Horus: A Flexible Group Communications System

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

Dept. of Computer Science, Cornell University

1 A layered process group architecture

If a broad definition of "process group" is accepted, process group structures can be identified in a great number of distributed systems and protocols. Groups of processes 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. The V [4] and Chorus [1] operating systems have notions of group services. Server groups are often found in fail-over architectures, where a request to a failed server is automatically reissued to an operational one. The AAS system, proposed by IBM for a next generation air traffic control application, was based on real-time process groups.

Many recent process-group systems, including Isis, Transis, Psync [12], Totem [10], RMP [20], and Rampart [15], provide a *virtual synchrony* execution model. The approach can support a variety of fault-tolerant tools, such as for load-balanced request execution, fault-tolerant computation, coherently replicated data, and security. Unfortunately, however, such systems are often rigid both in terms of what a process group "means", and how it can be accessed. Applications must be designed with process groups in mind, hence older applications that use standard distributed computing environments are denied the benefits of the process group mechanisms.

This paper reports on the Horus project, 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 a operating system kernel or microkernel, or be split between them.

Horus is used through an *interface proxy*. Proxies can support toolkits that use groups explicitly, or hide groups beneath parallel programming back-ends (for example, the Panda subsystem of the Orca language [2], and IBM's PCODE subsystem for MPI). Finally, we have developed a proxy that *intercepts* certain classes of system calls and maps them into Horus operations, for example to provide security or fault-tolerance features transparently [3]. This proxy has also been used to embed Horus into Tcl/TK and Python, both popular prototyping languages.

Different uses of groups create different requirements on the membership and communication functionality of the system. Some properties (like virtual synchrony) may impose undesirable overheads, or even be incompatible, with other properties (like real-time communication). Moreover, the mechanisms required to implement a desired group communication property depend on the runtime environment. In an insecure environment, one might accept the overhead of data encryption, but wish to avoid this cost when running within a firewall.

[1]This work was supported by grants from ARPA/ONR (N00014-92-J-1866), the Schweizer Nationalfonds, and GTE Corporation.

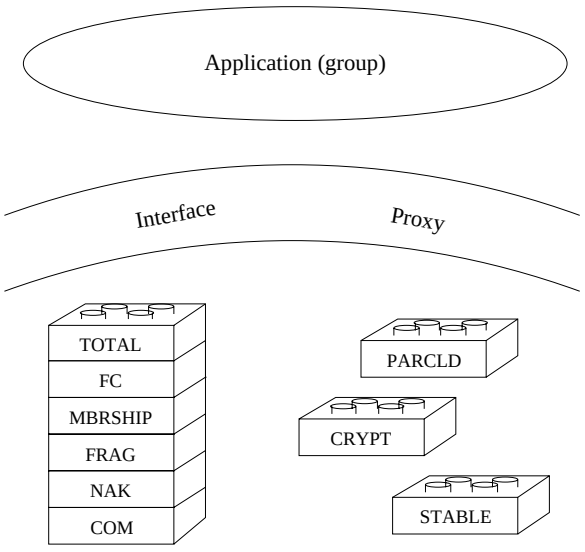

Figure 1: Group protocol layers can be stacked at run-time like Lego^{tm} blocks, and support a variety of applications through a convenient "interface proxy."

On a platform like the IBM SP2, which has reliable message transmission, protocols for message retransmission would be superfluous.

These observations motivate an architecture in which the protocol supporting a given group can be specified at runtime to match the application and runtime environment. On the other hand, it makes no sense to build a large collection of protocols that differ in small ways. More reasonable would be to factor out common functions, reusing code when possible.

We find it useful to think of the central protocol abstraction that emerges from this view as resembling a Lego^{tm} block; the Horus "system" is thus like a "box" of Lego blocks. Each type of block would be a microprotocol that implements 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.

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 (types) in a type-inheritance scheme, and readers may wish to think of protocol blocks as members of a type hierarchy.

To use Horus, an application that creates or attaches itself to a group specifies the stack of layers required for its purposes. 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

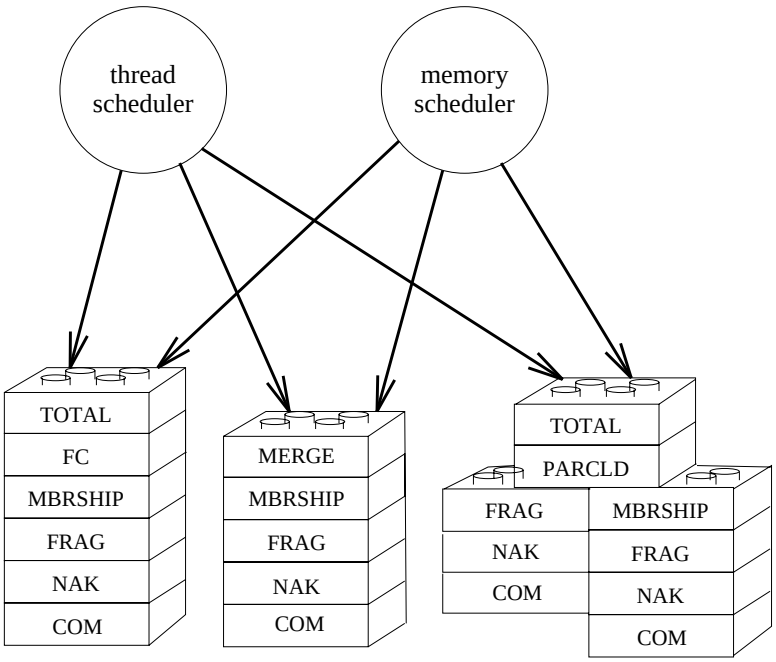

Figure 2: The Horus stacks are shielded from each other, and have their own threads and memory, each of which is provided through a scheduler.

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 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.

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 the state of the group, merging a group that has partitioned, flow control, etc. 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 hog the available memory. This is particularly important if running 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. 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.

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. Since a group object is purely local, Horus 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 as necessary.

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. As necessary, a layer may drop a message, or 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 [14] to reconstruct the order where necessary.

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.

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: each member of a group of "parent" process is responsible for a set of "child" processes. 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 is that if the interface is not rich enough, extensive use will be made of general purpose control operations (a la *ioctl*), which reduces configuration flexibility. (Since the control operations are unique to a layer, the Lego blocks do not "fit" as easily.) The *Horus Common Protocol Interface* (HPCI) therefore supports an extensive interface, which goes beyond the functionality of earlier layered systems such as *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.[2] 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, 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 [19].

Layered protocol architectures sometimes perform poorly. Clark and Tennenhouse have suggested that the key to good performance rests in *Integrated Layer Processing* (ILP) [5]. Traditional layered systems impose an order on in which protocols process messages, limiting opportunities for optimization, and imposing excessive overhead. 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.

3 Using Horus to build a robust groupware application

Earlier, we commented that Horus can be hidden behind standard 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. A Horus interface proxy is used to intercept certain system calls issued by Tcl/TK (see Figure 3), such as the UNIX *open* and *socket* system calls. These can then be redirected to Horus functions capable of establishing process groups and registering an appropriate protocol stack 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

[2] 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."

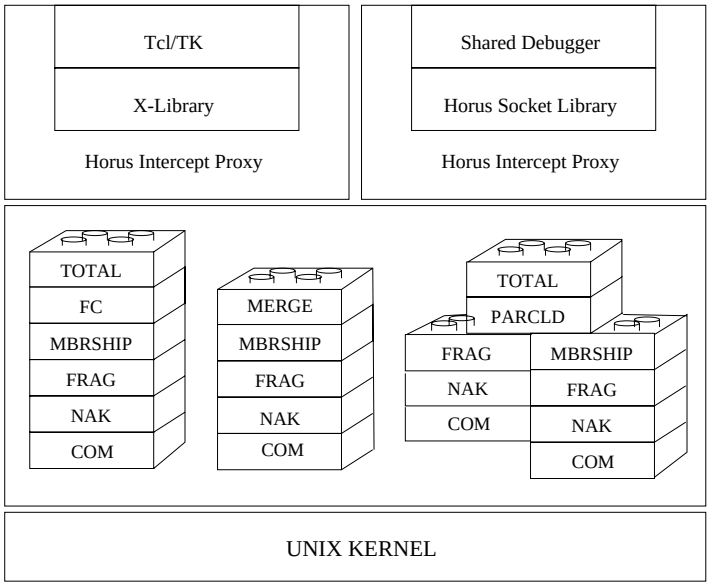

Figure 3: UNIX system calls can be intercepted by Horus using a concept called *intercept proxy*. These allow the implementation of new socket domains in user space, and permit us to safely link thread-unsafe applications with the Horus system.

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 prove our approach, we ported a sophisticated Tcl/TK application to Horus. The Continuous Media Toolkit (CMT) [18], is a Tcl/TK extension that provides objects that read or output audio and video data. These objects can be linked together in pipe-lines, and are synchronized by a *logical timestamp* (LTS). LTS is an object which represents time. It 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 one of the servers is responsible for video, the same, or possibly another server is responsible for audio, and other servers stand by to back up failed or slow primaries.

The original CMT implementation depends on extensions to Tcl/TK which support UDP and TCP communication. CMT uses TCP to losely synchronize LTS objects on different machines, assigning one to be master and others to be slaves. Obviously, this approach is intolerant of faults. Over UDP, the CMT toolkit supports a protocol, called Cyclic UDP, which implements multi-media communication over standard UDP. The Cyclic UDP object consists of two halves, a sink object that accepts 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 implementation does not allow for multicast.

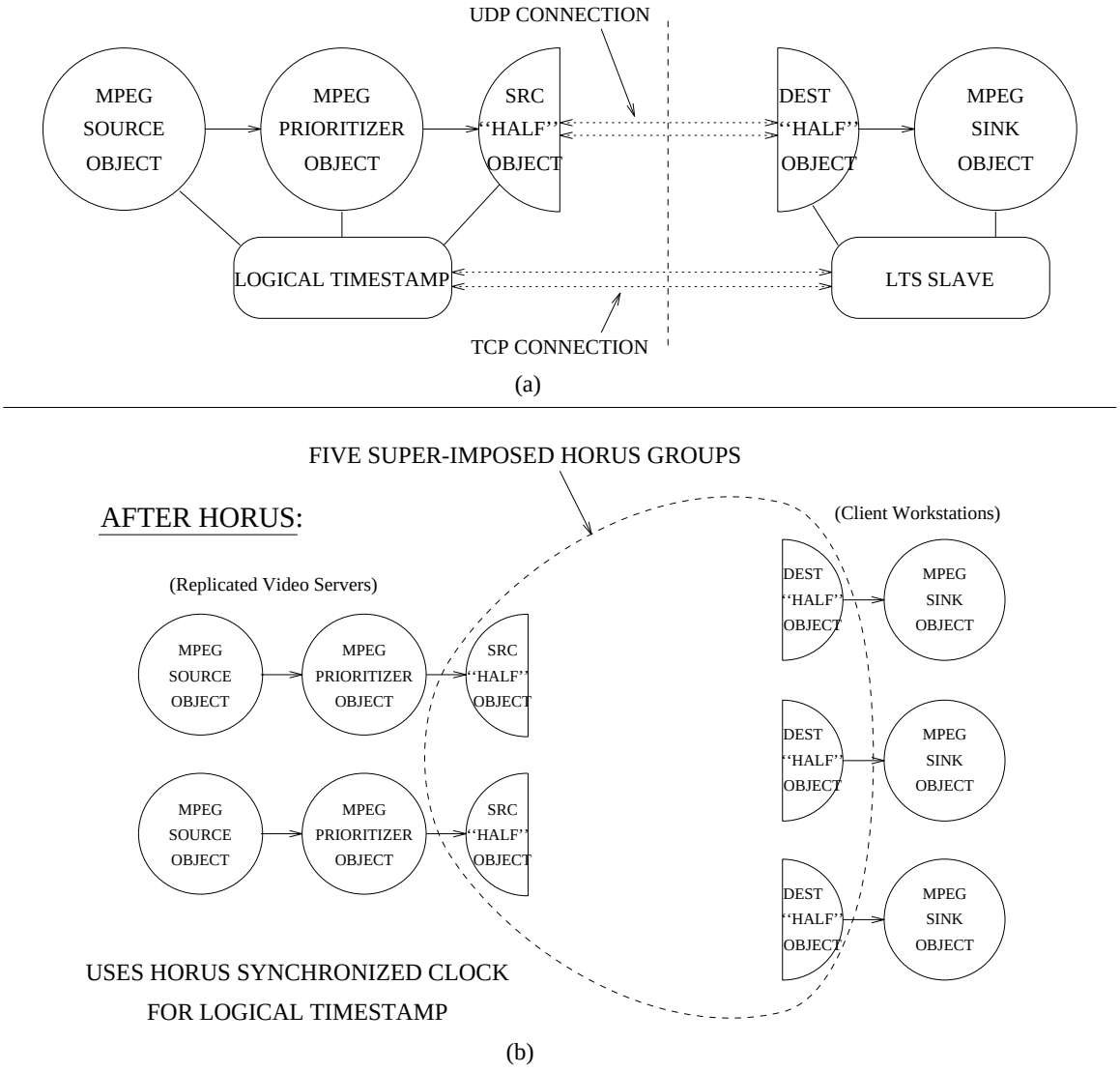

Figure 4: This figure shows an example of a video service implemented using the Continuous Media Toolkit. In (a), a standard, fault intolerant set-up is depicted. In (b), Horus was used to implement a fault-tolerant version that is also able to multicast to a set of clients.

Using Horus, it was straightforward to extend CMT with fault-tolerance and multicast capabilities. Five Horus stacks were required. One of these, MERGE:MBRSHIP:FRAG:NAK:COM, is hidden from the application, and implements a probabilistic clock synchronization protocol [6]. The MERGE protocol is necessary so that the different machines find each other automatically (even after network partitions), while the MBRSHIP protocol assigns one machine to be the master clock.

The second synchronizes the speeds and offsets with respect to real time of the LTS objects. To keep these values consistent, it is sufficient that they are updated in the same order. Therefore, the stack is similar to the previous, but includes the TOTAL protocol: MERGE:TOTAL:MBRSHIP:FRAG:NAK:COM.

The third one, PARCLD:MERGE:MBRSHIP:FRAG:NAK:COM, keeps track of who are the servers and the clients. Using a deterministic rule based on the ranks that the MBRSHIP layer assigns to each server process, 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: NFRAG:NNAK:COM. NNAK implements a multi-media generalization of the Cyclic UDP protocol. NFRAG is similar to FRAG, but will reassemble messages that arrive out of order, and drop messages with missing fragments (*cf.* Application-Level Framing [5, 7]).

From the description of these changes, one might expect that a huge amount of recoding would have been required. 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. The NNAK layer consists of 1800 lines of C code (ignoring the boilerplate lines), while the NFRAG layer consists of 300 lines of C code. 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 interface proxies. Throughout the preceding sections, object-oriented computing themes have surfaced repeatedly. It was therefore natural for us to ask if object-oriented software tools might be used to support a "plug and play" 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 V.2 compliant interface to Horus, which we call Electra [9]. Electra can be used without Horus, and vice versa, but the combination represents a more complete system.

Electra is a CORBA Object Request Broker (ORB). 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 CORBA 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, running on any vendor's ORB, without special provisions for group communication. This means that a service can be made fault-tolerant without changing its clients, a form of transparency lacking in most work on fault-tolerance using process groups.

When a method invocation occurs within Electra, the ORB detects object references bound to a group, multicasting such requests to the member objects (see Figure 5). Requests

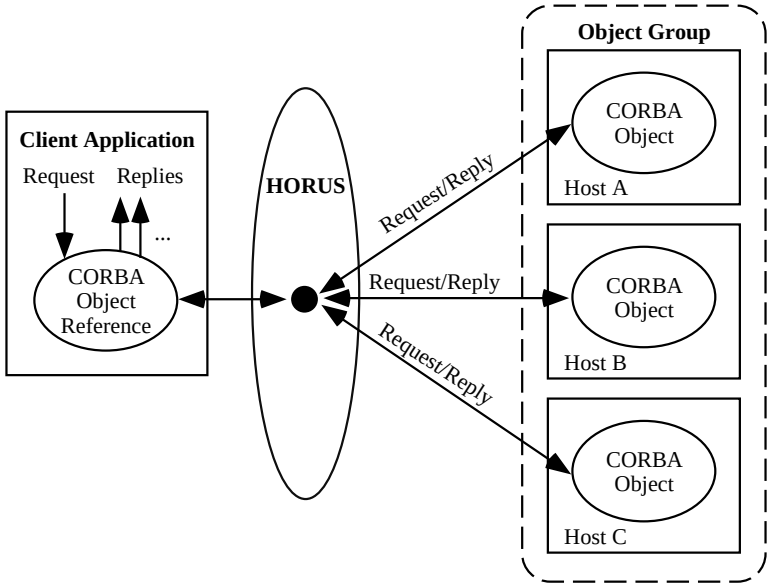

Figure 5: Object group communication in Electra.

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 methologies. 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.

To highlight some of the performance numbers: we achieve a one-way latency of 1.2 msecs over the MBRSHIP:FRAG:NAK:COM:udp stack (over ATM, it is currently 0.7 msecs), and, using TOTAL:MBRSHIP:FRAG:NAK:COM:udp, 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 layer, which buffers

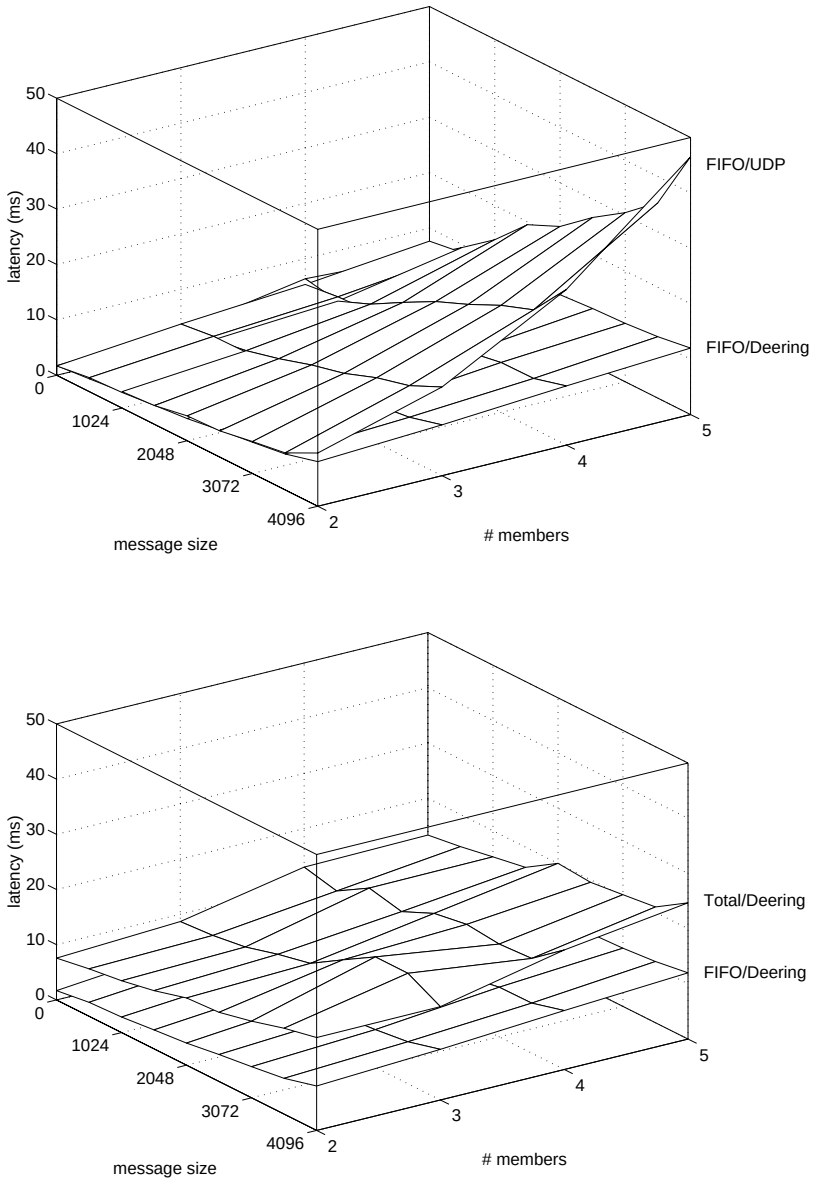

Figure 6: The top figure compares the one-way latency of 1-byte FIFO Horus messages over straight UDP and UDP with the Deering IP multicast extensions. The bottom figure compares the performance of total and FIFO order of Horus, both over UDP multicast.

heavily using the "message list" capabilities of Horus [8]. 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.

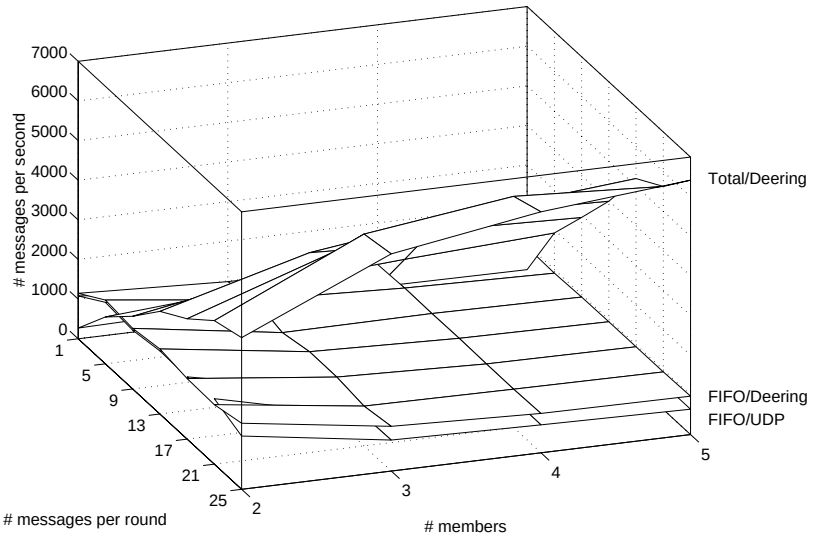

Figure 7: These graphs depict the message throughput for virtually synchronous, FIFO ordered communication over normal UDP and Deering UDP, and for totally ordering communication over Deering UDP.

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. In case of only one sender, the one-way latency is only 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 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.

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 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 Hebrew University to develop a group security architecture and general purpose tools for building robust applications that are also secure and private [16].

7 Related Work

We are not the first to realize that a structured framework can benefit the designers of communication protocols and systems. The best-known framework for composing a set of protocols is the STREAMS framework [17]. This work does not support group communication and has limited opportunities for concurrency. A related but more sophisticated approach is used in the *x*-kernel [13]. Horus was motivated by ideas from *x*-kernel, but with group communication as the fundamental abstraction. *x*-kernel is designed mainly for point-to-point communication, and configuration is done at compile-time.

Horus improves on this work by providing 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, layers can be mixed and matched. In both STREAMS and the x-kernel, the different protocol modules supply many different control operations, and design their own address format, both severely limiting such configuration flexibility (see also [5]). We note that a follow-on to the *x*-kernel project, called Consul [11], is attempting to deal with some of these disadvantages by supporting sophisticated micro-protocols between protocol modules.

Acknowledgements

We gratefully acknowledge the contributions of Yair Amir, William Chan, Robert Cooper, Danny Dolev, Roy Friedman, Sophia Georgiakaki, Brad Glade, Barry Gleeson, Katie Guo, Takako Hickey, Rebecca Isaacs, Dalia Malki, Mike Reiter, Gautam Thaker, Alexey Vaysburd, and Werner Vogels.

References

[1] François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier. Revolution 89 or Distributing UNIX brings it back to its original virtues. Technical Report CS/TR-89-36.1, Chorus systèmes, 6 Avenue Gustave Eiffel, F-78182, Saint-Quentin-en-Yvelines, France, August 1989.

[2] Henri Bal, Frans Kaashoek, and Andrew Tanenbaum. Experience with distributed programming in Orca. In *Proc. of the Int. Conf. on Computer Languages '90.* IEEE, 1990.

[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, Colorado, December 1995.

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

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

[7] 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 Symp. on Communications Architectures & Protocols*, Cambridge, MA, August 1995. ACM SIGCOMM.

[8] Roy Friedman and Robbert van Renesse. Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. Technical Report 94-????, Cornell University, Dept. of Computer Science, July 1995.

[9] 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.

[10] 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.

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

[12] Larry Peterson. Preserving context information in an IPC abstraction. In *Sixth Symp. on Reliability in Distributed Software and Database Systems*, pages 22–31. IEEE, March 1987.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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*, Ottawa, Ontario, August 1995. ACM SIGOPS-SIGACT.

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