Performance of the ISIS Distributed Computing Toolkit*

Kenneth P. Birman
Timothy Clark

TR 94-1432
June 1994

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

* This work reported was supported by ARPA/ONR grant N00014-92-J-1866 and by ISIS Distributed Systems, Inc. As described in the text of the paper, the performance figures reported here are for a specific suite of test programs and a particular hardware and software environment, and may not hold for other combinations of hardware and software, or for versions of ISIS other than the one evaluated. Both authors are employees of ISIS Distributed Systems, Inc. Ken Birman is also Professor of Computer Science at Cornell University. Email addresses for authors: ken@cs.cornell.edu, tim@isis.com. Please cite as Technical Report TR-94-1432, Dept. of Computer Science, Cornell University.

Performance of the Isis Distributed Computing Toolkit

Kenneth P. Birman and Timothy Clark*

June 22, 1994

Abstract

The Isis Toolkit is a programming environment for building process-group structured distributed software. The system is widely used in settings requiring high reliability, strong distributed consistency guarantees, and high speed communication. In this paper, we describe experimental studies of Isis performance. Our work explores the impact of hardware support for multicast performance, with a focus on flow control mechanisms. The use of hardware multicast in Isis has not been discussed elsewhere. One conclusion of the paper is that although Isis performance is limited primarily by flow control considerations, this type of hardware support can lead to significant performance improvements for certain communication patterns. A second conclusion was that the Isis flow-control problem is surprisingly difficult. More work in this area, and on the underlying operating system communications layer (UDP), could have significant impact on the system.

Keywords and phrases: Distributed computing, performance, process groups, atomic broadcast, causal and total message ordering, cbcast, abcast, multiple process groups, hardware multicast, IP multicast, virtual synchrony, fault-tolerance.

1 Introduction

This paper discusses the performance of Isis, a Toolkit for building distributed applications. Isis was developed as a UNIX-oriented distributed programming environment, although the system has now migrated to a number of other platforms. Isis has been used in many commercial applications, including military and commercial air traffic systems, telecommunications switch control for the ESS-7 architecture, network and application management and control,

*The work reported was supported by ARPA/ONR grant N00014-92-J-1866 and by Isis Distributed Systems, Inc. As described in the text of the paper, the performance figures reported here are for a specific suite of test programs and a particular hardware and software environment, and may not hold for other combinations of hardware and software, or for versions of Isis other than the one evaluated. Both authors are employees of Isis Distributed Systems Inc. Ken Birman is also Professor of Computer Science at Cornell University. Email addresses for authors: ken@cs.cornell.edu, tim@isis.com. Please cite as Technical Report TR-94-1432, Dept. of Computer Science, Cornell University.

financial trading, wide-area database management, weather and environmental monitoring systems, factory floor process control, scientific computing and other applications. Many of these classes of applications have demanding performance requirements, and with effort we have been successful in meeting most such demands. Yet despite having a user community for which performance is a major concern, Isis performance has not previously been studied in any detail. This paper undertakes such a study, working with Isis V3.1.0 (Jan. 1994) on SUN Sparc 10/51 computers running Sun OS 4.1.3c with IP multicast software on a 10Mbit ethernet.

There has been a considerable amount of work on the performance of reliable multicast protocols [2, 1, 5, 6, 7, 8, 9]. However, the prior work of which we are aware has focused on the multicast performance of a single group of processes at a time, confining attention to issues such as throughput (messages per second), latency, and delay to stability. These single-group issues are also explored here, but because we worked with the Isis system itself, as opposed to a simulation or a mock-up of a single protocol, our work also grapples with the realities of operating over the UNIX operating system, the overhead of the Isis execution model (a multi-threaded one, supporting a sophisticated message representation), and synchronization mechanisms for using multiple process groups in a single application.

Isis supports an execution model called virtual synchrony, and this places protocol obligations on the system that have not generally been considered in previous studies. Moreover, Isis needs to give good performance for a wide range of communication patterns, including RPC, multi-RPC, streaming on a point-to-point and group basis, one-shot multicasts issued in rapid succession within different groups, and also must perform well for varied sizes of messages. These properties of the system have a major impact on protocol performance tradeoffs.

Additionally, we report for the first time on the impact of using hardware multicast mechanisms in Isis. With the growing commercial acceptance of IP multicast as a vendor-supported internet transport protocol, exploitation of multicast no longer requires the development of special operating systems or network device drivers. Here, we demonstrate that the impact of such communication features can be as dramatic in a system such as Isis as in a system with less complex protocols and architecture. This supports a long-term thesis of our work, namely that although Isis is internally complex, this complexity need not result in lower performance than for much simpler systems running on the same sort of UNIX layering. The flow control mechanism used in our IP multicast layer is reported in considerable detail here, with experimental results from some of the alternatives we explored but decided not to retain within the system.

2 Isis System Architecture

Although we do not wish to repeat material that has appeared elsewhere, discussion of the performance issues seen in Isis requires some familiarity with the programming model and approach to reliability embodied in our work. Accordingly, this section reviews key ideas in Isis, and summarizes architectural considerations relevant to our performance studies. Readers already aware of our work may wish to skip directly to Section 3.

Isis is a toolkit for building distributed applications. The key premise of the approach is that by offering application developers a powerful collection of primitives for creating and programming with groups of cooperating processes, important new classes of distributed applications could be developed.

From a software perspective, the important features of the system are these:

- It introduces a layer of software support that extends UNIX (or NT, Mach, Chorus, OS/9, etc) with tools for programming with distributed process groups and for performing reliable, ordered, group communication.

- It provides a layer of programming tools for group-oriented applications, supporting replicated data, distributed synchronization, monitoring and failure-triggered actions, parallel computation, and so forth. This layer is available from languages such as C, C++, Ada, Common Lisp, SmallTalk, Fortran, etc.

- The programming model is multi-threaded. Each incoming message is handled by a newly created thread, an overhead that we accept because of the simplification it brings in user-level code. This overhead is reflected throughout the performance figures seen below.

- Isis messages are sophisticated objects. Each message is a container for a set of data fields, represented by tuples: *(field-name,instance-number,type,length,data)*. Messages can contain embedded messages and references to out-of-line data, and can be accessed through a message-I/O library patterned upon the UNIX printf/scanf library. Data representation transformations are automated by the system. As with the task model, the overhead of this scheme is tolerated because of the simplification it yields.

- Isis extends the basic local-area network tools into an "enterprise" communication and computing environment, including higher level applications such as reliable distributed file servers, publish/subscribe message bus tools, wide-area spooling and communication facilities, reliable distributed application management and control software, etc. The typical large Isis application makes heavy use of these pre-existing subsystems, customizing them by adding new reliable services that operate over our tools layer.

In addition to these basic features, Isis implements an execution model that allows the user to reason rigorously about the possible behaviors of the system, and to develop algorithms with which group members can be proved to behave consistently with respect to one another. By consistency we refer to a collection of properties that allow group members to cooperate, for example to simulate a single, highly reliable process, to replicate data and support state transfer to a joining member, to dynamically reconfigure in the face of changing load, failures or recoveries, to synchronize actions, subdivide tasks, etc. The model we developed for this purpose is called *virtual synchrony*, and can be thought of as an adaptation of *transactional serializability* to a setting characterized by groups, cooperative computing, and communication interactions (in contrast, the transactional world is dominated by databases stored on persistent storage and accessed concurrently by independent application programs). The properties of this model are summarized in Figure 1.

The virtual synchrony model can be formalized and related to the theory of asynchronous distributed systems. Doing so has permitted us to develop rigorous correctness and optimality proofs for our protocols, and greatly simplifies the development of complex distributed algorithms that operate over the basic Isis tools [3].

As is the case with other aspects of Isis, virtual synchrony has some associated cost. But it also *reduces* costs within application programs, by providing a strong model that simplifies the application process. For example, an RPC performed in the Isis framework reports failures with strong semantics: failure means that the service to which the RPC was issued has genuinely crashed. A conventional RPC environment, such as OSF's DCE environment or SUN's ONC environment will only report timeouts, with no guarantees that these correspond to failures.[1] The RPC programmer who uses Isis may thus be able to launch a failure handling mechanism after a single failed RPC, where the DCE or ONC programmer would potentially loop retrying the same request, or otherwise engage in a more complex algorithm that attempts to cope with failures.

Thus, when comparing the cost of a mechanism within Isis to the corresponding mechanism in some standard technology, one should bear in mind that many applications will be simpler - and do less communication - when run over Isis than when constructed using alternatives.

Virtual synchrony or closely related models have used by several other research projects, such as Transis, Ameoba, Gossip, Psync and Totem. In our experience, these approaches open new classes of distributed applications, which would be extremely difficult to build in more conventional stateless distributed environments, such as the standard RPC and stream facilities provide.

Table 1 summarizes the performance of some of these basic mechanisms.

Basic Costs			
Isis Tasks over Solaris 2.3 threads		**Messages**	
Task create	164 usecs.	Create (empty)	6 usecs.
Context switch	78 usecs.	Put integer	7 usecs
Null tasks/sec	3067	Extract integer	8 usecs.
		Overhead (2k msg)	144 bytes
Native Solaris 2.3 threads		Put 2k characters (copy)	48 usecs.
Task create	155 usecs.	Extract 2k (copy)	48 usecs.
Context switch	75 usecs.	Put by-reference	10 usecs.
Null tasks/sec	3268	Reconstruct from external format	12 usecs.

Table 1: Isis V3.1.0 Primitives

[1] One can, of course, integrate Isis with RPC systems like DCE and ONC, in which case the DCE or ONC RPC would inherit the properties of the Isis RPC.

Virtually synchronous groups model:

- Each process group has an associated *group view* in which members are ranked by the order in which they joined the group.
- Group view changes are reported in the same order to all members.
- Any multicast to the full group is delivered between the same pair of group views, and to all members of the group. Group membership is determined from the process group view last delivered prior to the delivery of the message.
- There is a way to transfer the state of a group member to a joining process at the instant when the group view reporting the join is reported.

Communication primitives:

- Multicasts are atomic (all destinations that remain operational will receive a message, if any destination receives it and remains operational sufficiently long).
- Multicasts are ordered. Isis supports a *causally* ordered multicast, *cbcast,* and a totally ordered multicast primitive, *abcast.* Causal order is a type of FIFO delivery ordering and is used for asynchronous streaming of messages. Total order extends causal order by ensuring that messages will be delivered in the same order to all group members, even when messages are sent concurrently.[a]

Failures:

- Failures are reported according to the fail-stop model; if a failure is reported to any process, all processes will see the same event.
- A failed process may be one that crashed, or one that was partitioned away. In the later case, when the partition is repaired, the process can only rejoin the system under a different process-ID, and after running a special disconnect protocol.
- Isis itself remains available only within a *primary partition* of the network. This means that any application that remains connected to Isis within a LAN can take actions on behalf of the entire LAN. Partitioning is handled through a cellular architecture that we discuss elsewhere.

[a]The causal ordering is used primarily as an optimization. Many Isis applications are initially designed to assume total ordering, but turn out to send asynchronous streams of multicasts only when holding some form of lock or mutual exclusion on a process group. In applications with this common communication pattern it is safe to substitute cbcast for abcast. The cbcast protocol will give the same ordering as abcast in this situation, but with significantly lower latency and much higher levels of "pipelining," which Isis exploits to pack many messages into each packet transmitted.

Figure 1: A summary of the virtual synchrony model

3 Communication Transport Protocols in Isis

Isis implements several protocol layers, using an architecture that is stacked somewhat like the ones seen in TCP/IP or the OSI architecture. The highest layer of protocols, labeled *vsync* in Figure 2, is concerned with supporting our full virtual synchrony model for multiple groups. Its overheads are as follows:

- It adds ordering information to messages, using what we call *compressed vector timestamps.* Timestamp size grows in proportion to the number of processes permitted to send in a process group, which is generally between 1 and 3 in Isis applications, but can rise to 32 in certain types of parallel codes. Even when multiple groups are used, the version of Isis we studied will not put more than one timestamp on each message.
- It may delay messages for atomicity and ordering reasons. There are a number of possible delay conditions. For example, transmission of a message may have to be delayed until some other message has been sent, a message may arrive before some other message that should precede it, the delivery of a totally ordered multicast may need to be delayed until the delivery ordering is known, and so forth.
- When a process group membership change is occurring, this layer ensures that each message is delivered atomically, before or after the membership change, at all members.

Below the virtual synchrony layer is a multicast transport layer. This layer has responsibility for delivering messages in the order they were sent (point-to-point ordering only), without loss or duplication unless the sender fails. Callbacks to the virtual synchrony layer report on successful delivery of a message to its remote destinations. The multicast transport layer exploits several message transport protocols:

- The UDP transport layer supports point-to-point communication channels using the UDP protocol (user datagram protocol).[2] This layer implements reliability using a windowed acknowledgement scheme, much as TCP does over the IP protocol. Our flow control algorithm attempts to balance I/O among all the channels that exist between a process and others with which it communicates, while also packing small messages together into larger ones: The cost of communication in UNIX includes a very large fixed overhead per-packet, and a much smaller factor that is size-related - up to the UDP limit, which varies from vendor to vendor, but is frequently limited to 8kb per packet.[3]

[2]TCP is not used in Isis, for two reasons. First, TCP does not balance I/O when one source has channels to multiple destinations: performance is sluggish and erratic when the number of communication channels grows large. Secondly, TCP does not report successful delivery to the user – when a TCP write completes, the message will often still be in the TCP sliding window. Although TCP sends its own acknowledgements when a message arrives at its destination, there is no provision for informing the sender that this event has occured. Isis needs this information, and the cost of implementing an additional acknowledgement layer over TCP was judged to be prohibitive.

[3]Isis is designed to interoperate between systems from multiple vendors and hence is limited to the least common denominator in a situation such as this. However, since UDP runs over IP protocol, which employs

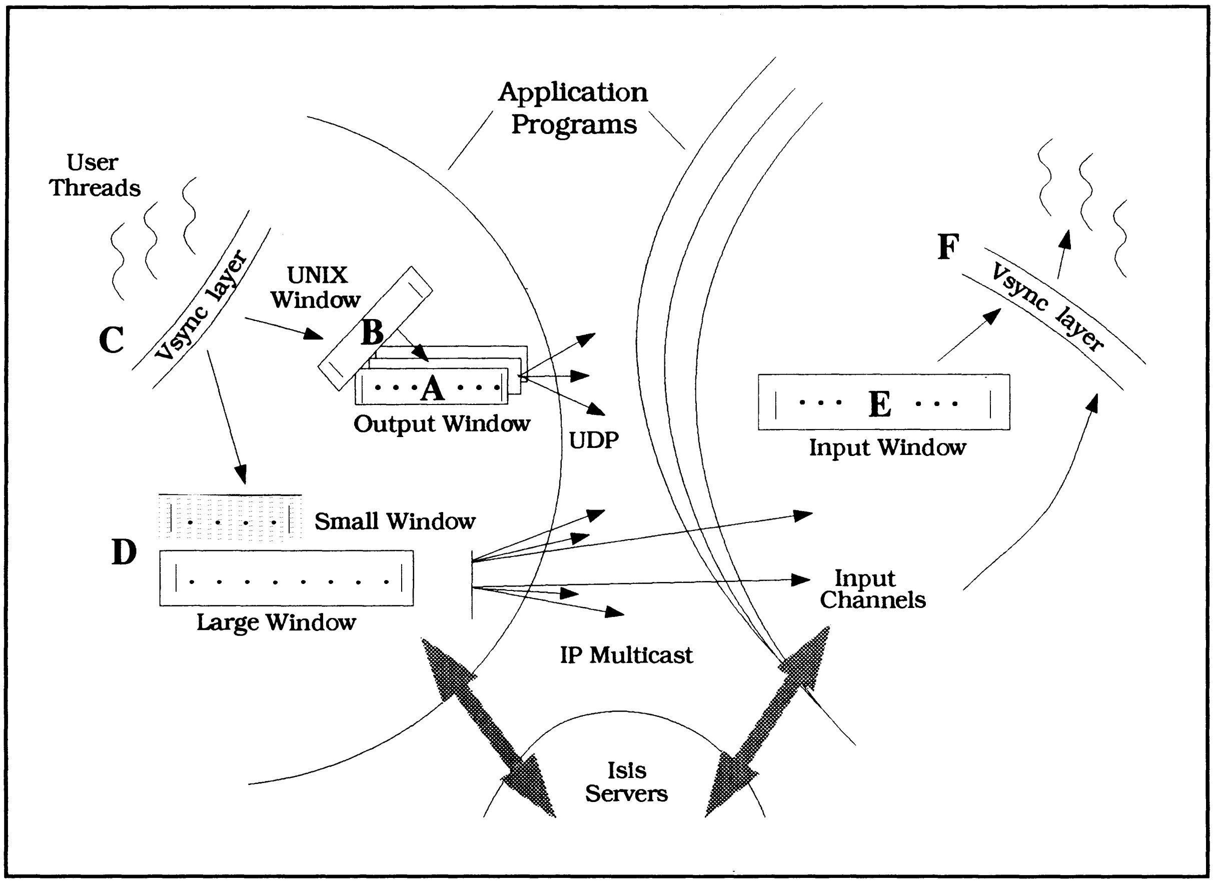

Figure 2: Isis Communication Architecture

Ref Fig. 2	Architectural Component	Flow-Control Strategy	Comments
(A)	Point-to-point Transport	Basic window	Positive/Negative Acks Rate and byte hi/low thresholds
(B)	Sender-to-UNIX	Aggregate window	Abstract window Timer-aged rate and byte thresholds
(C)	Initiate new multicast	Block when overloaded	Delays asynchronous senders Allows RPC *replies*
(D)	IP-multicast	Dual windows	Small window for basic flow control Large window for atomicity
(E)	Input window	Don't drain when overloaded	Creates backpressure in pipelined configurations
(F)	Vsync layer	Delay incoming messages	Reorder multicasts Implement vsync group addressing

Table 2: Flow control mechanisms used by Isis

Packing has an important implication that will be relevant later in the paper. If process p has channels to processes q and r, notice that the actual UDP packets travelling over these respective channels may be very different. Isis supports various point-to-point and RPC communication mechanisms, and q and r may not be members of the same set of process groups. Since the packing algorithm will be applied to the the *aggregate* traffic from p to q and r, the packet stream used in each case will differ.

- The IP-multicast transport layer. IP-multicast is similar to UDP, in that it provides an unreliable datagram mechanism. However, whereas UDP operates on a point-to-point basis, IP-multicast supports group destinations and the associated routing facilities. IP-multicast is not reliable, hence our protocol implements its own flow-control and error correction logic. Notice that although IP-multicast offers a reduction in the number of packets needed to send a given message to a set of a destinations, this assumes that identical packets must be sent to the destination processes. Recalling the discussion of UDP transport, above, the reader will see that the benefits of IP multicast relative to UDP communication depends upon the actual pattern of communication and the degree to which this assumption is valid.

- A transport layer using shared memory for local communication. This layer is planned for the future: Isis does not currently optimize for local communication, although the growing availability of shared memory in modern operating systems makes this feasible. An extension of Isis to communicate between processes on the same machine using a shared memory pool would greatly enhance local performance, as well as remote performance for groups having some local destinations.

- A transport layer for Mach IPC. We did not instrument the performance of this layer, which is experimental.

The above layering is used when application programs are able to communicate *directly*. Most Isis communication is direct (what we call "bypass" communication), and although there are other communication paths available within Isis, we will not discuss them here.

In addition to this question of communication performance, Isis has a number of replicated utilities and servers to which system calls are directed. These servers provide a communication name space, handle some aspects of failure detection and reporting, provide logging and spooling functions, do load-balancing and fault-tolerant file management, etc. The cost of these system calls depends upon two factors: the manner in which the application is connected to the server or utility program, and the number of processes involved in responding to the request. Representative costs are illustrated in Figure 3.

3.1 Message loss in UNIX and in lower-level transports

Message loss is a major issue in developing a reliable distributed system, and is the dominant problem with which Isis flow control must be concerned. The normal producer-consumer flow

1440 byte packets, there is little benefit to sending UDP packets that are far larger than 8kb. A smaller number of system calls are needed to transmit larger packets, but the relative performance impact of lost packets rises, and since the demand for kernel buffering space is increased, packet loss is more probable.

Isis System Calls	
Establish connection to Isis servers	
Connect via UNIX pipe (local)	39 ms.
Connect via TCP pipe (remote)	154 ms.
Lookup group address (local TCP)	
Group known to local server	20 ms.
Group known to remote server	22 ms.
Group unknown	24 ms.
Create group (local TCP)	
One "protos" server	20 ms.
Four "protos" servers	79 ms.

Table 3: Isis V3.1.0 System Calls

control issues are much easier to resolve, for reasons that will be more clear shortly.

Systems such as UNIX can drop messages both on the sending and receiving side, and in neither case is it common to report failures back to the application. This is illustrated dramatically in Figure 3, which graphs loss rates for the UDP protocol internal to a single machine and between two remote workstations. This test was run for a variety of messages sizes at the maximum transmission rate (no delay between sends), as measured by the Unix *spray* utility. Although the loss rate was high in this experiment, *not a single error was reported to the sending program by UNIX.* This speaks to a broader issue: since modern operating systems do not report failures - even when they know that a failure has occurred, systems like Isis must labor to guess that a loss has occurred and to overcome it - work that is in a strict sense not necessary, since perfect information about loses on the sending side should be available from the operating system, and a great deal of information about loss may be available even when this occurs on the receiving side! (A related issue involves detection of program termination and machine crashes.)

Reasons for packet loss include shortages of kernel buffering space in the fragmentation algorithm by which a UDP packet is carved into multiple IP packets, queue length restrictions on the network, buffer size limitations, etc.

Clearly, when working over UDP, one must keep in mind that this protocol is unreliable!

Message loss can also occur on the communication medium, typically in the receiver interface. This data is also expressed in Figure 3 by repeating the experiment in the remote case. Loss of data on the wire is uncommon, but overruns in the receiver interface can occur if a sequence of back-to-back packets are delivered without adequate time for the interface driver to resupply the interface with memory into which incoming messages can be copied. Loss can also occur at higher levels of the operating system, including that concerned with reassembly of UDP packets from the IP fragments that are actually transmitted and lack of memory in the delivery buffer of the eventual recipient. As the graph demonstrates, in the absence of flow control, packet loss can reach 100 percent for large packets (greater than

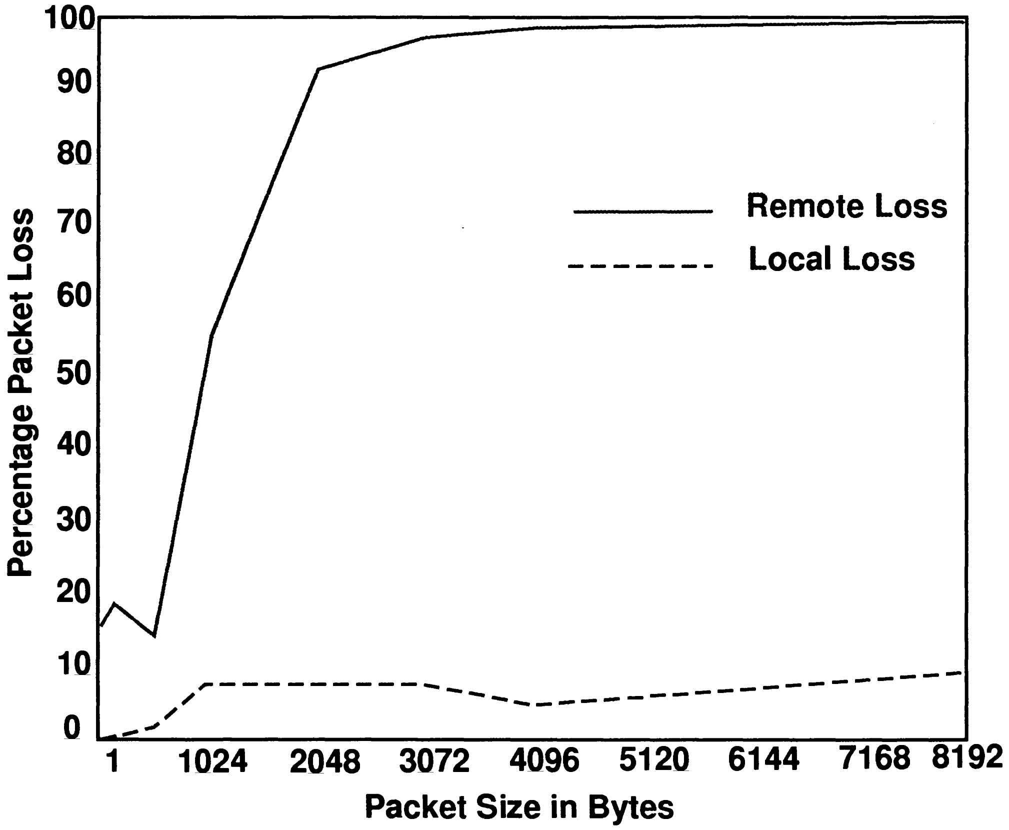

Figure 3: UDP Packet Loss Rates Without Flow Control

4K) in the remote case. Extreme loss rates arise primarily from lack of memory on the destination machine.

When using hardware broadcast (in which every message is delivered to every machine on the network segment), the frequency of back-to-back receptions can rise substantially. Protocol designers who use broadcast on hardware and machines shared with other general purpose applications must therefore avoid situations in which high volumes of broadcasts could be transmitted. The term "broadcast storm" is commonly used to describe a type of network thrashing, in which very high rates of message loss (of all types) is triggered by a heavy volume of broadcasts. In particular, if broadcast is used as a transport, even machines like file servers - which would not normally run Isis applications - might be exposed to high loss rates for normal non-Isis traffic. For this reason, Isis does not make use of hardware broadcast.

A related technology, called IP-multicast, has become standard and popular precisely because it offers greater selectivity than broadcast. IP multicast will transmit a UDP packet to a set of machines, which must be set up in advance of the multicast; any given machine can participate in a limited number of IP multicast sessions, which is limited by the hardware. The maximum number of sessions in our hardware configuration was about 60.

Like UDP itself, IP multicast is not reliable, and packet loss can occur both in the sender and upon reception. IP multicast is less likely to trigger storms than broadcast, because broadcast storms are often a reflection of packet loss occurring network-wide, and any excessive loss rates associated with IP multicast are confined to the recipient machines.

Nonetheless, our experiments with IP multicast showed that elevated loss rates can be observed if a single machine is the destination of a high rate of multicasts from many senders. Interestingly, this problem does not occur if a single sender originates the messages. We surmise that the problem is associated with delivery of back-to-back packets off the wire. With a single sender, there are normally small gaps between packets. Multiple senders, however, can potentially generate an almost continuous stream of data into a single destination, giving rise to a "multicast storm."

Summarizing, one sees that it will be extremely difficult to isolate the source of a message loss problem, and particularly difficult to do so in software that must run in heterogeneous environments. The only successful strategy for flow control is one that avoids transmission rates at which substantial message loss can occur. Experimentally, Isis performance is seen to degrade severely if packet loss exceeds about 1% of all packets transmitted. In the flow control methods discussed below, Isis will be seen to use several approaches, each aimed at a different loss scenario. These include rate-based flow control on the sending side, back-pressure mechanisms by which a receiver can choke back a sender, positive acknowledgement mechanisms, and negative acknowledgement mechanisms. This mixture of flow control approaches emerged over time, with each mechanism responding to a category of applications that performed poorly in the absence of the mechanism.

The subsections that follow discuss Isis flow control in detail; the tactics used are summarized in Table 2. In reading this table, one should keep in mind that Isis buffers its own messages in the application address space. The size of this buffering region is thus one measure of congestion. However, UNIX will also be buffering some amount of data, and Isis is not able to determine how much of this data is in use or whether incoming messages are being dropped because of a lack of buffer space. The messages buffered within the application are normally retained until they become "stable", meaning that there is no further risk that they will need to be retransmitted to ensure atomicity or to overcome packet loss.

3.2 UDP transport performance

Output flow control in the UDP layer is governed by three mechanisms:

- The basic form of output flow control is to inhibit message transmission when too much data is in transit, or when the remote end of a connection lacks space to accept further messages. Isis implements this form of flow control at two levels:
  - One approach is to block a sending task if too many input messages have yet to be consumed, with the goal being to encourage the consumption of input messages. Of course, consumption of input messages may also trigger the creation of new tasks, so this rule can only be applied in certain conditions.
  - A second form of flow control is based on windowed acknowledgement: in this approach, the number of messages in transit to a remote node is limited, and when the limit is reached, transmission of new messages ceases until the current set of messages has been consumed. Our implementation of sliding windows allows the recipient to place back-pressure on the sender, by leaving messages unconsumed in the input window of a channel. Such a situation is detectable by the sender,

which will cease to transmit until the window is drained below the threshold by the receiver. However, one must also be concerned about a receiver that is congested precisely because it has not yet received some message which this tactic might leave trapped in the window, such as a message that will cause a lock to be released. A positive/negative acknowledgement scheme controls this windowing mechanism, using rules that suppress the frequent transmission of acknowledgements when data appears to be streaming from sender to destination, but increases the acknowledgement rate when the pattern suggests a datagram style of interaction, in which the sender may be delayed waiting for an acknowledgement for even a single packet.

- An additional form of flow control is concerned with the O/S overload problem on the sending side. Isis uses a virtual window to limit the total rate of data transmission through the sender-side operating system, with the goal of avoiding an overload in which the operating system might lose large numbers of messages before they have been sent. This virtual window has two control components. The first involves a sum of data in transit, computed over all open inter-process communication channels, with acknowledgements and retransmissions for any channel also being applied to a running estimate of the amount of data in this window. The second control component is temporal: as time elapses, Isis drains data from the sender's window using a purely temporal formula intended to model the O/S mechanisms that, hidden from Isis, are presumably draining outgoing sockets and outgoing communication backlogs and queues.

As noted earlier, backpressure is associated with a form of input flow control. The basic input flow control mechanism in Isis is to delay delivery from an input window if too much memory is in use. On the other hand, memory consumption can rise rapidly during periods when the application is buffering messages while waiting for a lock to be released, an ordering message to arrive, a new group view to be installed, or some other condition. Delaying the very message that will release the lock would then exacerbate the overload! Thus, the flow control algorithm faces a problem of second-guessing the application.

For example, consider the flow control that would be best for an application such as:

```
grep ''Subject" mailbox | grep -v ''Horus'' | uniq | sort
```

in which the "grep" command generates messages that flow down the pipe towards the right. We can understand this example as an instance of a pipeline data flow pattern: $p \rightarrow q \rightarrow r \rightarrow s$. Now, suppose that s is slower than p. Clearly, data will need to pile up somewhere in the pipeline, and eventually p should be forced to stop initiating new messages. A simple way to do this is to limit the amount of buffered data in p, q and r. But now consider a slightly different application, in which r is responsible for sorting data for certain messages and forwarding others, as in this case, where the sort is performed by process r:

```
grep ''Subject" mailbox | grep -v ''Horus'' | sort | uniq
```

In both cases, r will experience a growth in buffered data, but whereas in the former case, an appropriate tactic is to put backpressure on q, waiting until data has drained from r to

s, in the latter case r should continue accumulating messages until it sees the end-of-file, at which point it can sort them and start emitting messages to s. In the second situation, any form of back-pressure would just slow the programs down, since r must wait for the end-of-file before it can sort its input! We would call the former problem a *pipeline flow control* scenario, while the latter one requires *I/O rate matching* (to avoid message loss due to overruns) but not pipeline flow control.

Backpressure is used by Isis to implement pipeline flow control. By setting a parameter, isis_pipeline_threshold, the application developer specifies a limit on the amount of memory that a process can consume before it begins to inhibit input, causing backpressure to the upstream sender. If r sets this parameter, for example to 500kb, than after 500kb of data accumulates in r, queuing will begin to occur in q. Normally, in such a system, q would also set this parameter, so that the backpressure will eventually inhibit the generation of new data in p. If neither q nor r uses backpressure, only rate matching will be done, and the potential exists for data to accumulate in the pipeline.

The reader may wonder why Isis does not automatically enable the backpressure mechanism. We have experimented with schemes to do this, but have not been able to discover a suitable heuristic for recognizing pipeline situations. As illustrated by the second example above, the risk is that if a non-pipelined application is subjected to a backpressure flow-control algorithm, the result will be very poor performance or even deadlock, e.g. if r is forced to wait indefinitely for a message (like the end-of-file) that is indefinitely delayed within the back-pressure buffering software!

We have been unable to design experiments that isolate specific elements of our flow control algorithm in a way that illustrates precisely how each element enters into the overall picture. However, we have instrumented the behavior of Isis with the whole mixture of flow control mechanisms in place, as illustrated in Figures 4-11. These graphs compare the performance cbcast (our less ordered, lightweight protocol) with that of abcast (our totally ordered protocol) for different group sizes, showing asynchronous and RPC latency (a multicast in which one reply is requested, from a single destination other than the sender) and throughput in messages and bytes per second. Here, one sees that the more ordered protocol has significantly greater end-to-end latency, but comparable throughput except for very small messages. The latency graph was computed for UDP transport; IP multicast latency was found to be constant and comparable to the two-member UDP case.

Our graphs do not include the impact of a special case for the abcast protocol. In Isis, an abcast initiated by the *oldest member of a process group* is transmitted using the cbcast protocol.[4] In a group where communication is randomly spread over the members, performance will thus be a weighted average of the cbcast and abcast figures.

[4]As originally developed, the Isis abcast protocol uses a token that can be moved from process to process. The token holder sends its own multicasts using cbcast, while non-holders use the slower abcast protocol. The current system does not implement a movable token because Isis itself generates abcast traffic from the oldest process in a group, which would tend to move the token back to this process in any case. Instead, Isis supports a token tool, and the holder of a token assigned by this tool can use cbcast instead of abcast, obtaining abcast ordering. Programmers of bursty applications are strongly encouraged to adopt this approach.

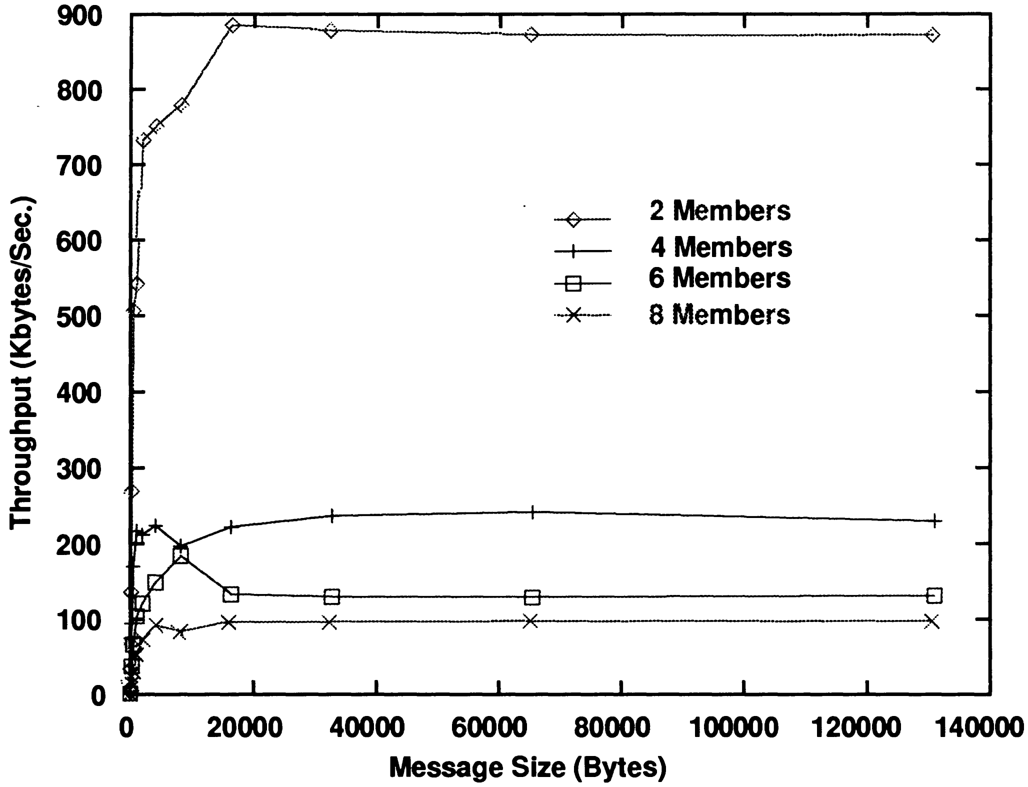

Figure 4: Asynchronous Cbcast Throughput in Kbytes/Sec.

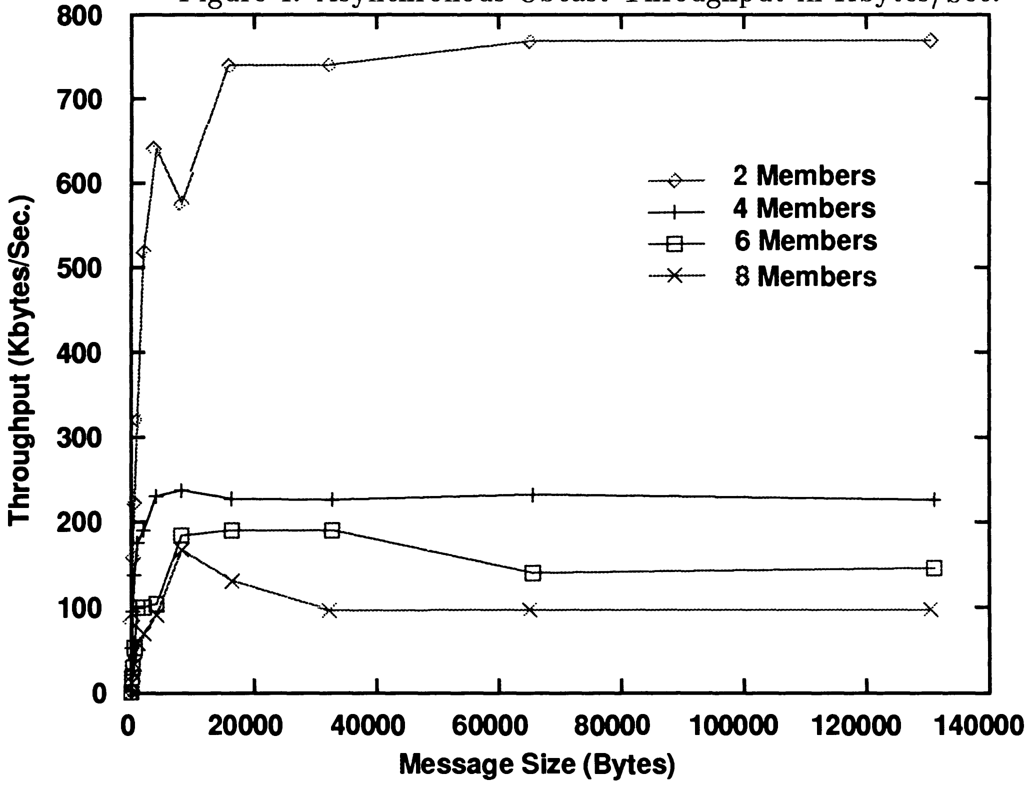

Figure 5: Asynchronous Abcast Throughput in Kbytes/Sec.

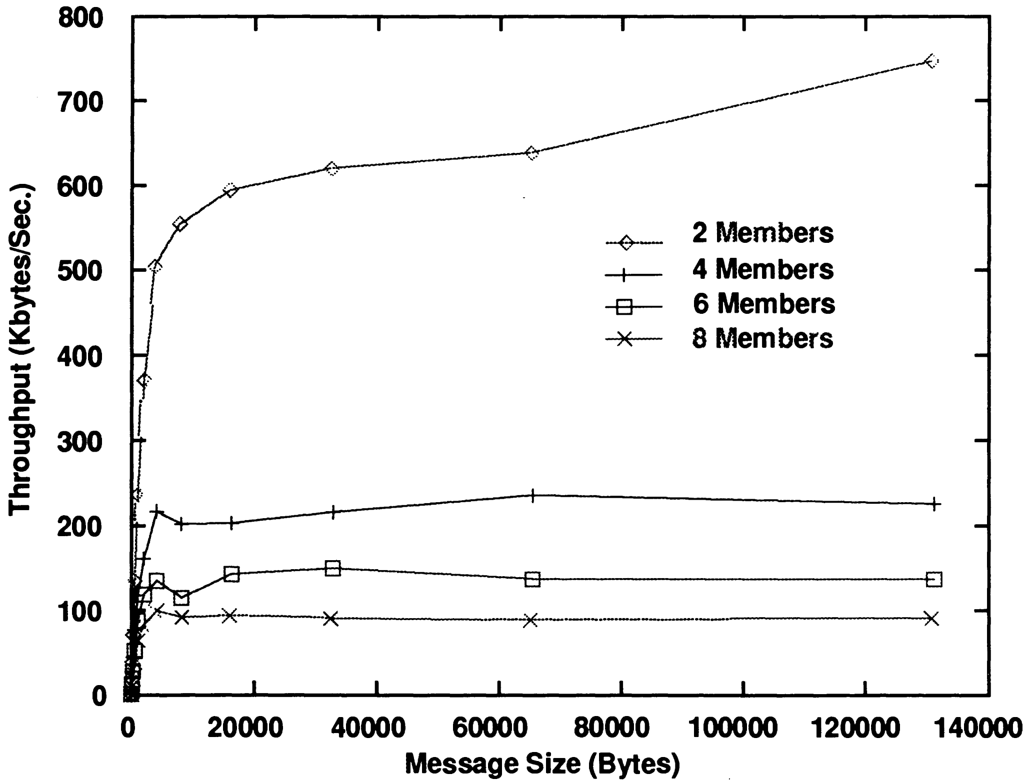

Figure 6: Synchronous Cbcast Throughput in Kbytes/Sec.

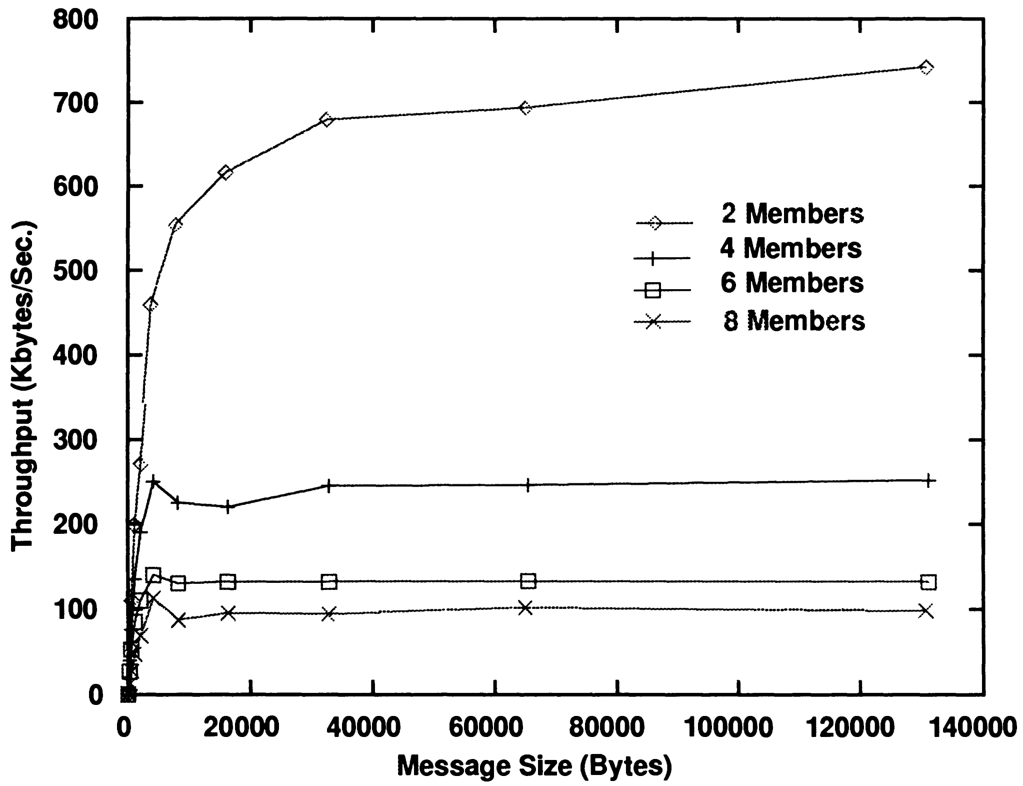

Figure 7: Synchronous Abcast Throughput in Kbytes/Sec.

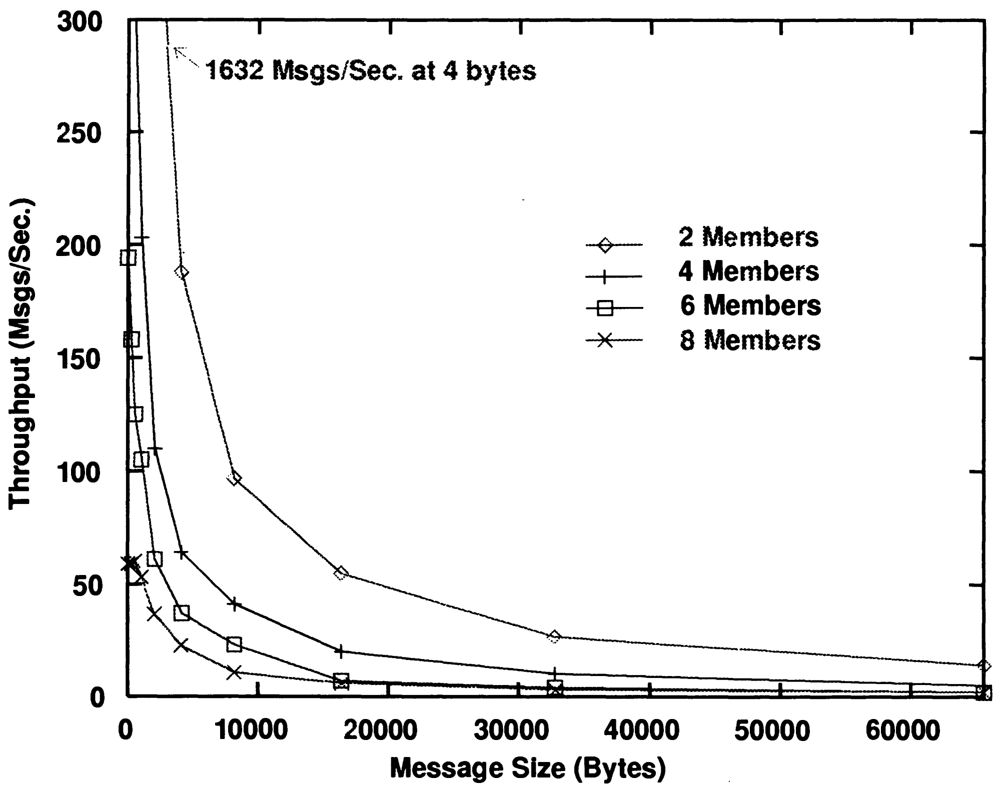

Figure 8: Asynchronous Cbcast Throughput in Messages/Sec.

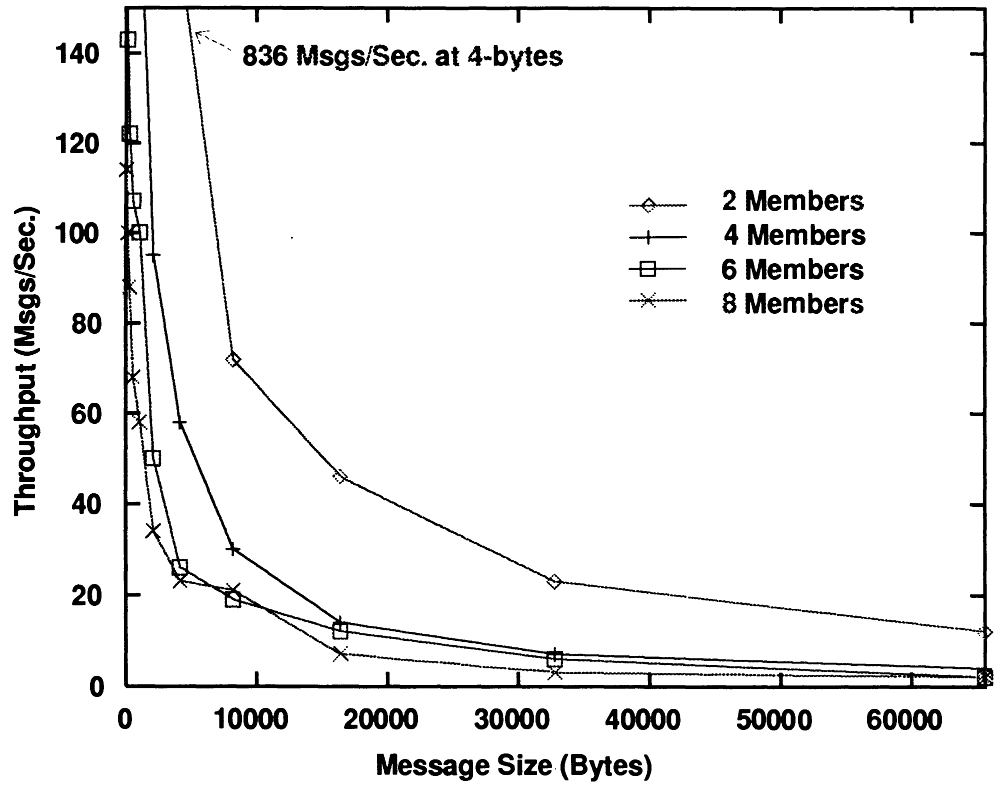

Figure 9: Asynchronous Abcast Throughput in Messages/Sec.

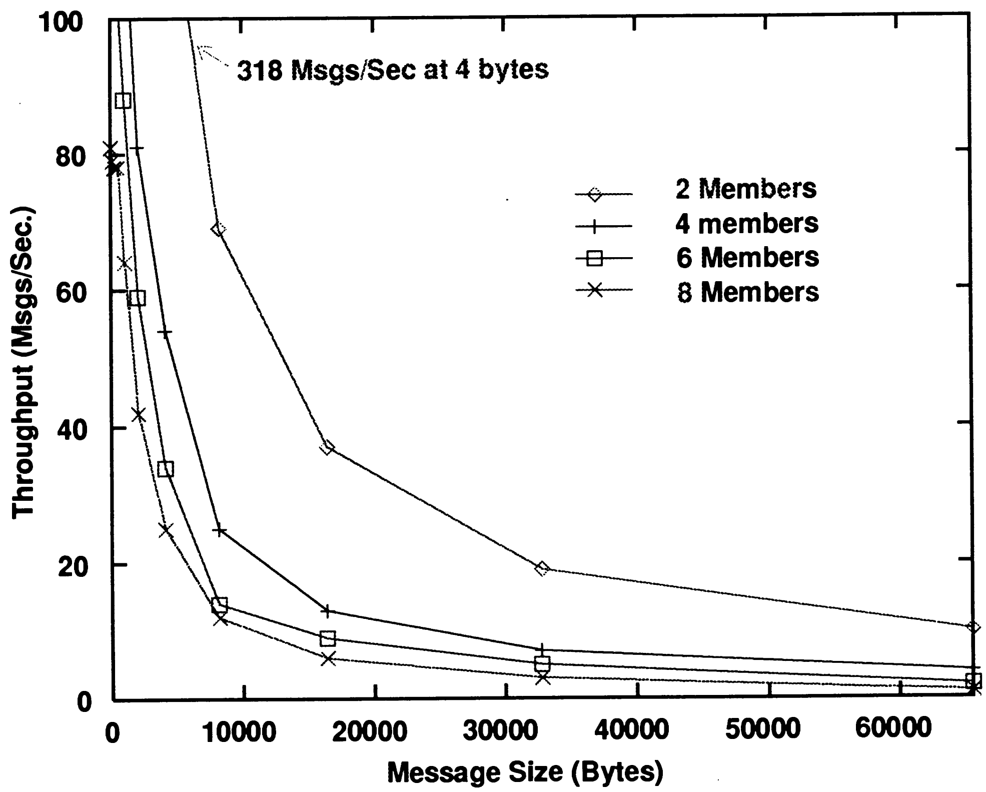

Figure 10: Synchronous Cbcast Throughput in Messages/Sec.

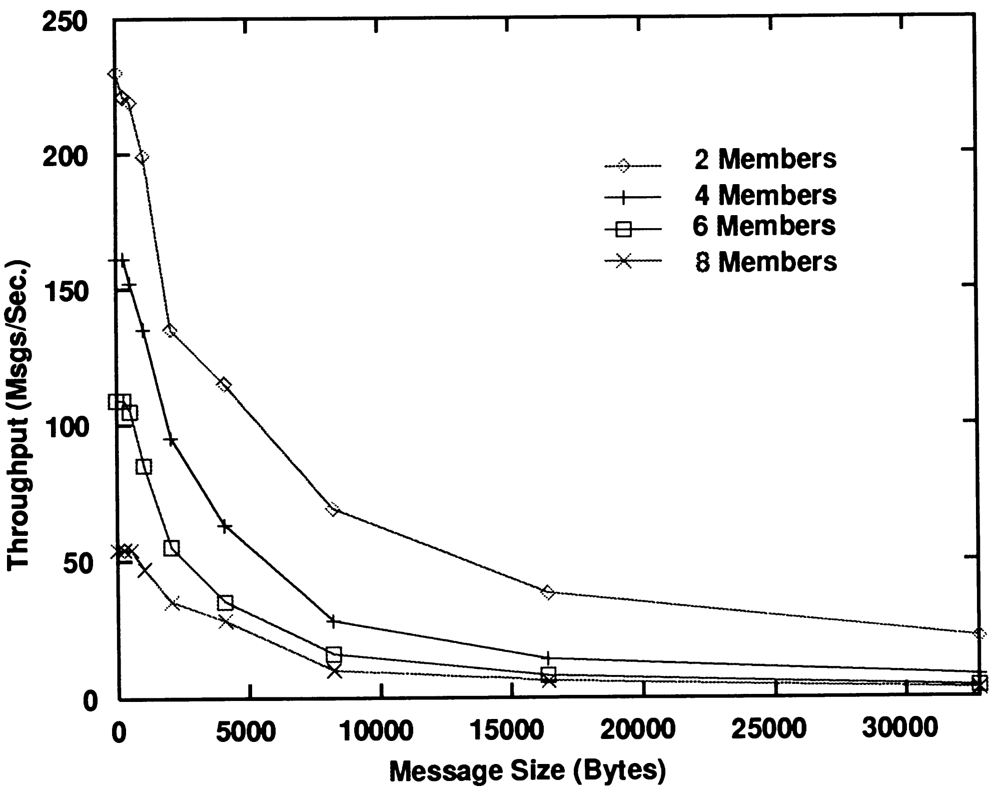

Figure 11: Synchronous Abcast Throughput in Messages/Sec.

3.3 Multicast transport performance

Whereas the UDP flow control mechanisms described above are quite complex, flow control for the Isis IP multicast transport is considerably simpler. Most important among the reasons is that because IP multicast can trigger storms, Isis users have been cautioned to only use the mechanism in settings where a small number of sources will be sending bursts or streams of data, and where no single machine will receive a rate of multicasts (even from overlapped groups) that approaches the performance limits of the operating system and interface. These cautions let us assume that within IP multicast, communication is far more regular than within the UDP layer of Isis. Unlike in the UDP case, where significant packing is done and this results in very different communication to different destinations, the IP multicast layer is concerned with sending the same message to all members of a group, and can optimize for bursts of multicasts from a single source to a fairly stable set of destinations.

Multicast flow control remains complex, however, in its need to handle message loss and in how it sends acknowledgements and negative acknowledgements. Previous work on multicast has favored ring schemes, in which some form of token circulates within a group, carrying messages and acknowledgement information [5, 6]. A multicast with point-to-point acknowledgements is often rejected because of the presumed linear performance degradation, even when each ack may cover several multicast messages. In our work, these approaches were compared. For use in Isis, somewhat surprisingly, ring-style protocols did not perform as well as the point-to-point algorithm.

To permit this type of experiment, our multicast flow-control and error correction approach has two components. Basic flow control occurs on the output side only, with no form of explicit back-pressure currently supported. As illustrated in Figure 2, the publisher maintains a pair of output windows for each process group, one much smaller than the other (labeled as "D" in the figure).

The basic approach is to use the small inner window for a first-level flow control to the "typical" group member. As acks are received, a message will move from the inner window to the outer one once a sufficiently high percentage of acks have been recorded, at which time new multicasts can be transmitted. After experimenting with appropriate values for this percentage, we settled on an approach whereby each packet remains in the inner window until half of its destinations have acknowledged it. A packet in the larger outer window is held there until *all* destinations have acknowledged it. (The handling of failures in which the sender crashes is through our reliable multicast protocols, which reside at a higher level of Isis and have been discussed in detail elsewhere).

The second component of our scheme is the positive and negative acknowledgement (ack/nack) protocol. We compared three approaches:

- In the simple point-to-point approach, each recipient can be viewed as sending an acknowledgement for each packet. However, acknowledgements are delayed to avoid sending them more often then about once every 100ms. Nacks are sent the same way. Moreover, we merge acks, so that a single ack packet can acknowledge many packets. Similarly, a nack packet contains the range of missed sequence numbers, so that the sender can retransmit the entire range of missed packets, if necessary. When a packet must be retransmitted, our approach is to multicast it again if several destinations have not yet acknowledged it, and to send it point to point if only a single destination has nacked the packet, or after multicasting it twice.

- In a *multicast ack* scheme, each ack and nack is multicast, and a process suppresses its own acks and nacks when some other process acks the same message that it was going to ack, up to a maximum delay of approximately 100ms. The idea is to convert the algorithm to a "mostly nack" scheme

- In a ring ack scheme, ack messages cycle around the group from member to member. Nacks are still sent point to point.

On detailed study of the behavior of the three approaches, we found that congestion on the interface of the sender accounts for the linear rise in cost seen by the first protocol. Packet loss on the recipient interfaces rises sharply with the multicast ack and nack scheme, and this causes degradation for high rates of message transmission. It would seem that on the machines we used for our tests, reception of as few as three back-to-back packets could provoke a significant loss rate. Finally, the token ring ack scheme performed well for small groups of lightly loaded processes. But it degraded significantly when background load or computation in the participants caused even small delays in servicing the token message upon arrival. The same type of small random noise had little detectable impact on the point-to-point acknowledgement method.

Our performance measurements include some "spikes" that were caused by the tendency of the UNIX system with which we worked to run short on kernel memory and drop packets at very high rates of multicasting. For example, although we were able to tune the IP multicast algorithm to achieve a throughput of 980KB/second from one sender to five destinations, for large messages and with large window sizes, this level of performance was only seen for unloaded machines and an unloaded network. On runs where these conditions did not hold, UNIX would begin to drop packets, and the resulting point to point retransmission overhead would reduce performance to something far lower (32Kb/second in one case, where UNIX ran low on memory in the destination machine). The performance reported above, in contrast, was reproducible over a wide range of background conditions. Intermittant packet loss is also the cause of the tail-off seen in Figure 14 for large messages, which is not seen for the (slower) abcast protocol, and the low performance of the asynchronous IP abcast protocol for very small packets in Figure 15. The sustained stream of large asynchronous cbcasts used in the former case triggered a low rate of packet loss in UNIX, while loss rates for the abcast graph were apparently due to the two-phase nature of the protocol, which resulted in back to back packets that triggered packet loss near the interface.

As noted earlier, loss of IP-multicast packets is not reported to the application by UNIX, and must be deduced from statistics maintained by the application itself, a challenging problem. This led us to tune the software using smaller windowing sizes, at which heavy UNIX loss rates are avoided.

Developers of TCP protocols have reported similar problems, and TCP protocols that dynamically vary window sizes are common. We believe that similar techniques could be adapted to the IP multicast case, and plan to explore this in future work.

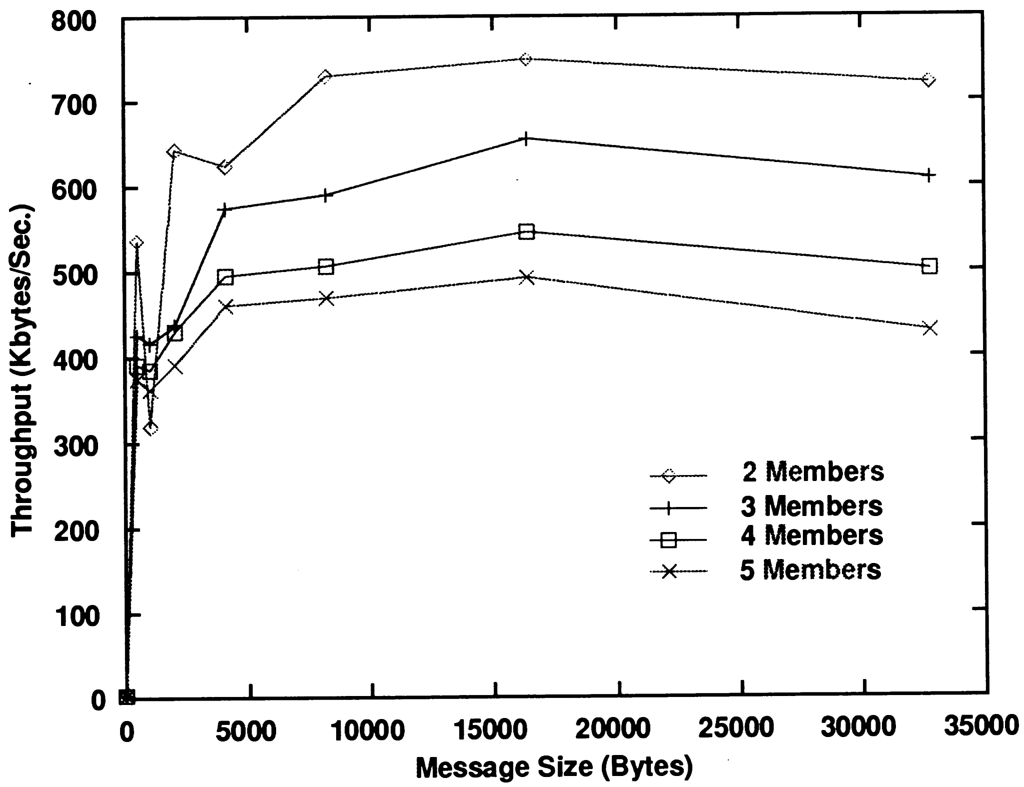

Figure 12: IP Multicast Asynchronous Cbcast Throughput in Kbytes/Sec.

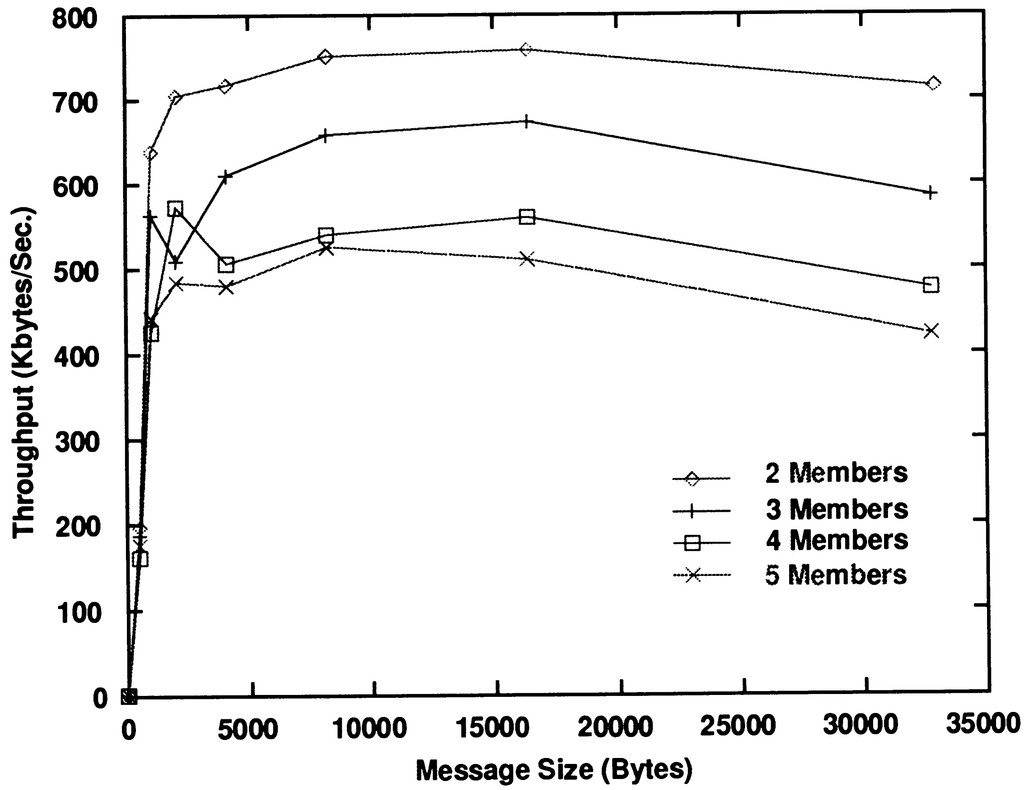

Figure 13: IP Multicast Asynchronous Abcast Throughput in Kbytes/Sec.

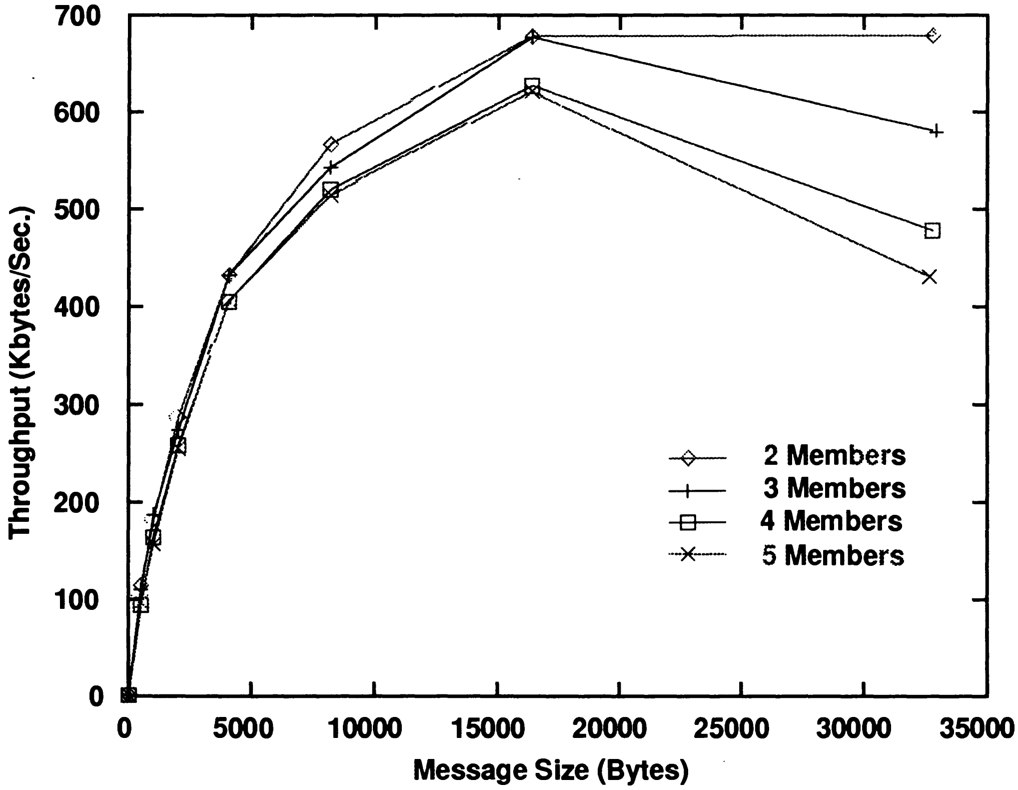

Figure 14: IP Multicast Synchronous Cbcast Throughput in Kbytes/Sec.

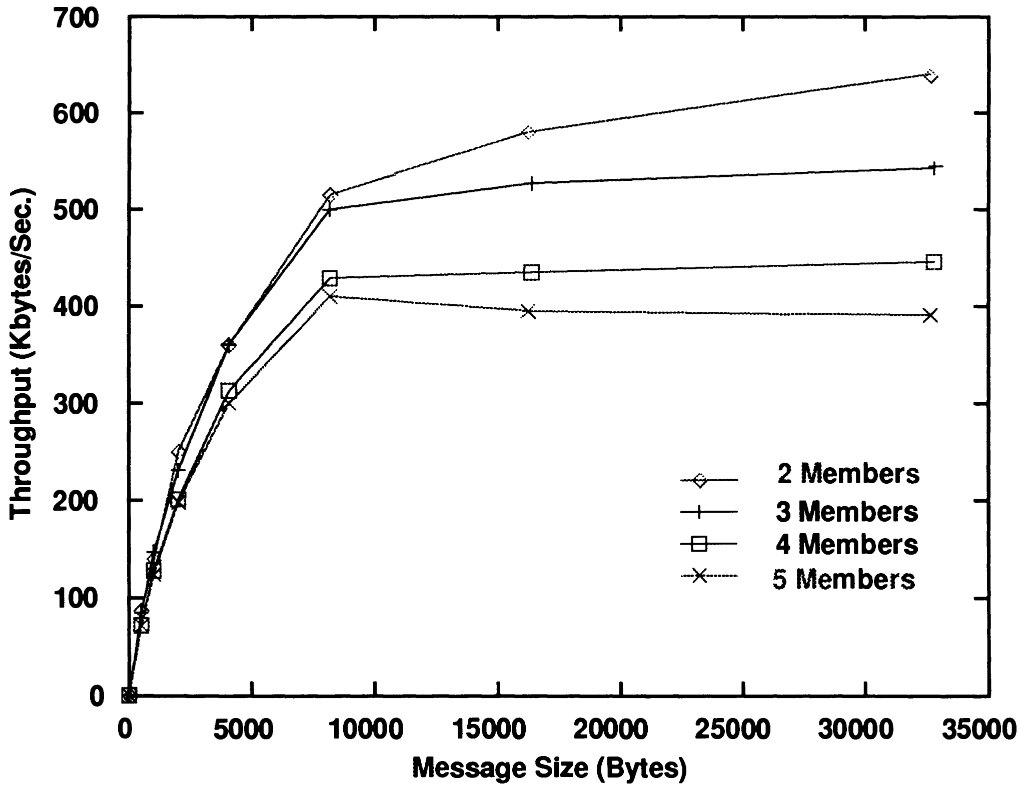

Figure 15: IP Multicast Synchronous Abcast Throughput in Kbytes/Sec.

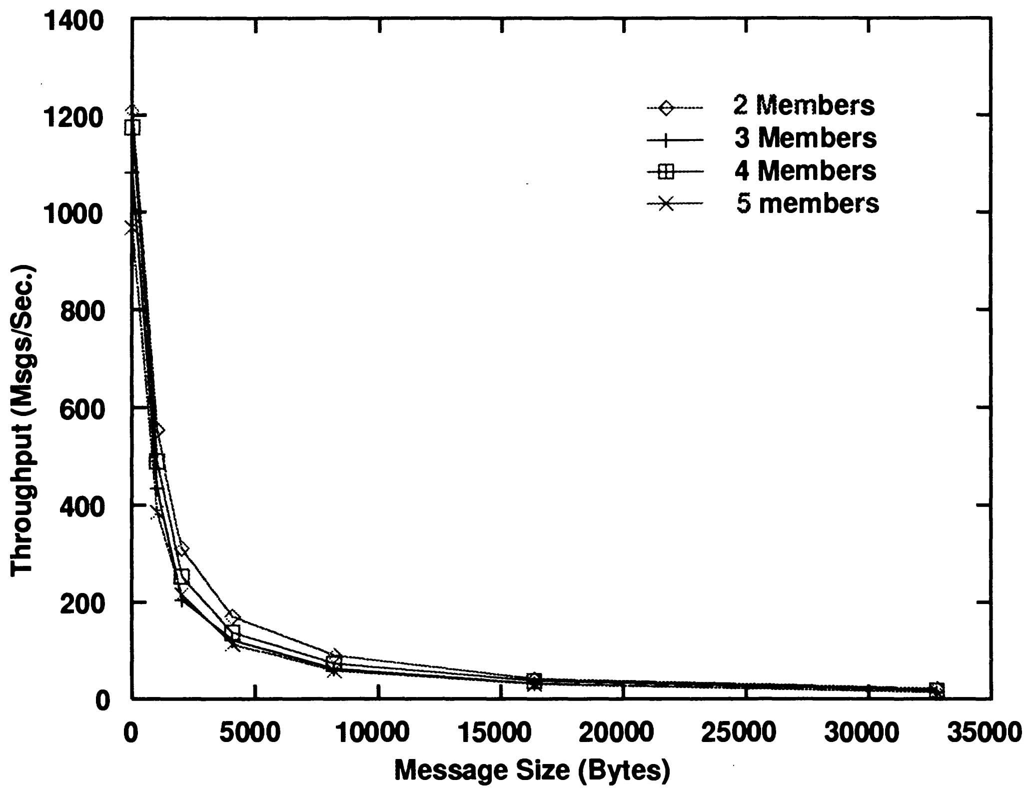

Figure 16: IP Multicast Asynchronous Cbcast Throughput in Msgs/Sec.

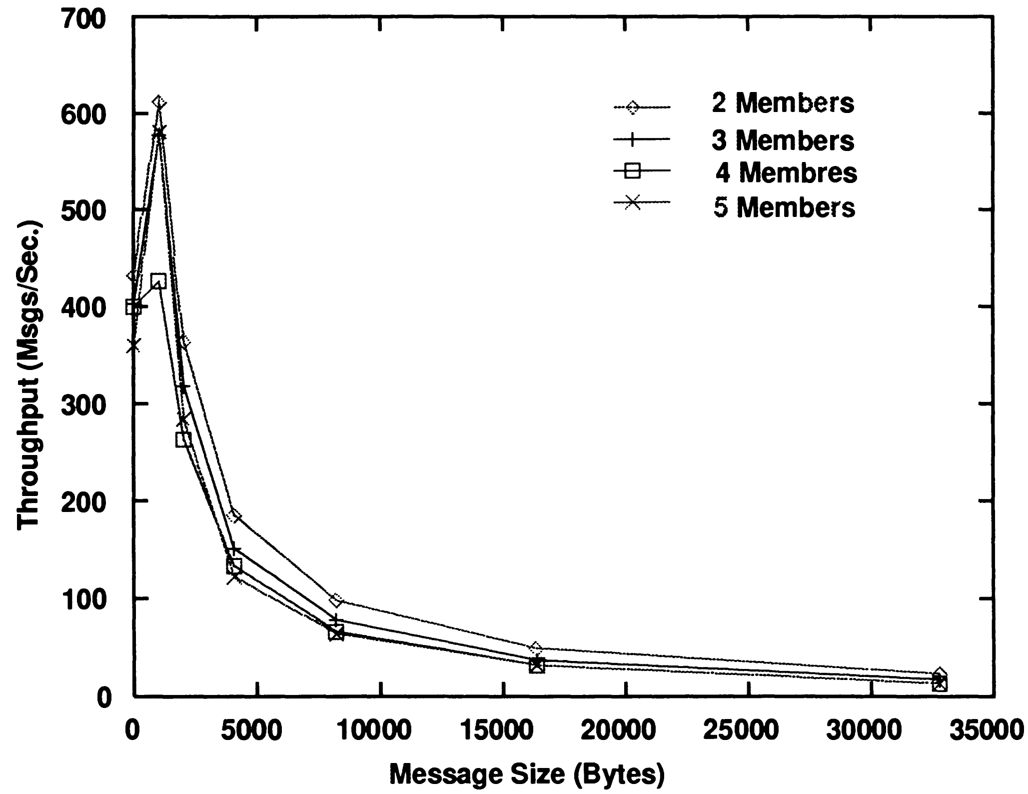

Figure 17: IP Multicast Asynchronous Abcast Throughput in Msgs/Sec.

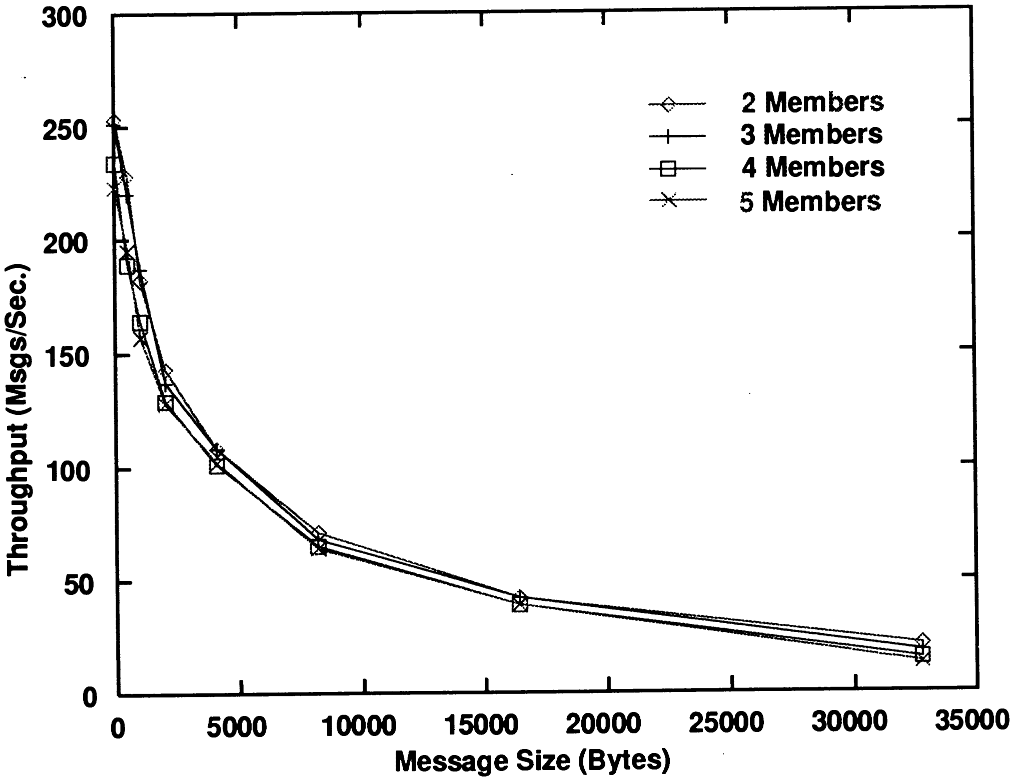

Figure 18: IP Multicast Synchronous Cbcast Throughput in Msgs/Sec.

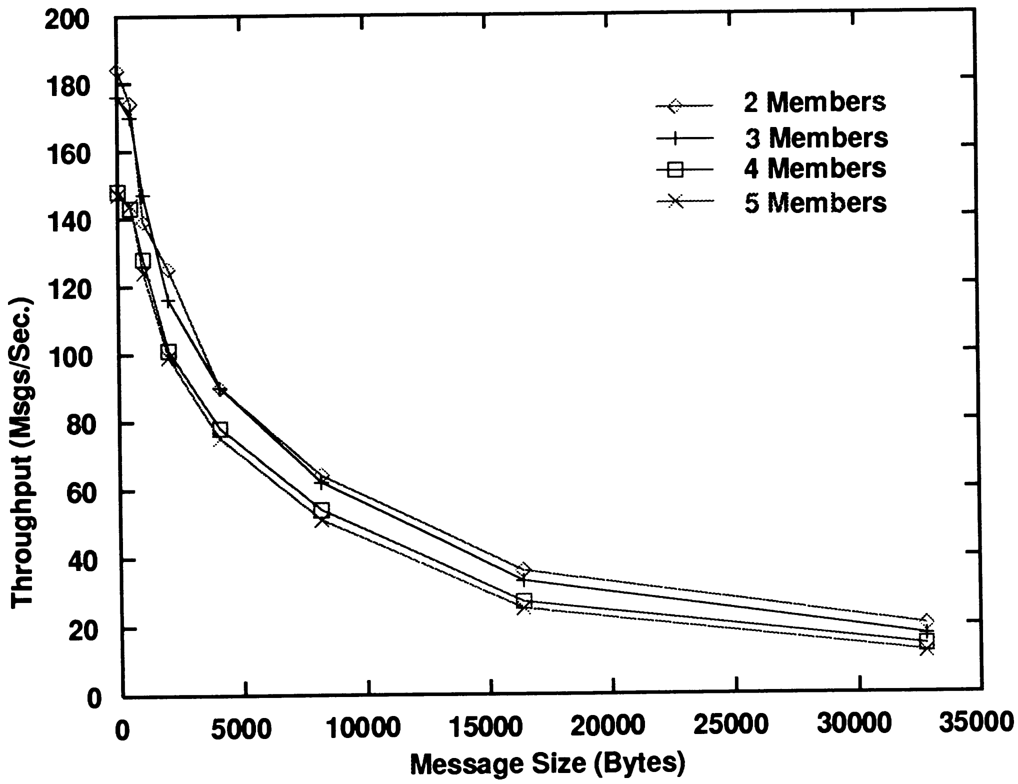

Figure 19: IP Multicast Synchronous Abcast Throughput in Msgs/Sec.

When we experimented with applications in which all processes used IP-multicast in the same group, we found that very high rates of loss resulted if many senders transmit simultaneously. In such situations, a mutual exclusion scheme worked well: a small number of tokens (3 or 4) were permitted to circulate among the group members, and to initiate a multicast, a process was first required to acquire a token. As mentioned in the discussion of abcast performance, Isis has a tool for this type of mutual exclusion. Using the token tool, we were able to obtain extremely high performance by limiting the number of concurrent senders in each group. In effect, the Isis toolkit thus offers the application programmer a way to layer additional flow control mechanisms over the flow control done in the IP multicast software itself. This avoids extra complexity in the IP multicast flow control layer, and perhaps more importantly, avoids the need for our flow control to detect the communication pattern present in the application.

This view of the application program and the communication subsystem working in concert to achieve flow control is one that appeals to us, and we are now exploring ways of extending the Isis interfaces to encourage more explicit cooperation between communication-intensive applications and our flow control layers. For less intensive uses, on the other hand, we want our basic mechanisms to work well.

3.4 Virtual Synchrony and Ordering Costs

Moving to higher layers of Isis, the most interesting issues concern the delays associated with cbcast and abcast ordering, and with the flush protocol used to install new process group views.

The role of the Isis flush protocol is to identify and deliver any pending multicasts that, according to the Isis execution model, need to be delivered prior to the delivery of a new group view. While the protocol is running, new multicasts are delayed, so the flush protocol is concerned only with multicasts that are already in progress at the time that it is started. The protocol operates through an exchange of copies of pending multicasts and of flush "mark" messages, and is very similar to an optimized version of the consistent cut algorithm of Chandy and Lamport [4]; the detailed protocol used is discussed in [2].

The cost of the flush protocol is shown in 21. This graph explores the impact of group joins and leaves on message latency, using a group of three members, which are periodically joined by a fourth member that loops issuing joins and leaves. In the background, one member (the oldest one) continuously sends cbcast messages, as rapidly as possible. The latency between the initiation of each multicast and delivery was measured. As can be seen from the figure, there is a increase in message latency associated with group size, on which is superimposed larger latency increases ("spikes") corresponding to the cost of the group flush protocol, which runs after each join or leave event. Smaller spikes represent background noise on the network, which was not isolated from other users during the test; the noise level was found to be similar when the same tests were performed but without having a fourth process join and leave.

From this study, we concluded that the effect (on latency) of adding a fourth member to the group is to significantly perturb latency during a period of about 20ms, while the flush protocol runs, after which a discrete "step" in average latency occurs, varying in size from

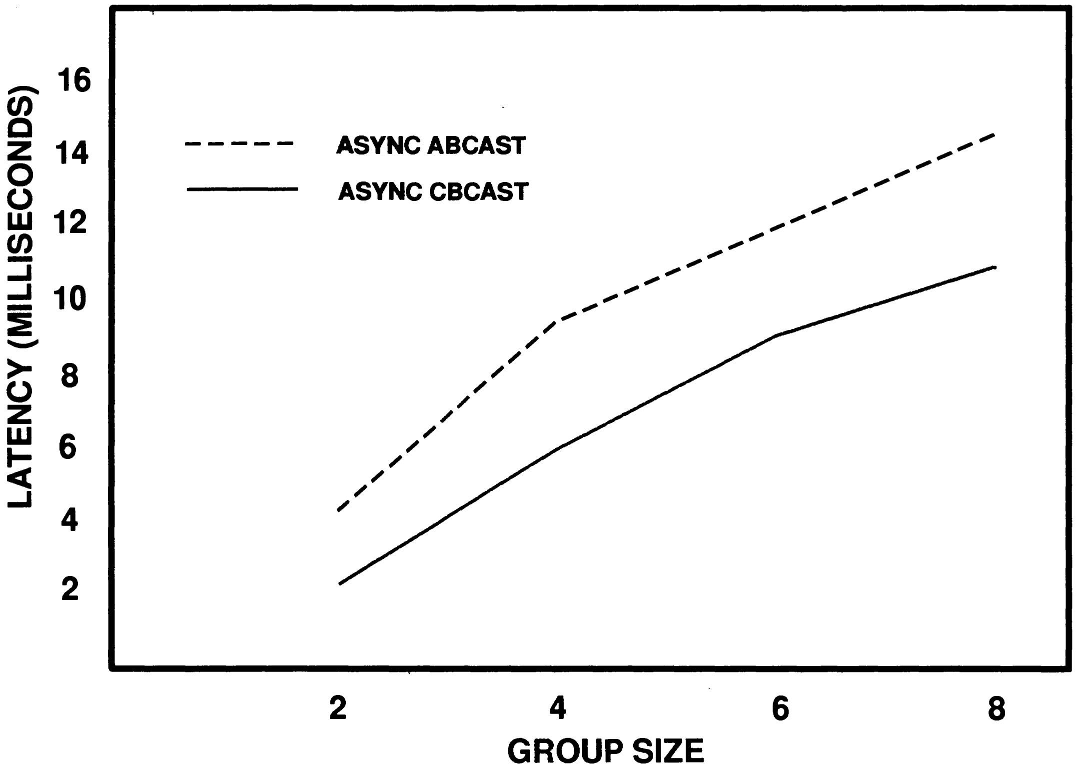

Figure 20: Asynchronous Abcast and Cbcast Latency over UDP Transport

approximately 4 ms. for 3 members to about 5 ms. for 4 members. These figures can be used to predict the responsiveness of Isis to process failures and join requests under constant load (when the failure detector timeout is set to 0).

One interesting issue, both from a protocol perspective and a performance one, concerns the integration of IP multicast data streams with UDP data streams in a single process group. Our communication protocol is designed to allow messages within a single group to follow different paths to the destination, with Isis handling the interleaving of the incoming messages to obtain the ordering and virtual synchrony guarantees of the system. Unfortunately, the requirement for brevity precludes a detailed analysis of the cost impact of this mechanism, which in any case is not expected to find common use in Isis applications. At present, most use of Isis employs groups that run either entirely over the UDP transport or entirely over the IP multicast transport, hence the impact of this merge mechanism on the typical application is insignificant.

3.5 Discussion

Flow control and resource management are poorly understood in complex distributed systems, and may not be amenable to completely general solutions when working over existing operating systems, because of their high loss rates and poor loss reporting mechanisms. Al-

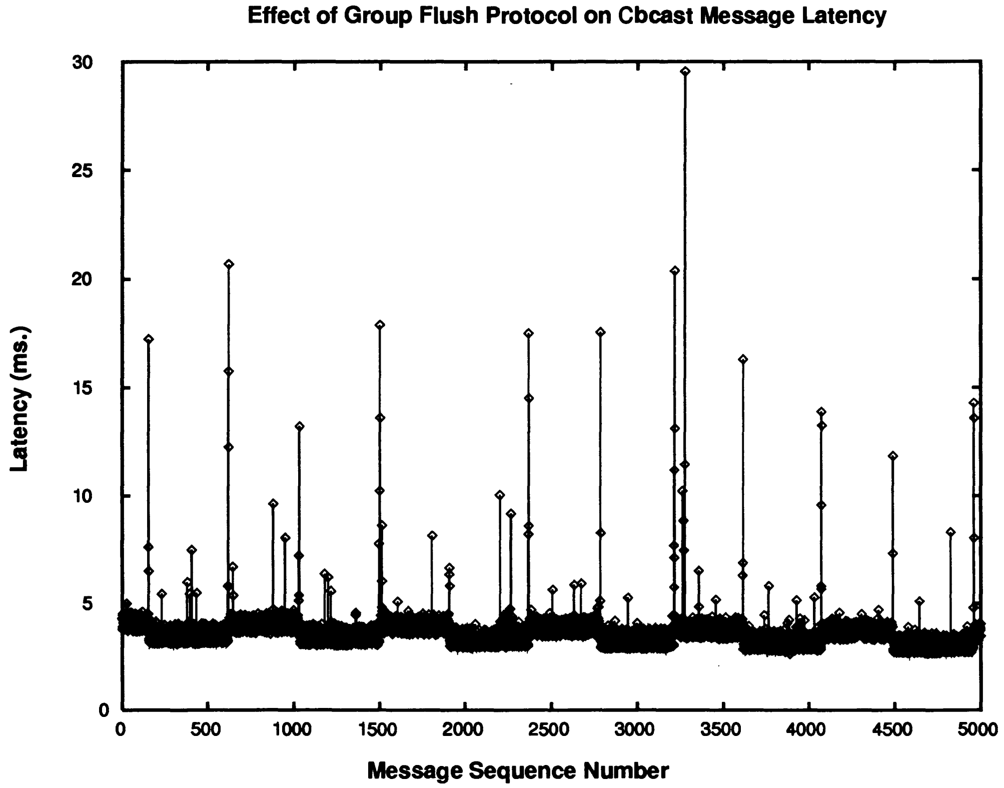

Figure 21: Effect of Group Flush Protocol on Cbcast Message Latency

though flow control in Isis is complex and heuristic, our studies of the literature suggest that very little is known about resource management in complex message passing systems, and that even virtual circuits are difficult to deal with dynamically.

It is a matter of some annoyance to us that operating systems such as UNIX provide very little information about their memory resources and state. Although monitoring *programs* are generally available, it is rare to find any system call or error code that would offer an accurate way to discover that a message was lost on the sending side, or that report any sort of information about a packet that was explicitly discarded on the reception side. As a result, information that is actually available within the operating system, and that could be invaluable for improving performance, is discarded - forcing the application to employ contorted heuristics to model what is apparently occurring in an approximate manner. For example, if instead of discarding a UDP packet because of a lack of memory, UNIX were to truncate it and deliver only the first few bytes, it would be easy to solicit retransmission of the lost data without first waiting one second or more for a retransmission timer to trigger. (Faster timers would only clutter the network with retransmissions and increase load, thus increasing loss rates). A minor change to UNIX could then have a substantial performance impact on Isis.

Advances in hardware, together with O/S mechanisms intended to greatly reduce message loss and to increase the information available to communication-intensive applications might render the flow control question moot. However, on current networks and operating systems, applications that seek to fully exploit the facilities stand as a major challenge to the designer of a general purpose communication packages.

4 Multiple-Group Performance Issues

Readers familiar with Isis will be aware that the current versions of the system use what we call a "conservative scheme" for enforcing multi-group causal ordering. Basically, if a process has been sending or receiving messages in a group g, but now wishes to send a message in g', the transmission of the latter message will be delayed until messages in g become stable. That is, the previous messages must be delivered to all of their destinations.

The problem seen here may actually be familiar to readers who have logged into a remote computer over a telnet line on which high latencies are observed. Because TCP can buffer 8k bytes, and the UNIX terminal software an additional 1-2k bytes, hundreds of lines of output can be generated and buffered in the channel between the remote machine and the user. If the user tries to interrupt a remote computation, the effect of the interrupt may not be seen until all of this output has been displayed - a very annoying problem if a program has gone into an infinite loop or is taking an undesired action! On the other hand, if a version of UNIX were to conceal this (e.g. by not printing the buffered data), the user might be left with a misimpression that the interrupt took effect much sooner than was actually the case. One thus faces a choice of the desired semantics, and UNIX can be understood as chosing the more conservative alternative. The causality delay in Isis reflects an analogous ordering obligation: the backlog of messages being transmitted to g can delay a message to g', unless one is willing to settle for a weaker ordering guarantee than is the default in Isis.

Group switching delays can be zero if communication occurs less frequently than the time needed for a multicast to become stable, which is given in the graph measuring latency for synchronous communication. In fact, if an application switches groups on every multicast, the resulting performance is the same as for synchronous communication (that is, asynchronous multicasts sent to a sequence of different groups will give the same performance as synchronous performance within a single group). On the other hand, if an application switches between groups after sending a burst of asynchronous messages, latencies will grow in proportion to the size of the backlog of data that was buffered by the transport layers of Isis; if a destination is particularly slow, such a delay can be as large as several seconds.

Group-switching delays of arbitrary size cannot arise, because there are backlog limits in the communications layer; when these thresholds are reached, tasks attempting to initiate new asynchronous multicasts block at the time the multicast is issued. The worst case delays arise if a destination p in group g fails, preventing a process q that has an unstable message to p from sending subsequent messages in group g'. In this situation, q may have to wait until the failure of p is detected, which can take several seconds. This behavior illustrates a design decision, which is to guarantee safety even at the risk of communication delays.

Not every application can tolerate potentially long delays when switching between groups.

Faced with this problem, an Isis user can explicitly request weaker ordering properties from Isis. As above, our philosophy is to do the best we can for the general case, but to offer the programmer tools for dealing with particularly demanding situations in which the general mechanisms are inadequate.

5 Conclusions

This paper presented the major flow control mechanisms in Isis, summarizing experimental results obtained by running a variety of communication patterns over the system. This real-world approach has significant disadvantages: by not isolating the impact of specific mechanisms, it is hard to see what role each plays, and because Isis is a complex system, the paper is forced to treat performance at something of a broad-brush level. On the other hand, the fact that we report behavior of a real system has, we think, some virtue. Much prior work has focused on a single mechanism at a time, and often in the absence of atomicity guarantees, flow control, and group membership synchronization. The lack of any execution environment can give unrealistically inflated performance figures, and can hide the sensitivity of a protocol to aspects of the runtime environment.

For example, we found that token-based multicast flow control did not work well in the Isis system, whereas other researchers have had good results with this approach. In our work, at least, the overhead of the Isis tasking and message passing system, combined with background load on the machines we used, network load, and scheduling delays, makes participants slow to react when the token reaches them, causing performance delays. In effect, the algorithm is overly sensitive to load on the recipients. A very simple "multicast-out, point-to-point in" scheme for acks and nacks performed far better, presumably because it is insensitive to the order in which acks are received and hence the sender can do useful work even if one of the participants is temporarily slow to respond.

It is unclear to us if Isis performs as well as possible, but we now believe that it performs well given the constraints and context on which these tests were performed. Major improvements should result from exploiting shared memory for local communication, porting Isis to operating systems on which packet loss in the kernel can be controlled or reported to the application, operating over faster communication hardware (such as ATM), and by running on dedicated processors. All of these directions are under investigation at this time.

This paper leaves open a number of areas that are in need of much more study. The whole issue of guaranteed data flow and guaranteed bandwidth represents a major challenge for packet communications systems. Our research shows that under heavy load, flow control mechanisms can have a huge performance impact. The very large impact of unreliability in the O/S message layers might surprise many readers: clearly, O/S communication interfaces need to be re-examined, with an eye towards providing better resource exhaustion feedback on the sender side, and towards providing some small amount of information about packet loss on the reception side. Any information at all might lead to significant improvements in our algorithms. IP Multicast flow control also needs further study, particularly because our preliminary results suggest that a hierarchical fanout may offer the best performance in very large networks.

Overall, our research suggests that flow control, for demanding applications, is best viewed as a partnership between the application and the system. If the system does not provide the application with adequate information, and does not provide sufficient flexibility to allow a customized approach to flow control, the result will be a solution that works well for one purpose - RPC over a vendor package, for example, or TCP - but poorly for other types of applications. To the degree that we begin to see distributed performance as a cooperative effort in which the programmer and subsystem provide one another with a maximum of information and flexibility, very high performance can be achieved.

Acknowledgements

The discussion in this paper benefitted from suggestions and comments by Robert Cooper, Brad Glade, Guerney Hunt, Dan Julin, Cliff Krumvieda, Stef Lucas, Heather Mardis, Tom Paquette, Robbert van Renesse, Ian Service, Will Sobel and Gautam Thaker.

References

[1] Yair Amir, L.E. Moser, P.M. Melliar-Smith, D.A. AAgarwal, and P. Ciarfella. Fast message ordering and membership using a logical token passing ring. In *Intl. Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[2] Ken Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *Transactions on Computer Systems*, 9(3):272–314, August 1991.

[3] Kenneth P. Birman and Robbert van Rennesse. *Reliable Distributed Computing Using the Isis Toolkit.* IEEE Press, 1994.

[4] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *Transactions on Computer Systems*, 3(1):63–75, February 1985.

[5] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *Transactions on Computer Systems*, 2(3):251–273, August 1984.

[6] M. Frans Kaashoek, A. Tannenbaum, S. Flynn Hummel, and H. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–20, October 1989.

[7] Rivka Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *Transactions on Computer Systems*, 10(4):360–391, November 1992.

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

[9] Brian Whetten. A reliable multicast protocol. Technical report, University of California, Berkeley, March 1994. In progress.