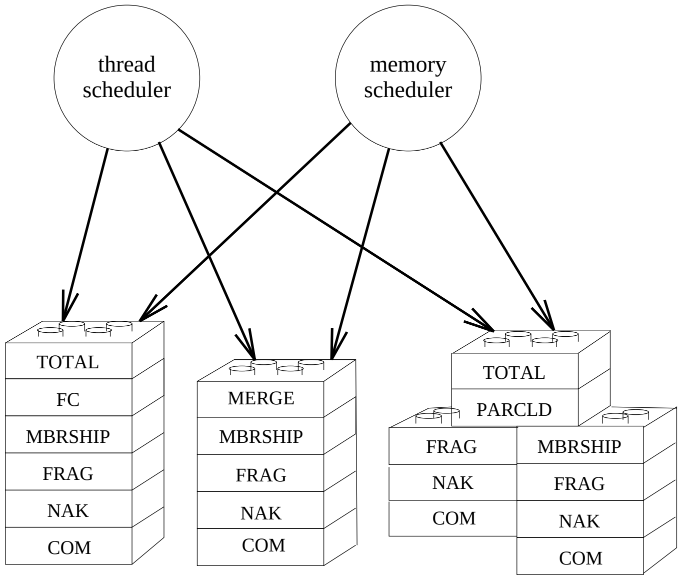


Figure 2: The Horus stacks are shielded from each other, and have their own threads and memory, each of which is provided through a scheduler.

2 Protocol stacks

The microprotocol architecture of Horus would not be of great value unless the various classes of process group protocols that we might wish to support can be simplified by being expressed as stacks of layers, perform well, and share significant functionality. Our experience in this regard has been very positive [18].

For example, the stacks shown in Figure 2 all implement virtually synchronous process groups. The left-most stack provides totally ordered, flow-controlled communication over the group membership abstraction. The layers FRAG, NAK and COM respectively break large messages into smaller ones, overcome packet loss using negative acknowledgements, and interface Horus to the underlying transport protocols. The adjacent stack is similar, but provides weaker ordering and includes a layer that supports "state transfer" to a process joining a group, or when groups merge after a network partition. To the right is a stack that supports scaling through a hierarchical structure, in which each "parent" process is responsible for a set of "child" processes. The dual stack illustrated in this case represents a feature whereby a message can be routed down one of several stacks, depending on the type of processing required. Additional protocol blocks provide functionality such as data encryption, packing small messages for efficient communication, isochronous communication (useful in multimedia systems), etc.

For Horus layers to fit like Lego blocks, they each must provide the same downcall and upcall interfaces. A lesson learned from the x-kernel [12] is that if the interface is not rich enough, extensive use will be made of general purpose control operations (similar to *ioctl*), which reduces configuration flexibility. (Since the control operations are unique to a layer, the Lego blocks would not "fit" as easily.) The *Horus Common Protocol Interface* (HCPI) therefore supports an extensive interface that supports all common operations in group communication systems, going beyond the functionality of earlier layered systems such as the *x*-kernel, Furthermore, the HCPI is designed for multiprocessing, and is completely asynchronous and reentrant.

Broadly, the HCPI interfaces fall into two categories. Those in the first group are concerned with sending and receiving messages, and the stability of messages.[3] The second category of Horus operations are concerned with membership. In the down direction, they let an application or layer control the group membership used by layers below it. As upcalls, these report membership changes, communication problems, and other related events to the application.

While supporting the same HCPI, each Horus layer runs a different protocol, each implementing a different property. 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. These constraints have been tabulated. Given information about the properties of the network transport service, and the properties provided by the application, it is often possible to automatically generate the minimal protocol stack that achieves a desired property [18].

Layered protocol architectures sometimes perform poorly. Traditional layered systems impose an order on which protocols process messages, limiting opportunities for optimization, and imposing excessive overhead. Clark and Tennenhouse have suggested that the key to good performance rests in *Integrated Layer Processing* (ILP) [4]. Systems based on the ILP principle avoid inter-layer ordering constraints, and can perform as well as monolithically structured systems. Horus is consistent with ILP: there are no intrinsic ordering constraints on processing, so unnecessary synchronization delays are avoided.

3 Using Horus to build a robust groupware application

Earlier, we commented that Horus can be hidden behind standard application programmer interfaces. A good illustration of how this is done arose when we interfaced the Tcl/TK graphical programming language to Horus.

A challenge posed by running systems like Horus side by side with a package like X-windows or Tcl/TK is that such packages are rarely designed with threads or Horus communication stacks in mind. To avoid a complex integration task, we therefore chose to run Tcl/TK as a separate thread in an address space shared with Horus. Horus intercepts certain system calls issued by Tcl/TK (see Figure 3), such as the Unix *open* and *socket* system calls. We call this mechanism an *intercept proxy.* The proxy redirects the system calls, invoking Horus functions which will create Horus process groups and register appropriate protocol

[3]It is common to say that a message is *stable* when processing has completed and associated information can be garbage collected. Horus standardizes the handling of stability information, but leaves the actual semantics of stability to the user. Thus, an application for which stability means "logged to disk" can share this Horus functionality with an application for which stability means "displayed on the screen."

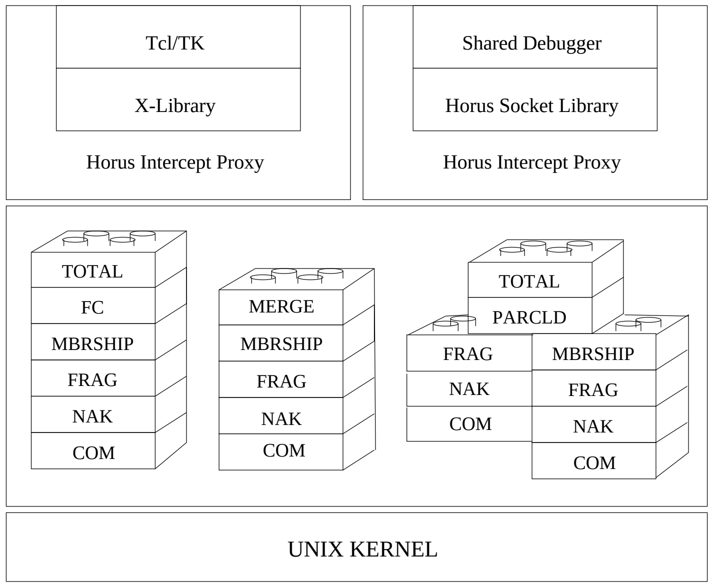


Figure 3: Unix system calls can be intercepted by Horus using *intercept proxies.* These allow the implementation of new socket domains in user space, and permit us to link thread-unsafe applications with the Horus system.

stacks at run time. Subsequent I/O operations on these group I/O sockets are mapped to Horus communication functions.

To make Horus accessible within Tcl applications, two new functions were registered with the Tcl interpreter. One creates endpoint objects, and the other creates group addresses. The endpoint object itself can create a group object using a group address. Group objects are used to send and receive messages. Received messages result in calls to Tcl code, which typically interpret the message as a Tcl command. This yields a powerful framework: a distributed, fault-tolerant, whiteboard application can be built using only eight short lines of Tcl code, over a Horus stack of seven protocols.

To validate our approach, we ported a sophisticated Tcl/TK application to Horus. The Continuous Media Toolkit (CMT) [17] is a Tcl/TK extension that provides objects that read or output audio and video data. These objects can be linked together in pipelines, and are synchronized by a *logical timestamp* object. This object may be set to run slower or faster than the real clock, or even backwards. This allows stop, slow motion, fast forward, and rewind functions to be implemented.

Architecturally, CMT consists of a multi-media server process that multicasts video and audio to a set of clients. We decided to replicate the server using a primary-backup approach. where the backup servers stand by to back up failed or slow primaries.

The original CMT implementation depends on extensions to Tcl/TK. These implement a master/slave relationship between the machines, provide for a form of logical timestamp synchronization between them, and support a real-time communication protocol called Cyclic UDP. The Cyclic UDP implementation consists of two halves, a sink object that accepts

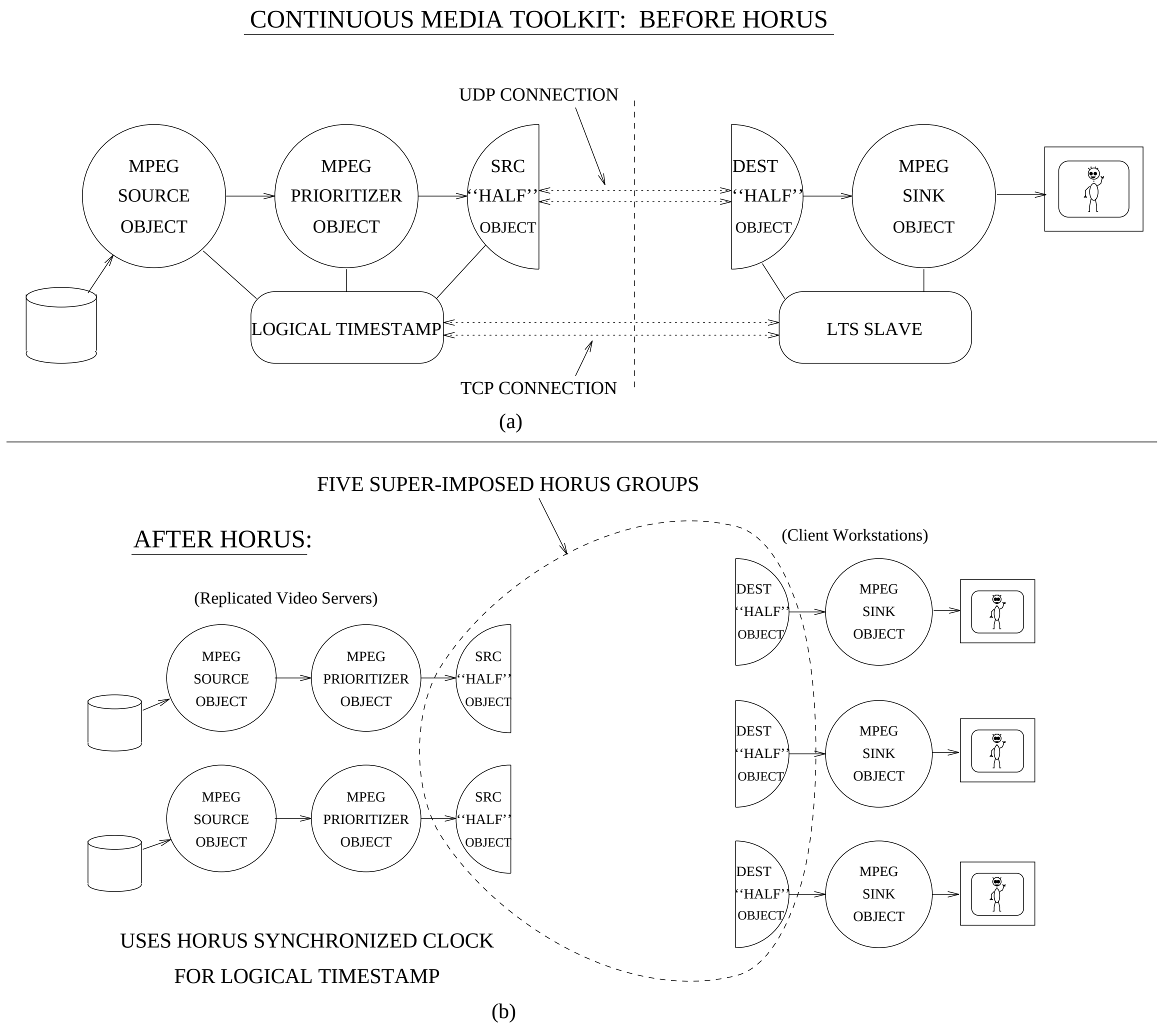


Figure 4: This figure shows an example of a video service implemented using the Continuous Media Toolkit. MPEG is a video compression standard. In (a), a standard, fault intolerant set-up is depicted. In (b), Horus was used to implement a fault-tolerant version that is also able to multicast to a set of clients.

multi-media data from another CMT object, and a source object that produces multi-media data and passes it on to another CMT object (see Figure 4a). The resulting system is distributed but intolerant of failures, and does not allow for multicast.

Using Horus, it was straightforward to extend CMT with fault-tolerance and multicast capabilities. Five Horus stacks were required. One of these is hidden from the application, and implements a probabilistic clock synchronization protocol [5]. It uses a Horus layer called MERGE to ensure that the different machines will find each other automatically (even after network partitions), and employs the virtual synchrony property to rank the processes, assigning the lowest ranked machine to maintain a master clock on behalf of the others. The second stack synchronizes the speeds and offsets with respect to real-time of the logical timestamp objects. To keep these values consistent, it is necessary that they be updated in the same order. Therefore, this stack is similar to the previous one, but includes a Horus protocol block that places a total order on multicast messages delivered within the group.[4] The third tracks the list of servers and clients. Using a deterministic rule based on the process ranking maintained by the virtual synchrony layer, one server decides to multicast the video, and one server, usually the same, decides to multicast the audio. This set-up is shown in Figure 4b.

To disseminate the multi-media data, we used two identical stacks, one for audio and one for video. The key component in these is a protocol block that implements a multi-media generalization of the Cyclic UDP protocol. The algorithm is similar to FRAG, but will reassemble messages that arrive out of order, and drop messages with missing fragments (*cf.* Application-Level Framing [4, 6]).

One might expect that a huge amount of recoding would have been required to accomplish these changes. However, all of the necessary work was completed using 42 lines of Tcl code. An additional 160 lines of C code supports the CMT frame buffers in Horus. Two new Horus layers were needed, but were developed by adapting existing layers; they consist of 1800 lines of C code and 300 lines, respectively (ignoring the comments and lines common to all layers). Thus, with relatively little effort and little code, a complex application written with no expectation that process group computing might later be valuable was modified to exploit Horus functionality.

4 Electra

The introduction of process groups into CMT required sophistication with Horus and its intercept proxies. Many potential users would lack the sophistication and knowledge of Horus required to do this, hence we recognized a need for a way to introduce Horus functionality in a more transparent way. This goal evokes an image of "plug and play" robustness, and leads one to think in terms of an object-oriented approach to group computing.

The Common Object Request Broker Architecture (CORBA) is emerging as a major standard for supporting object-oriented distributed environments. Object-oriented distributed applications that comply with CORBA can invoke one-another's methods with relative ease. Our work resulted in a CORBA compliant interface to Horus, which we call Electra [8]. Electra can be used without Horus, and vice versa, but the combination represents a more complete system.

In Electra, applications are provided with ways to build Horus process groups, and to directly exploit the virtual synchrony model. Moreover, Electra objects can be aggregated to form "object groups," and object references can be bound to both singleton objects and object groups. An implication of the interoperability of CORBA implementations is that Electra object groups can be invoked from *any* CORBA-compliant distributed application, regardless of the CORBA platform on which it is running, without special provisions for

[4]This protocol differs from the *Total* protocol in the Trans/Total[9] project in that the Horus protocol only rotates the token among the current set of senders, while the Trans/Total protocol rotates the token among all members.

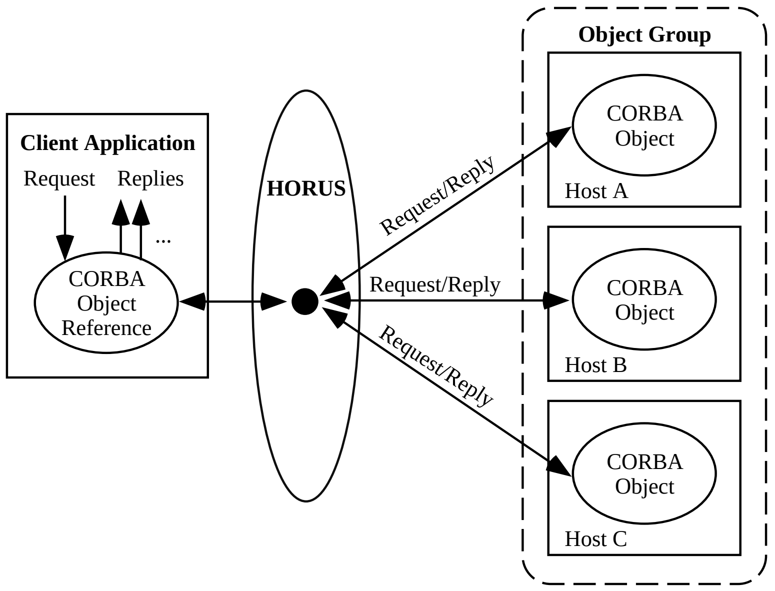


Figure 5: Object group communication in Electra.

group communication. This means that a service can be made fault-tolerant without changing its clients.

When a method invocation occurs within Electra, object-group references are detected and transformed into multicasts to the member objects (see Figure 5). Requests can be issued either in transparent mode, where only the first arriving member reply is returned to the client application, or in non-transparent mode, permitting the client to access the full set of responses from individual group members. The transparent mode is used by clients to communicate with replicated CORBA objects, while non-transparent mode is employed with object groups whose members perform different tasks. Clients submit a request either in a synchronous, asynchronous, or deferred-synchronous way.

Our work on Electra shows that group programming can be integrated in a natural, transparent way with popular programming methodologies. To the degree that process-group computing interfaces and abstractions represent an impediment to their use in commercial software, technologies such as Electra suggest a possible middle ground, in which fault-tolerance, security, and other group-based mechanisms can be introduced late in the design cycle of a sophisticated distributed application.

5 Performance

A major concern of our architecture is the overhead of layering, hence we now focus on this issue. We present the overall performance of Horus on a system of SUN Sparc10 workstations running SunOS 4.1.3, communicating through a loaded Ethernet. We used two network transport protocols: normal UDP, and UDP with the Deering IP multicast extensions (shown as "Deering").

To highlight some of the performance numbers: we achieve a one-way latency of 1.2 msecs over an unordered virtual synchrony stack (over ATM, it is currently 0.7 msecs), and, using a totally ordered layer over the same stack, 7,500 1-byte messages per second. 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 Flow-Control layer, which buffers heavily using the "message list" capabilities of Horus [7]. We easily reach the Ethernet 1007 Kbytes/second maximum bandwidth with a message size smaller than 1 kilobyte.

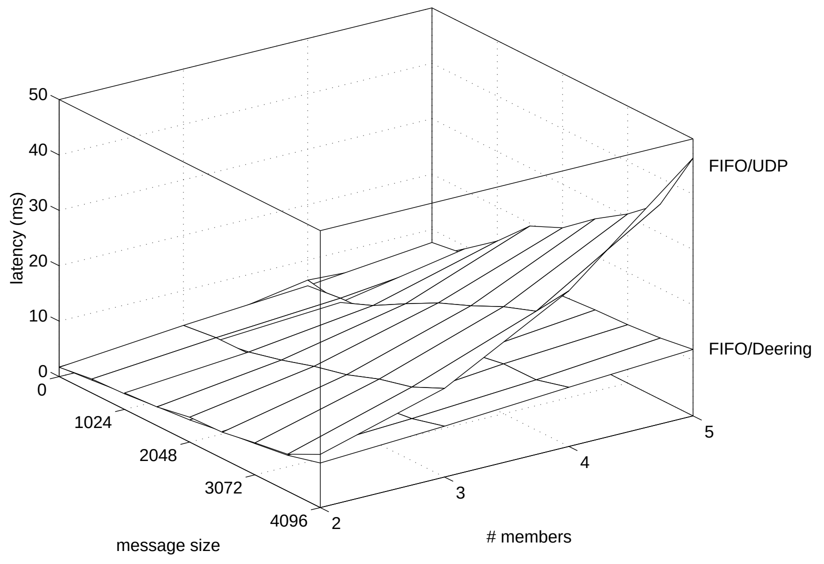
Our performance test program has each member do exactly the same thing: send k messages and wait for $k \times (n-1)$ messages of size s, where n is the number of members. This way we simulate an application that imposes a high load on the system while occasionally synchronizing on intermediate results.

Figure 6 depicts the one-way communication latency of 1-byte Horus messages. As can be seen in the top graph, hardware multicast is a big win, especially when the message size goes up. 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. A problem with the totally ordered layer is that it can be inefficient when senders send single messages at random, and with a high degree of concurrent sending by different group members. With just one sender, the one-way latency drops to 1.6 milliseconds.

Figure 7 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. For totally ordered communication we see that the throughput becomes better if we send more messages per round (because of increased concurrency). 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.

6 Ongoing work

Although the initial version of Horus is nearing completion, significant challenges remain. Broadly, we are interested in moving Horus to more advanced platforms, such as stripped-down computing nodes linked by ATM. 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, we anticipate that this configuration of the system will expand our application domain to parallel computing, high performance I/O servers, multi-media, and computer-supported collaborative work. To enable these new types of applications, we are extending Horus to support real-time features, and are cooperating with the Transis project at the Hebrew University to develop a group security architecture and general purpose tools for building robust applications that are also secure and private [15].



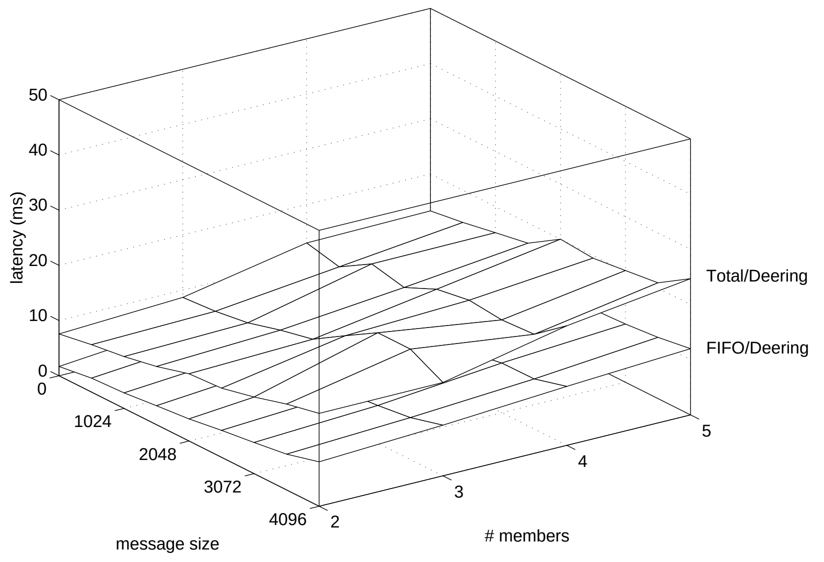


Figure 6: The top figure compares the one-way latency of 1-byte FIFO Horus messages over straight UDP and UDP with the Deering IP multicast extensions. The bottom figure compares the performance of total and FIFO order of Horus, both over UDP multicast.

Acknowledgements

We gratefully acknowledge the contributions of Yair Amir, William Chan, Robert Cooper, Flaviu Christian, Danny Dolev, Roy Friedman, Sophia Georgiakaki, Brad Glade, Barry Gleeson, Katie Guo, Takako Hickey, Rebecca Isaacs, Dahlia Malkhi, Michael Melliar Smith, Louise Moser, David Powell, Mike Reiter, Andre Schiper, Gautam Thaker, Alexey Vaysburd, and Werner Vogels.

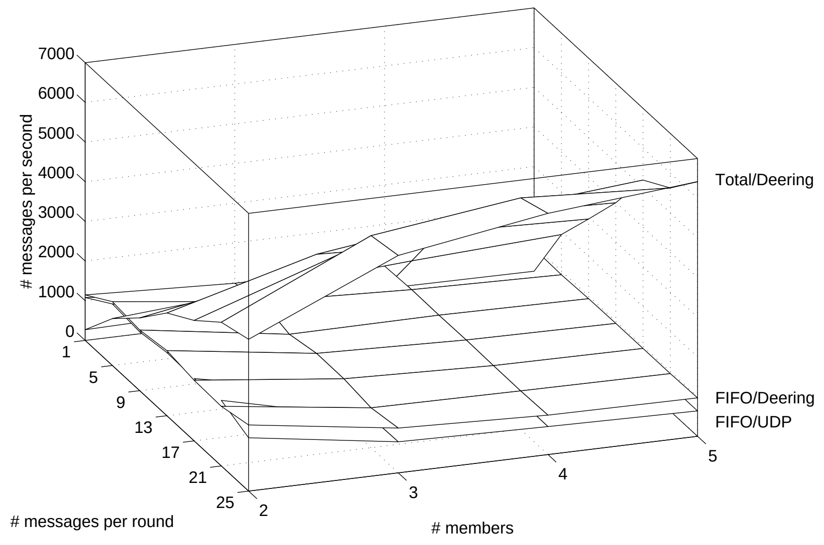


Figure 7: These graphs depict the message throughput for virtually synchronous, FIFO ordered communication over normal UDP and Deering UDP, and for totally ordering communication over Deering UDP.

References

[1] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication subsystem for high availability. In *Proc. of the Twenty-Second Int. Symp. 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] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. In *Proc. of the Fifteenth ACM Symp. on Operating Systems Principles*, Copper Mountain Resort, CO, December 1995.

[4] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. of the '90 Symp. on Communications Architectures & Protocols*, pages 200–208, Philadelphia, PA, September 1990. ACM SIGCOMM.

[5] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[6] Sally Floyd, Van Jacobson, Steven McCanne, Ching-Gung Liu, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proc. of the '95 Symp. on Communications Architectures & Protocols*, Cambridge, MA, August 1995. ACM SIGCOMM.

[7] Roy Friedman and Robbert van Renesse. Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. Technical Report 94-1527, Cornell University, Dept. of Computer Science, July 1995. Submitted to IEEE Transactions on Networking.

[8] Silvano Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies*, Monterey, CA, June 1995. USENIX.

[9] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1, Jan 1990.

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

[11] Larry L. Peterson, Nick C. Bucholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

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

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

[14] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, November 1994.

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

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

[17] Lawrence A. Rowe and Brian C. Smith. Continuous media player. In *Proc. of the Third Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, San Diego, CA, Nov 12-13 1992.

[18] Robbert Van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A Framework for Protocol Composition in Horus. In *Proc. of the Fourteenth ACM Symp. on Principles of Distributed Computing*, pages 80–89, Ottawa, Ontario, August 1995. ACM SIGOPS-SIGACT.

[19] B. Whetten. A Reliable Multicast Protocol. In K.P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes on Computer Science*. Springer-Verlag, July 1995.