# Performance of the Horus Asynchronous Group Communication System under High Load

Takako M. Hickey and Robbert van Renesse *

Dept. of Computer Science
Upson Hall
Cornell University
Ithaca, NY 14853

**Abstract**

Horus is an asynchronous group communications system designed to support a variety of distributed and parallel applications. This paper reports the performance of continuous $n \times n$ multicasts for a Horus protocol stack that provides fully reliable communication over Ethernet supporting IP multicast. This communication pattern not only is highly demanding from the communications point of view, but also is common for an important class of distributed applications.

Specific findings reported in this paper are: providing reliability is expensive under high loss; the cost of providing full reliability is modest compared to that of providing simple reliability; the lack of receivers' resources contributes to high loss; the lack of synchronization among group members also contributes to high loss; and controlling the send rate based on receivers' resource availability is a promising method for maintaining high throughput under high load.

## 1 Introduction

Programming a distributed application can be tedious. Programmers may have to deal with different types of machines and networks that have different interfaces. Programming a distributed application is tricky. Programmers have to worry about getting synchronization correct among different application components and about having all necessary components, which may be scattered on different machines, up and running. In a fault-tolerant setting, not only must programmers worry about having individual components detect failures of other components, programmers may have to ensure that all components agree on which members have failed—all this in the face of transient failures and/or network partitions.

Group communications systems aim to simplify the task of distributed programming. They identify functionality central to distributed programming and provide their efficient implementation. For example, this functionality includes the failure detection problem described above, as well as membership, virtual synchrony, and causal and total ordering of messages (see, for example, [Bir93]).

*The work reported was supported in part by ARPA/ONR grant N00014-92-J-1866, the National Science Foundation under Grant No. CCR-9003440, DARPA/NSF Grant No. CCR-9014363, NASA/DARPA grant NAG-2-893, AFOSR grant F49620-94-1-0198, and Bellcore support for doctoral education program.

Horus is an asynchronous group communications system that is designed to support a variety of distributed and parallel applications [vRBF+95]. The key to Horus is layering. Different layers understand the same interface but implement a different semantics. Stacking multiple layers results in the same interface but different semantics. Each layer typically implements a small amount of semantics to increase maintainability and reusability. Different applications can use different stacks of layers as appropriate for their needs. An application can avoid paying a performance penalty associated with unwanted functionality by stacking only those layers that provide the needed semantics. Since different stacks have the same interface, a single application can also experiment with different semantics or even offer different semantics to different members. Thus, Horus allows applications with different semantics and performance needs to coexist in a single system. To minimize the overhead of layering, Horus also allows integration of processing at multiple layers [CT90]. Bhatti and Schlichting reports another high level communications system that offers different services to different applications [BS95].

This paper reports performance of a protocol stack for the Horus group communications system that provides fully reliable delivery under high load over Ethernet supporting IP multicast [Dee89]. The fully reliable stack is a useful base for many group communication protocols. It provides group membership in addition to traditional reliable communications. In reliable communications, reliability is provided by retransmitting messages that are lost. To be fully reliable, all messages that can potentially be lost must be buffered. This requires the knowledge of potential receivers—or group membership. Otherwise, the message log will grow indefinitely. Traditional reliable communications work skirts around this issue by making an arbitrary decision about when to discard messages [HSC95, FJM+95]. This suffices for messages that lose value as time passes (e.g, video broadcasting), but not all messages are of this type (e.g., banking systems).

In our experiments, high load was generated with continuous $n \times n$ multicasts. That is each member of a group of size $n$ continuously multicasts messages to all members of the group. This communication pattern incurs an intensive use of the underlying communications system and also models some interesting classes of applications, such as the branch-and-bound technique and $n$-way video conferencing.

Ethernet is a predominant network where group communications applications run today. Problems with performance of group communications systems over Ethernet have been repeatedly reported, yet no systematic study has been done.

Our findings are:

- Providing reliability is prohibitively expensive under high loss with or without Horus.
- The cost of providing full reliability is modest compared to that of providing simple reliability.
- The lack of receivers' resources contributes to high loss.
- The lack of synchronization among group members also contribute to high loss.
- Controlling the send rate based on receivers' resource availability is a promising way to maintain high throughput under high load.

Note that the focus of this paper is high-level communications. Thus, in this paper the term multicast refers to the act of a sender sending messages to all members of a group. In general, this may or may not correspond to an underlying transport multicast mechanism. In studies reported in this paper, a group multicast does correspond to a transport multicast, since we used IP multicast for all our group communications.

The rest of this paper is organized as follows. Section 2 describes layers of the fully reliable stack that we studied. Section 3 describes some examples of real applications with a $n \times n$ communication pattern and presents an application we developed for our studies. Section 4 describes the experimental hardware platform used for our experiments. Section 5 presents the performance of the same $n \times n$ communication application that embeds a reliable communication mechanism obtained without Horus. Section 6 presents the performance of a $n \times n$ communication application layered over a fully reliable Horus stack. Section 7 summarizes what we learned from our studies and lists areas for further research.

# 2 Layers for Fully Reliable Communications

This section describes essential layers of the fully reliable stack that we studied. We describe layers bottom up. For each layer, we describe the functional requirement of the layer, the semantics resulting from adding the layer to the stack created by layers below it, and a particular implementation that achieves the layer's functionality.

## 2.1 Basic Communication (COM)

The COM layer by itself provides a uniform interface for lossy communication over a variety of transport protocols. The current C implementation of Horus supports UDP/IP (including Deering IP extension) and ATM. An application chooses a specific transport protocol via an argument to the COM layer. The COM layer converts the Horus send and receive interface into actual transport calls.

## 2.2 Negative Acknowledgment (NAK)

The NAK layer provides detection of lost messages, requests for retransmission of lost messages, and retransmissions of requested messages. The stack resulting from placing the NAK layer over the COM layer provides reliable communication. The NAK layer also provides FIFO delivery of messages to the layer above.

The basic implementation of the NAK layer is as follows. Each sender stamps every message going to the same set of receivers with a monotonically increasing sequence number. Once in a while, each receiver checks the sequence numbers of messages it received. If a receiver detects a missing sequence number, it requests retransmission of the lost message to some subset of members that have the message. There are various possible choices for this subset. The simplest choice is the set containing only the sender, and this is what the Horus currently implements.

Another issue for the NAK layer is when to check for lost messages. One obvious choice is at every receipt of a message, but this does not work if traffic is intermittent. If a message is lost, the receiver will not discover the loss until the sender sends another message. If the sender is idle for a long time after sending a message that is lost, it would take a long time for the receiver to notice. In Horus this problem is solved by is to send periodically sending a "gossip" message that informs receivers of the highest sequence number on messages sent so far.

Deciding when to issue a retransmission request is another complex issue. Sending a retransmission request too early may result in spurious requests and subsequent spurious retransmissions, since a receiver may send requests for messages that are in transit and not lost. Deciding late may result in longer time to finally receive the message. This reasoning applies not only to the loss of original data messages, but also to the loss of retransmission requests and loss of retransmissions.

In Horus, a timeout is used for retransmission requests. When a receiver detects a missing message, it issues a retransmission request only if enough time has passed since the last request. This alleviates flooding that may result from lost retransmission requests and retransmissions. It still does not, however, solve flooding that may result from a premature decision of lost messages.

### 2.3 Membership (MBRSHIP)

The MBRSHIP layer provides virtual synchrony, that is, it guarantees that members of the same view receive the same set of messages between every pair of view changes[1]. The MBRSHIP layer over the reliable communication stack provides *fully* reliable communications—that is, receivers are ensured that they receive all the messages that other members in the same view received.

Full reliability requires knowledge of group membership. To see this, note that full reliability requires the sender to keep each message that it sent until it knows all receivers have received it. If a receiver asks for a message at some later time and the sender does not have it, then the sender has violated the full reliability requirement. An alternative of saving all messages indefinitely requires infinite storage, which is not implementable. In turn, to know that all members have received a particular message, the sender must know who the potential receivers are—or equivalently, the membership of the group.

During normal communications, the MBRSHIP layer carries out two functions: buffering of unstable messages[2] and detection of member failures. The detection of stable messages and detection of member failures require additional message traffic. The overhead is reduced by piggybacking some of this information into the NAK update traffic.

### 2.4 Rate Control (RATE)

To provide full reliability, only the layers described previously are required. For efficient use of system resources under high load, however, an additional mechanism—flow control—is necessary. Providing reliability is expensive. When a message is lost, it incurs at least two additional message sends (a request for retransmission and a retransmission) in addition to CPU processing at the end hosts. Flow control attempts to minimize message loss, thereby reducing overall cost.

The RATE layer provides flow control by regulating the speed at which senders send messages. This approach is based on the observation that messages often get dropped at end hosts when they run out of resources (e.g., system buffers). If a sender sends messages at a rate slower than the maximum speed receivers can handle, fewer messages are lost. The decision about what rate receivers can handle messages is left up to receivers. Prior to sending, a sender asks each receiver a permission to send at a certain rate. Each receiver, knowing its resources and prior commitments to other senders, decides whether it can handle the requested send rate. If it can, it gives the sender permission.

More specifically, each sender specifies the send rate as a pair of numbers: the number of messages to send and an interval over which these messages are sent. In the current implementation of Horus, the sender also specifies the duration over which this rate should be valid.

Internally, the RATE layer regulates the send speed at some internal interval. When a sender tries to send a message, the rate control layer checks to see if the sender has exceeded the limit within the current interval. If not, the message is sent. If it is, the message is buffered. When the

[1]The MBRSHIP layer is actually a complex layer consisting of many sub-functionalities, each with multiple implementation possibilities. Horus also offers multiple layers that together provide the MBRSHIP functionality.

[2]We say message is stable if it is received by all the receivers.

rate interval is up, the counter for the number of messages sent is reset and buffered messages are sent.

Our implementation of rate control is similar to the leaky bucket approach [Tur86]. Even if the sender is too slow in sending at some point and cannot send at the requested rate in one interval, it is not allowed to send at a rate that exceeds the requested rate in some future interval.

Note that our rate control layer only concerns end system resources[3]. Thus, to control the rate in multi-hop networks, we need help from a network-level rate controller (see, for example, [BCS94, CSZ92]).

# 3 An Application: $N \times N$ Multicasts

The communication structure of the application used in our studies is continuous $n \times n$ multicast. This contrasts with one-way broadcast patterns (or $1 \times n$) often studied in multimedia systems. Obtaining high performance with $n \times n$ multicast pattern in an asynchronous environment is challenging because of the extremely high load. If left uncontrolled, the system can easily get congested, causing many messages to get dropped and throughput to degrade rapidly.

Examples of real applications that involve continuous $n \times n$ multicasts among members include the branch-and-bound technique (see, for example, [LS85]), the alpha-beta search technique (see, for example, [MC82]) and n-way video conferencing.

Branch-and-bound is a technique useful for solving a large class of combinatorial optimization problems. Like divide-and-conquer, branch-and-bound divides a problem into multiple parts (branching), but unlike divide-and-conquer, processors working on subproblems gossip about their partial results as they make progress, so that other processors can avoid wasting time working on subproblems that will not contribute to the final solution.

The $n \times n$ communication application we designed is called *ring*. The ring application forms a group with a specified number of members ($n$), and performs the following: multicast a specified number of messages ($k$) of a specified size ($s$) to the group and concurrently wait for the same number of messages from each member.

The ring application written for experiments in Section 5 took parameters in addition to the basic parameters discussed above. The first is the degree of interleaving ($i$). It specifies how many messages are sent before each member attempts to receive. Lower values for this parameter correspond to tighter synchronization among the members of the group.

Note that it is normally the system software's job to decide how to interleave send and receive operations. Applications in general neither care to tune communications performance at this level nor are deterministic enough to do so effectively. We gave control to the application to learn just how well the underlying system can perform.

The second parameter is the maximum number of nonblocking receive trials per message to be received. Since messages do get lost, actually waiting to receive a specific number of messages before moving on to sending more messages could result in unbounded wait without any progress in unreliable communication. The receive trial value gives a way of giving up receiving a message before going on to the next stage. This number can be thought as timeout value for blocking receive. We fixed this number to be 100 (i.e., it tries to receive up to 100 times per message before giving up and go on to receiving next message), which resulted in very little loss without sacrificing throughput when other parameters are tuned appropriately.

[3] Although our main concern in the experiments reported in this paper were system buffers, we envision controlling all system resources in the future.

# 4 Hardware Platform

The platform we used in all our studies is SunOS4 running on four SparcStation 10s connected by a 10Mb Ethernet bus[4]. We used IP multicast for all multicast communication. Our underlying platform is comparable to, if not typical of, environments used for parallel computing today. The machines and the network were not dedicated to our experiments, but were generally lightly loaded.

In all our experiments, we run one group member per machine. We fixed the number of members to be four, since it is large enough to exhibit interesting aspects of multicast communications and yet small enough for us to carry out many experiments. We often repeated experiments with different numbers of members to check that our results were general to group communications and not particular to having four members.

We used a medium sized message (1000 bytes) for all experiments. We ran all experiments multiple times. This paper presents data from typical runs.

# 5 Baseline Performance

To enable a meaningful analysis of performance observed under Horus, we first report the performance of the ring application without Horus.

## 5.1 Software Structure

The software for the baseline experiments consists of three modules. Two are communications modules: one for unreliable communications, and one for reliable communications. The remaining module is the ring application.

The communication module implements send and non-blocking receive operations. Thus, to receive a message, an application may have to attempt to receive multiple times.

The reliable communication module attempts to simulate the Horus reliable communication stack (i.e., the NAK layer and the COM layer) without the overhead of layering. Each module has various parameters we do not discuss in this paper. In general, we tuned non-experimental parameters of each module until we obtained the best performance. The reliable communications module does not implement the semantics of the membership layer. The ring application and the reliable communications module together are about 2000 lines of C code.

## 5.2 Result

Figure 1 shows the baseline performance of *unreliable* communications. For this run we increased the degree of interleaving ($i$) from 10 to 1000 in increments of 10. Each member sent a total of 1000 messages (i.e., $k = 1000$). It shows that as the degree of interleaving is coarse, more messages get lost and throughput degrades. When the degree of interleaving is tight, this configuration almost fills up the underlying capacity of the network (1000 bytes/messages $\times$ 1000 messages/sec = 1MB/sec). However, throughput quickly dropped to one tenth of the underlying capacity as the loss rate exceeded 15%.

Figure 2 shows the baseline performance of *unreliable* communications in more detail. The pair on the left shows a run of the degree of interleaving from 15 to 50 in an increment of 1. The pair shows that there is roughly an inverse linear correlation between loss rate and throughput. The pair on the right shows a run of the degree of interleaving from 1 to 70 in an increment of 1. The

[4]In reality, we picked the four most lightly loaded machines among five on the Ethernet.

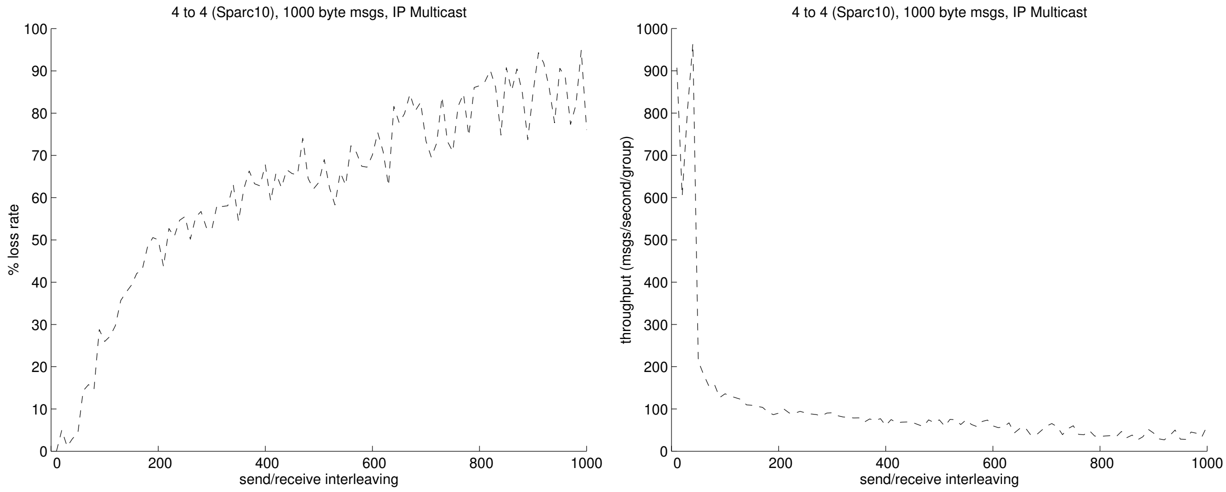


Figure 1: The baseline loss rate and throughput for *unreliable* communications

pair shows that the inverse linear correlation disappears as the degree of interleaving gets larger. Although loss rates increase linearly with the degree of interleaving from 40 to 70, throughput drops sharply around interleaving of 46, which has a loss rate of 5.65%. This suggests that the degree of interleaving, and not just the loss rate, strongly affects the throughput. We suspect that this has to do with the fact that when members are not synchronized, some time is wasted while a member waits idly for another member to complete some task. We have preliminary evidence on this theory, but need further study to verify it.

The loss rates shown in Figures 1 and 2 are actually the average of loss rates experienced by individual members. An inspection of per-member data used to generate Figure 1 reveals that in many cases the loss rate varies widely among members of the group. This suggests that not all members lose the same set of messages and that it is often a few slow members that lose most of the messages. This observation together with the fact that our Ethernet is a bus suggests that at least some, if not most, losses occur due to a lack of receivers' system resources and not due to the sender's system resources nor due to the network. One system resource that receivers run out is UNIX receive buffers. All members had UNIX receive buffer size of 32 messages.

Figure 3 shows the baseline performance of reliable communications. This shows that the NAK algorithm does not help in providing better throughput when the loss on the first send gets too high[5]. When the loss of the first send is below 3%, we get reasonable throughput and completely recover all losses within a reasonable time limit[6]. At a loss rate of 7% we get throughput one half of the underlying capacity and have trouble recovering some losses. At a loss rate above 20% we get throughput one third to one fifth of the underlying capacity and cannot recover a significant portion of losses.

An interesting observation is that with the NAK mechanism, the loss on the first send is hardly ever at a level between 7% and 15%. The loss rate is either well below 7% or jumps above 15%. This comes from a combination of two factors. First, the loss on the first send shown in Figure 3 is not a true loss on the first send, but rather what is perceived to be loss on the first send. The receiver perceives that a message as lost if it receives a message out of order. With the periodic update

[5] Changing the retransmission request timeout values had no effect. The graph shown used timeout value of 0.

[6] With MBRSHIP semantics in place, unrecoverable loss would result failure of a member.

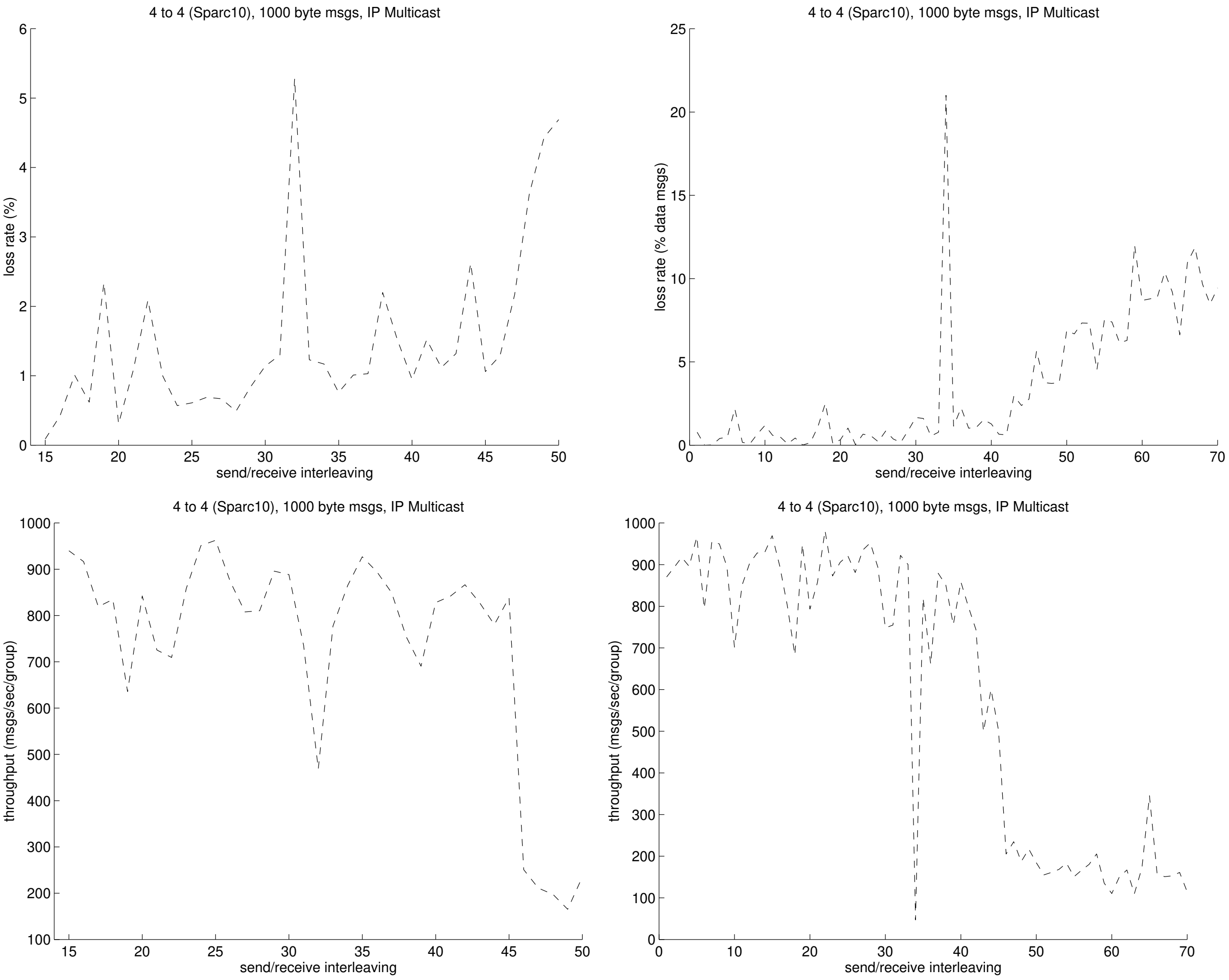


Figure 2: The baseline loss rate and throughput for *unreliable* communications in detail

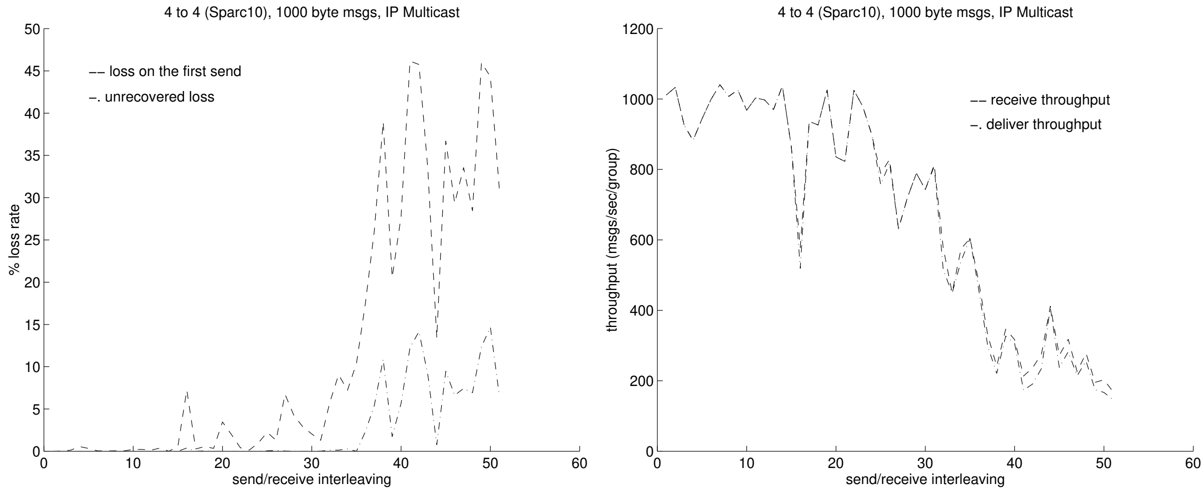


Figure 3: The baseline loss rate (before nak and after nak) and throughput (receive and deliver) for reliable communications

mechanism, it is possible for a message to be lost yet be recovered before the next message arrives at the receiver. (This theory is based on actual data, which is not shown in this paper.) This would drive the perceived loss lower than the actual loss on the first send. This benefit disappears as the actual loss on the first send gets too high, thus the jump in the perceived loss rate. Second, as more messages get lost, the NAK mechanism introduces more load into the network driving the loss rate even higher. This explains the general decline in throughput from the degree of interleaving 25 to 35, despite the same level of (perceived) loss on the first send. The actual loss must get higher as the degree of interleaving becomes coarse. The general decline in throughput might also be explained by idle time introduced by the coarser degree of interleaving.

We can conclude from experiments in this section that the cost of providing reliable communication under a high loss rate is prohibitively expensive, at least with the NAK algorithm we implemented. Of course, since the cost of supporting reliable communications under high loss is prohibitively expensive in the point-to-point case [Jac88], it is hardly surprising that this is the case for multicasts as well. A more interesting conclusion we can make from these experiments is that the availability of receivers' resources is a key to obtaining good throughput. Synchronizing members of a group is one way to control use of receivers' resources, although this option is not available to all applications. Synchronizing members of a group has another potential benefit that it helps reduce idle time created by a member waiting for another member in completing some task.

# 6 Performance of Horus

This section reports the performance of the ring application over the Horus group communications system over the platform described in Section 4. Performance of Horus depends on the layer stack used by applications. We present performance with two different stacks: the MBRSHIP stack, which provides fully reliable communications, and the RATE stack (i.e., the RATE layer over the MBRSHIP stack). For experiments in this section, measurements were taken after the membership was established. Thus, data includes the overhead of the MBRSHIP layer during normal communications but not the overhead of membership establishment.

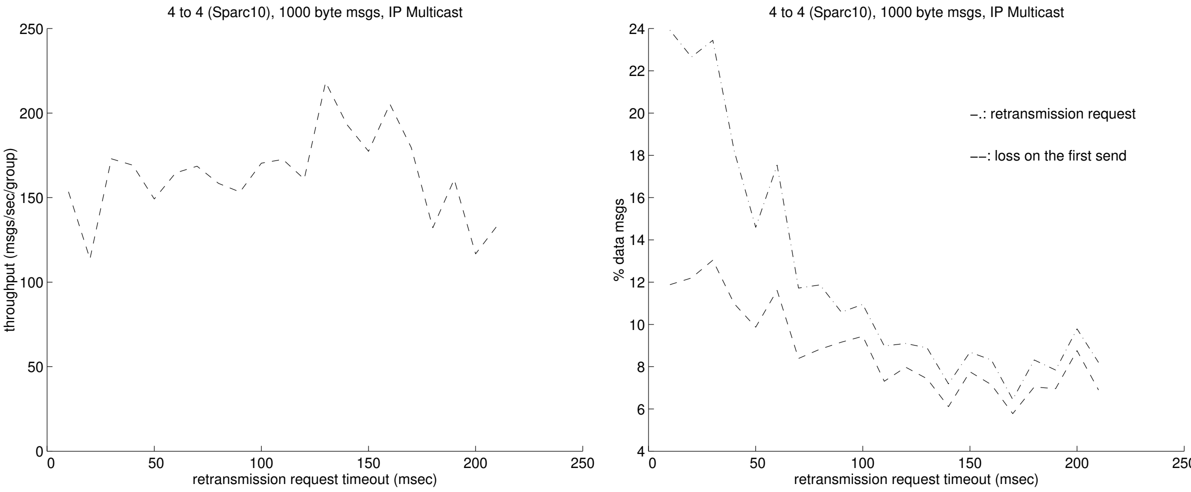


Figure 4: The throughput, the loss on the first send, and the retransmission requests (latter two as a percentage of data messages) observed with the Horus MBRSHIP stack.

The ring application for experiments in this section had no control over the degree of interleaving. It sends all messages at once and lets Horus decide how to interleave send and receive operations.

## 6.1 Performance without Rate Control

Figure 4 shows the deliver throughput obtained, the loss (i.e., the number of data messages that required retransmission(s) as a percentage of data messages received), and the percentage retransmission request (i.e., the number of retransmission requested as a percentage of data messages received), observed with the MBRSHIP stack with various retransmission request timeout values. Longer retransmission request timeouts result in fewer retransmission requests. Notice how the loss rate and the request rate converge as the retransmission request timeout gets longer. This suggests that the reduction in the number of retransmission requests is partially due to a reduction in redundant requests. Interestingly, longer retransmission request timeouts also result in a lower absolute level of loss rate. This is probably because the reduction in retransmissions caused by longer retransmission request timeouts imposes a smaller load on the network, which in turn causes fewer messages to be lost. This is not a one way relationship. Lower loss in turn causes fewer retransmission requests.

Unfortunately, this reduction in loss rate did not translate into better throughput in this experiment. This is because the minimum observed loss rate of 6 to 8% is still rather high. This level of loss corresponds to the degree of interleaving of about 35 in the baseline case for reliable communications presented in Section 5. We observed that throughput declined sharply to 200 messages per second around this degree of interleaving. Thus the performance of Horus is comparable to the baseline case. Once membership is established, the MBRSHIP layer introduces little additional overhead. The MBRSHIP stack does perform poorly, but not much more poorly than the base case given the level of loss. As in the baseline case, we observed (data not shown in this paper) that variance in loss rate is high among members of the group. This once again indicates that availability of receivers' resources is a key to obtaining good throughput.

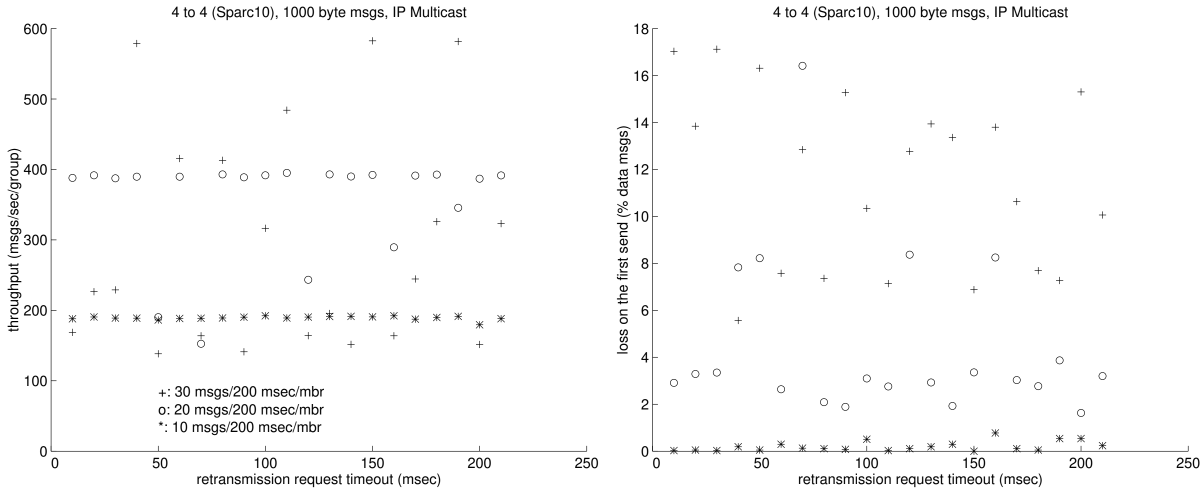


Figure 5: The throughput and the loss on the first send observed with rate control over the Horus MBRSHIP stack. Timeout on retransmission requests seem to have no correlation to either the throughput nor the loss rate. This suggests that avoiding loss (by controlling rate) is better than managing loss.

The performance obtained with the MBRSHIP stack was disappointing. The throughput obtained was at most one fifth of available bandwidth. This is somewhat expected. The ring application for these experiments relied completely on Horus for providing synchronization. Yet the MBRSHIP stack has no synchronization. It has to rely completely on the Horus thread scheduler. We observed in Section 5 that lack of synchronization leads to poor receivers' resource utilization, which leads to high loss rates, which in turn results in low throughput.

## 6.2 Performance with Rate Control

As described earlier, the RATE layer manages receivers' resources by controlling the send rate based on receivers' resource availability. Figure 5 shows the deliver throughput and the loss rate observed with the rate control layer over the MBRSHIP stack. In our experiments, we set the send rate to be the same for all members. Three types of curves in Figure 5 corresponds to three different rates. A per member rate of 10 messages per 200 milliseconds, which is equivalent to 50 messages per second, corresponds to an aggregate group throughput of 200 messages per second for a four member group. A per member rate of 20 messages per 200 milliseconds corresponds to a group rate of 400 messages per second, and per member rate of 30 messages per 200 milliseconds corresponds to a group rate of 600 messages per second.

Figure 5 shows that the RATE layer does improve the throughput of the MBRSHIP stack. It can fairly reliably maintain at least twice the level of the best throughput attainable without the RATE layer[7]. The current implementation of the RATE layer still has trouble maintaining the level of 600 messages per second, which is only 60% of the underlying capacity. The occasions when it succeeds in providing the 600 messages per second throughput correspond to loss rates below 8%.

[7] We believe that occasional failure to provide this level is due to the fact that our experiments were run on a non-dedicated environment.

Figure 5 also shows that there is no correlation between the retransmission request timeout and maintainable throughput. This suggests that controlling the send rate is a far superior mechanism for combating losses than controlling the retransmission request timeout. In other words, it is better to avoid losses than to manage losses. This is similar to the conclusion drawn for why TCP Vegas performs better then TCP Reno [DLY95].

## 7 Summary and Future Work

Our studies showed that providing reliability is prohibitively expensive under high loss with or without Horus, and the cost of providing full reliability in addition to simple reliability is minimal once the membership is established. Our studies also showed that a lack of receivers' resource and a lack of synchronization among group members contribute significantly to high loss.

Our studies demonstrated that controlling the send rate based on receivers' resource availability is a promising way to maintain high throughput under high load. The current implementation of the rate control mechanism could achieve up to 40% of underlying bandwidth, or twice the level achievable without it. However, the current implementation still had trouble maintaining 60% capacity, despite the fact that we can achieve 100% capacity by tightly synchronizing members of a group.

Our studies raise a question: can we control the loss rate further in an asynchronous group communications system? We are now investigating the following areas:

- Better rate control. The current implementation of RATE layer maintains an average rate over an interval rather than an exact rate. We might obtain lower losses by using tighter rate control.

- Better scheduling mechanism. In particular, we are moving away from the thread model used in the current implementation of Horus, which is hard to control, and investigating more flexible scheduling of events across layers.

- Use of hints from applications for better synchronization. Although this is not an option for some applications, it appears to be a promising approach for more deterministic applications.

## Acknowledgments

We are grateful to Mark Hayden and Jason Hickey for their comments on this work, and particularly to Scott Stoller, for his thorough copy-editing, which greatly improved the paper.

## References

[BCS94] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview. Request For Comments (RFC) 1633, June 1994.

[Bir93] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[BS95] Nina T. Bhatti and Richard D. Schlichting. A system for constructing configurable high-level protocols. In *Proceedings of ACM SIGCOMM '95*, pages 138–150, Cambridge, Massachusetts, August 1995. ACM SIGCOMM.

[CSZ92] David D. Clark, Scott Shenker, and Lixia Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *SIGCOMM '92 Conference Proceedings*, pages 14–26, Baltimore, Maryland, August 1992. ACM SIGCOMM.

[CT90] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generations of protocols. In *SIGCOMM '90 Symposium*, pages 200–208, Philadelphia, Pennsylvania, 1990. ACM SIGCOMM.

[Dee89] S. Deering. Host extension for IP multicast. Request For Comments (RFC) 1112, August 1989.

[DLY95] Jon Suk Ahn Peter B. Danzig, Zhen Liu, and Limin Yan. Evaluation of TCP Vegas: Emulation and experiment. In *Proceedings of ACM SIGCOMM '95*, pages 185–195, Cambridge, Massachusetts, August 1995. ACM SIGCOMM.

[FJM+95] 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 *Proceedings of ACM SIGCOMM '95*, pages 342–356, Cambridge, Massachusetts, August 1995. ACM SIGCOMM.

[HSC95] Hugh W. Holbrook, Sandeep K. Singhal, and David R. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. In *Proceedings of ACM SIGCOMM '95*, pages 328–341, Cambridge, Massachusetts, August 1995. ACM SIGCOMM.

[Jac88] Van Jacobson. Congestion avoidance and control. In *SIGCOMM '88 Symposium*, pages 314–329, Stanford, California, August 1988. ACM SIGCOMM.

[LS85] N. Lai and A. Sprague. Performance of parallel branch-and-bound algorithms. *IEEE Transactions on Computers*, C-34(10):962–964, October 1985.

[MC82] T. A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys*, 14(4):533–551, December 1982.

[Tur86] Jonathan S. Turner. New directions in communications (or which way to the information age?). *IEEE Communications Magazine*, 24(10):8–15, October 1986.

[vRBF+95] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in Horus. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 80–89, Ottawa, Ontario, August 1995. ACM SIGACT-SIGOPS.