Incorporating System Resource Information into Flow Control

Takako M. Hickey and Robbert van Renesse*

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

Abstract

Upcall-based distributed systems have become widespread in recent years. While upcall-based systems provide some obvious advantages, experiences with these systems have exposed unanticipated problems of unpredictability and inefficiency. Incorporating system resources information into flow control is essential in solving these problems. Variants of window-based flow control suitable for distributed systems are investigated. Next, message packing, which improves network bandwidth usage efficiency, and, consequently, message throughput, is presented. Finally, a back pressure mechanism which controls admission of messages into the system by blocking applications at high load is presented. The combination of the window mechanism and the back pressure mechanism provides end-to-end management of system resources. The former manages network resources, while the latter manages operating system resources. The combination maintains good throughput even under high load.

1 Introduction

With the advent of threads, upcall-based distributed systems have become widespread. Such a system starts a thread for each arriving message in the destination process. This contrasts with the traditional approach of having applications wait or periodically poll the system for incoming messages. In an upcall-based system, a message does not have to occupy a system buffer indefinitely, waiting for the application which may never poll. Also, the application does not have to waste time polling the system for messages that may not be there. Examples of upcall-based communication systems include the x-kernel [6] and Isis [3].

While upcall-based systems provide some obvious advantages, experience with these systems has exposed unanticipated problems. The first problem is unpredictable performance. Predictability is not only desirable, but also a necessary property for applications such as a

*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, and AFOSR grant F49620-94-1-0198.

video-on-demand service, which requires a minimum number of bandwidth in order to reproduce video of acceptable quality. Unfortunately, most existing upcall-based systems will accept any request to send messages, whether they have the resources to send and deliver them or not. If not, messages may be dropped. A common situation is when many messages arrive at a destination from multiple sources at the same time, and messages are dropped because of a lack of buffer space. In such a congested situation, messages may be dropped again on retransmission, and, in fact, retransmission may contribute to the congestion [5].

A second problem with upcall-based systems is inefficient resource sharing. Here we are concerned in particular with resources that are reusable, but require exclusive access. Many resources fall into this category, including network buffers and threads. Sharing of such a resource is inefficient if the number of useful tasks that complete in some amount of time is much lower than optimal. For example, sharing of a network buffer is inefficient if only 10 message transfers that used the buffer have completed, while the buffer could have completed 20 message transfers in the same amount of time. Much of this is due to a contention of resource requests. When multiple requests arrive at an exclusive-use resource simultaneously, only one of them can be served. The other requests will incur extra overhead, either through resubmission (and some reprocessing of the requests), or through queueing of the requests. When too many requests for the same resource occur at the same time, the overhead can get prohibitively high.

Another reason for inefficient resource sharing are conservative allocation policies. Many systems limit the number of resources that a process can use, so as to keep them available for other processes. This results in underutilization of resources.

We claim that a single solution—resource management—can solve both unpredictability and inefficiency problems of upcall-based systems. A proper resource management scheme does not overcommit any resources and only accepts those tasks for which every required resource can be committed at an appropriate, predictable time. Also, proper resource management never refuses a request if it can commit every resource needed for a request.

Traditionally, resource management for message passing is done by a flow control mechanism. We believe that existing flow control mechanisms are inadequate for complex distributed systems. Current flow control schemes are executed at the network level, and manages only network-level resources. In upcall-based systems, sending and receiving messages require a variety of system resources at all the levels of the system, including network buffers, threads, and memory at both senders and receivers. If any one of these resources is unavailable, a message cannot be processed in a predictable fashion.

Thus, a proper flow control mechanism for upcall-based systems should take all these resources into account in regulating message traffic. If a receiver is low on a resource needed to receive a message, a sender should wait until the receiver has processed enough messages and has released a sufficient number of resources. Accounting for all resources required in the message passing avoids committing partial resources to those tasks that keep them unnecessarily, and allows efficient overall use of system resources. This is an end-to-end argument, as in [7]. A message should be sent only if the receiver is ready to receive it. Or, in the case of multicast, a message should be sent only if every receiver is ready.

In this paper, we study different flow control mechanisms in upcall-based systems, demonstrating a benefit of incorporating resource information into flow control. We will start by presenting a window-based scheme [8] generalized to multicast. We then describe *packing* of multiple buffered messages as a way to improve network bandwidth usage efficiency, and, consequently, message throughput. Finally, we describe *back pressure*, which blocks message send operations based on resource availability at the receiver. That is, the back pressure mechanism controls admission of messages into the system. A combination of the window-based flow control together with back pressure provides end-to-end management of system resources. The window-based scheme controls flow from operating system into the network based on network resource usage, and back pressure controls flow from applications to operating system based on operating system resources. With the combination flow control can maintain the peak throughput even under high load.

The rest of the paper is organized as follows. In Section 2, we explain our experimental methodology. We then present three flow control techniques in Section 3, Section 4, and Section 5. In Section 6, we relate our work to other work on flow control. We end the paper with discussions and future work in Section 7.

2 Experimental Methodology

We tested our flow control mechanisms in Horus [10], which is a dynamically layered, upcall-based system for group communications. In our experiment we simply added another layer designed to test various flow control techniques. We will refer to this layer as the flow control (FC) layer. To see how well a mechanism performed, we compared response time and throughput with and without that mechanism.

We developed a benchmark application, called *ring*. It works as follows. Each group member repeats the following routine for a specified number of rounds: multicast a specified number of messages of a specified size to other members of the group, and receive the same number of messages from each of the other members. Sending and receiving of messages in each round are concurrent to some degree. The degree of concurrency can be adjusted using the priority of the send and the receive threads. The key parameter in measuring response time and throughput is the number of messages sent per round. To measure response time, only one message need be sent using a lightly loaded system. To measure throughput, as many messages as practical should be sent to load the network to its maximum capacity. The message size and the number of members may be varied to simulate a system of interest.

To enable running many tests efficiently, we also developed another application called the *session service*. The session service takes a list of jobs to run on a set of machines, and runs each job on the least loaded subset of registered machines. The session service itself is distributed. Each individual session server runs on a separate machine, and is responsible for monitoring the load and running jobs on that machine.

We began our experiments by manually exploring various parameters. Once we obtained a feeling for what ranges of experimental parameters are interesting to observe (such as the window size of Section 3), we ran automatic experiments using the session service for those ranges of parameters in coarse increments. Finally, we designed specific experiments, with certain parameters fixed, while varying others in fine increments, and ran these. We repeated the final step several times to make sure that our results were reproduceable.

We ran all our tests on Sparc1s and Sparc IPCs that reside on our department Ethernet. (We chose these machines over faster machines available in our lab, because we have many of them.) We used Deering's UDP multicast [4] for multicasting messages. The machines we used were not dedicated to our experiment, but they were lightly loaded. Our curves are not smooth, because we did not omit any of the outliers. Since none of outliers correlated with experimental parameters across different runs, we think that these outliers are exclusively due to situational factors, such as unplanned load imposed by jobs run by other users. We ran our experiment over a period of nine months, during which we repeatedly redesigned our mechanisms and calibrated our system. All the data reported here were taken in the last few weeks of experimentation. For each of the data points in our results, we ran the ring application for a 100 rounds and used the average. We took all data points for a single curve in a single run.

3 Credit Window

Window-based flow control is a classic technique. In it, the system allows only a certain number of messages to stay unacknowledged between every sender–receiver pair at any given time. If more messages than the window size are unacknowledged by the receiver, then the system stops sending messages. The rationale behind window-based flow control is that the network has some maximum capacity of bandwidth and that the system needs to regulate the number of messages on the network. Once a machine has sent so many messages, it should not send any more until it knows that some of its messages have been delivered or have been lost with high probability. Detecting whether a message has been delivered can be implemented using acknowledgments. Message loss usually cannot be detected reliably, but is suspected when a certain time elapses without receiving an acknowledgment for a message. In real systems, network resources are shared and traffic on the network is unpredictable. Therefore, the window size need to be adjusted dynamically. Jacobson has come up with some good techniques for dynamically adjusting the window size in point-to-point communication [5].

While window-based flow control is wide-spread in traditional communication systems, its generalization to distributed systems has not been studied systematically. Distributed applications are characterized by a set of loosely-coupled communicating entities. The first step in generalizing the window-based flow control to a distributed environment is to think of these communicating entities as organized in sets of groups rather than as a large, unstructured collection of point-to-point connections. Whether to send messages to members of a group or to buffer them for later transmission, is decided based on the state of every member of the group.

There are two natural extensions of window-based flow control suitable for communication in a group. In the first variation, there is one window per sender. The window is adjusted when the last (slowest) receiver acknowledges a message from that sender. In other words, the system calculates the number of unacknowledged messages for each receiver, and takes the maximum. If this maximum exceeds the window size, the message is not sent to any of the members. We will refer to this first algorithm as the *local* algorithm, since each member only considers its own messages. Formally let s_i be the number of messages sent by process

i. Let r_{ij} be the number of acknowledgments that i received from process j. Then the local algorithm only sends messages if $\max_j(s_i - r_{ij})$ is smaller than the window size.

In the second variation of the window scheme, the sender decides whether to send a message based on how many messages are not known to be fully acknowledged. These messages include the messages it sent itself and the messages that it received from other members. We will refer to this algorithm as the *global* algorithm, since each member considers all messages in the system [1]. For each member in the group, a sender calculates pairwise for that member and each of other members the number of unacknowledged messages. It then takes the maximum for that member. The maximum values are added across all the members. When this sum exceeds the window size, the message is not sent. Formally the global algorithm uses $\sum_i \max_j(s_i - r_{ij})$ to compare with the window size when any member sends a message. The rationale behind the global algorithm is that if a process is slow in receiving messages from a particular sender, then all members should refrain from overloading that receiver. Determining the exact global state of a distributed systems is expensive and impractical. A practical approach, one that we used in our experiment, is to approximate the global state by piggybacking information on existing traffic, and only introducing new messages in the absence of application-level communication. The quality of the approximation depends on the time interval at which members inform each other of their states.

In our experiment we varied the size of the window and observed the performance. In the preliminary phase of our experiment, we also varied the frequency with which acknowledgments are sent as a function of the number of unacknowledged messages (*i.e.*, an acknowledgment is sent if the number of unacknowledged messages exceeds some threshold). We discovered that performance was not so sensitive to the variation in this frequency for the ranges we are examining, so we decided not to vary this parameter in our final experiments.

Note that the window size has a different meaning in the local and the global algorithms, and cannot be compared easily. While the local window is concerned with the unacknowledged messages between the sender and each of its receivers, the global window is concerned with the unacknowledged messages between all pairs of senders and receivers. Thus rather than comparing figures on a per-window basis, we should only be looking at the peak performance.

For these measurements, we used a group size of four and a message size of one byte. We turned off network-level flow control at the multicast layer, so that our measurements are not obfuscated by multiple levels of flow control. Unfortunately, this means that when network buffers are overrun, or packets get lost for other reasons, it is not always possible to recover a lost packet and the connection will fail. Thus, under certain circumstances, no communication was possible. This is not due to a malfunction in Horus, but due to an absence of low-level flow control.

Figure 1 shows how the two variations of the window mechanism behave. For a group size of four, the local algorithm obtained a peak value of about 230 messages per second, whereas the global algorithm only achieved 200 messages per second. Based on this and other runs, we believe that the local algorithm achieves better throughput overall, but do not rule out the possibility that there are cases where the global algorithm will perform better.

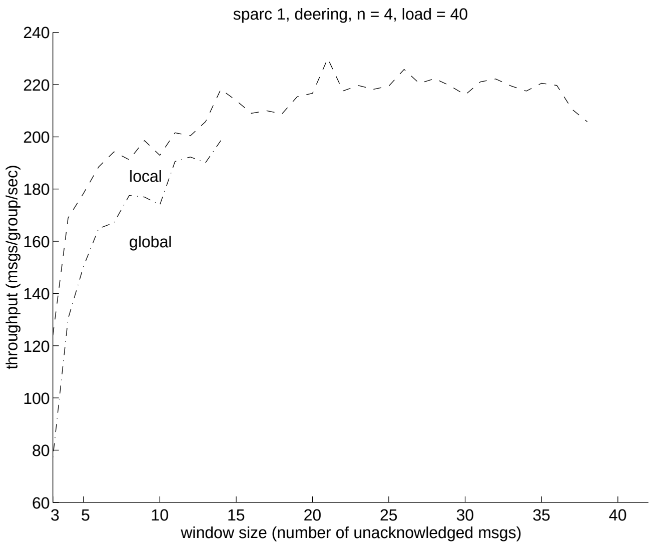

Figure 1: This graph displays the obtainable throughput over Horus using different credit windows. The top graph uses the *local* algorithm. The bottom graph uses the *global* algorithm. Because low-level flow control is disabled, messages may be dropped and communication becomes impossible if the window is chosen too large. As can be read from the graph, this happens at a window size of 40 for the local algorithm, whereas for the global algorithm this happens at a window size of 15.

This graph also shows that communication becomes impossible for a window size of 40, if the local algorithm is used, or 15, if the global algorithm is used. As mentioned before, window sizes for the two algorithms are incomparable. For the global algorithm, however, we observed that as the group size goes up, the window range for which communication is possible goes down. At a group size of eight, we were not able to push the load on the system higher than about 5 messages per round. This seems to say that the local algorithm is better than the global algorithm. This may be either inherent to the global algorithm itself or to the way we implemented the global algorithm.

For the local algorithm, we observe that the performance first improves as the window becomes larger, but then degrades beyond a certain optimal window size. From examining the network driver statistics, we find that the network driver drops messages, both at the sending and the receiving end, when too many messages are sent or received. Thus there is an optimal range for the window size. When the window size is too small, we underutilize the network. When the window size is too large, we congest the network. In the next two sections we present two techniques that deal with this issue.

4 Message Packing

Sending messages in current operating systems is expensive. The sheer act of sending requires processing and resources at both the sender and the receiver, and at all operating system levels. History shows that relative slowness of processing never seems to catch up with the speed of the network. Although the cost of processing has been steadily decreasing due to development of faster processors, invention of faster networks such as ATM has once again widened the gap between processing and networking speeds. On dedicated multi-processors, the senders and receivers are closely synchronized to avoid buffering all together, so as to achieve maximal performance. In a shared environment, even if the processing speed matches the network speed, some amount of buffering will always be needed, and the amount required is roughly proportional to the degree of concurrency in message sends.

Packing multiple messages into a single packet allows sharing of the cost of message passing among multiple messages, thus reducing the per message cost. If we combine message packing with the window schemes described in Section 3, systems can simply pack already buffered messages (done by the window scheme) rather than artificially delaying messages. This way there is little overhead in terms of response time due to message packing.

One caveat of message packing is that the technique is useful only if a significant number of messages are small. Network-level packets have a size limit. Once a message exceeds that limit, the network layer would have to fragment the message anyway. In an Ethernet (where our experiments are run) this size is about 1500 bytes. Fortunately, many real systems are characterized by predominantly small sized messages.

To measure the effectiveness of message packing, we compared performances of three types of Horus protocol stacks. The first stack, MBRSHIP, consists of protocols that deal with membership, fragmentation, and reassembly, but no (window-based) flow control or message packing. The second stack, TOTAL, consists of the MBRSHIP stack with an additional layer that provides total ordering to the messages. The TOTAL stack does not explicitly implement flow control, but uses the message packing technique for performance. The TOTAL protocol enforces that only the token holder can send a message. Although token was used to provide the semantics of the TOTAL protocol, we found that token also acts as an orthogonal form of flow control. The last stack, FC, consist of our flow control layer over the MBRSHIP stack.

Figure 2 shows performance of the MBRSHIP, the TOTAL, and the FC stacks. Our results show that message packing improves throughput dramatically. The basic MBRSHIP stack simply cannot take a large load without any flow control. The size of this load is a function of the buffer capacity in the system. In our system, the capacity is 16 buffers at the network layer. Thus, it makes sense that the MBRSHIP stack cannot take much more than a load of 16 messages per round (with network-level flow control enabled, the MBRSHIP layer can sustain a reasonable throughput.) As we impose a higher load, throughput increases for both the TOTAL stack and the FC stack. The FC stack can take a much higher load than the TOTAL stack, however. The peak throughput for the TOTAL stack is about 5000 messages per group per second at a load of 160 messages per member per round. When the load reaches 240 messages per round, the TOTAL stack starts to loose messages (appearing as a big dip in the curve) and throughput rapidly declines. In contrast, the peak throughput for the FC stack with the window scheme and message packing is 8000 messages per group per second at a load of 800 messages per round. (We will show that the FC stack can take an indefinite increase in load with the back pressure mechanism presented in the next section.) The TOTAL stack provides expensive functionalities, and these interfere with the

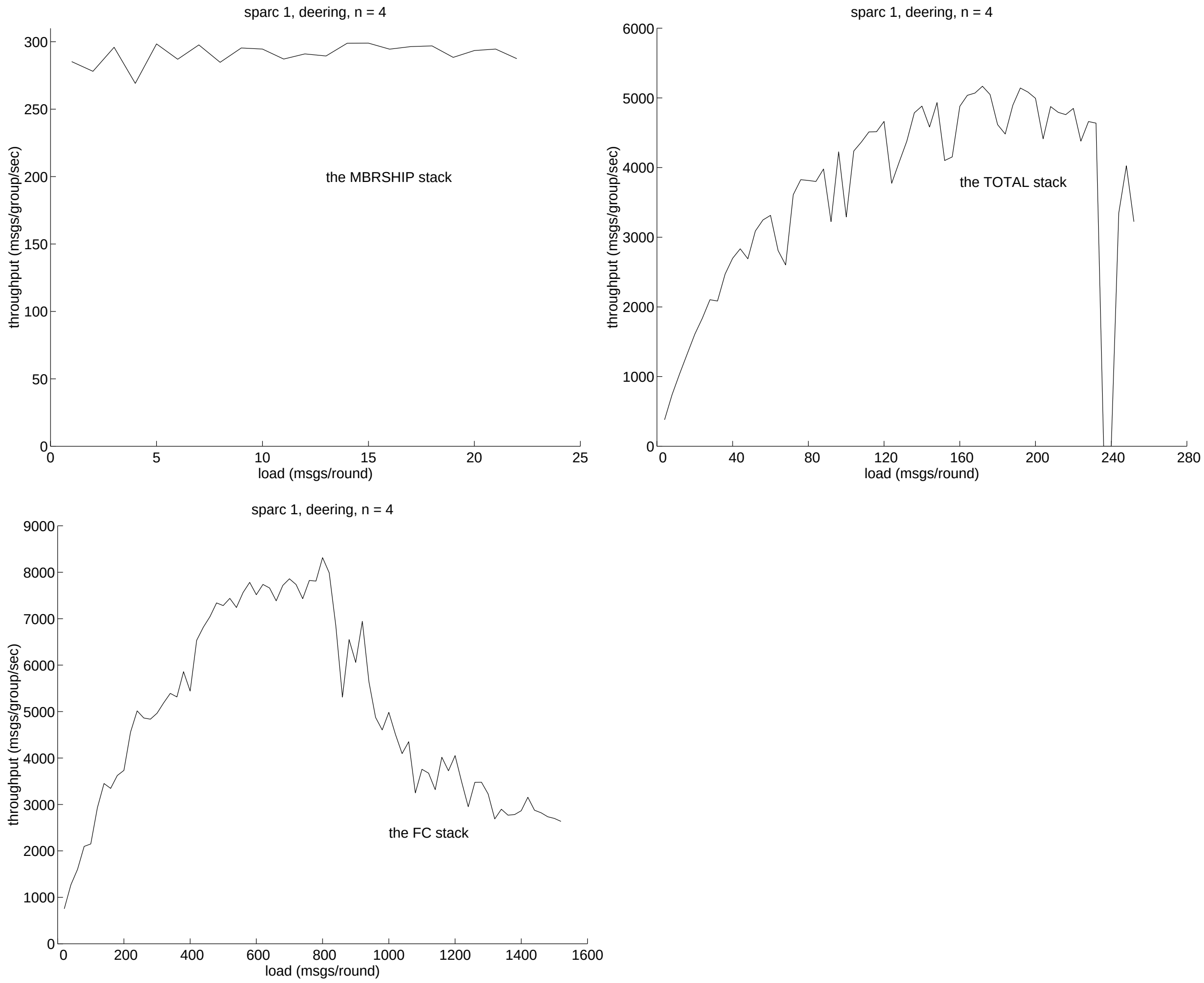

Figure 2: These graphs display the obtainable throughput over Horus with various stacks. The MBR stack uses no flow control. The TOTAL stack uses message packing. The FC stack uses message packing and window-based flow control.

effectiveness of the packing mechanism. This is an argument for why flow control mechanisms should be separated from other functionalities.

5 Back Pressure

A system has a limited amount of physical memory that must be shared using virtual memory pages. When a process working set exceeds a certain limit, the system starts thrashing as the system spends too much time serving page faults caused by accessing active pages that cannot be kept in main memory. Buffering messages takes up memory. When the system buffers too many messages, the system starts paging.

The system also has a practical limit on the number of threads. A thread takes up various system resources such as memory for a stack (typically on the order of 10 Kbytes), a slot

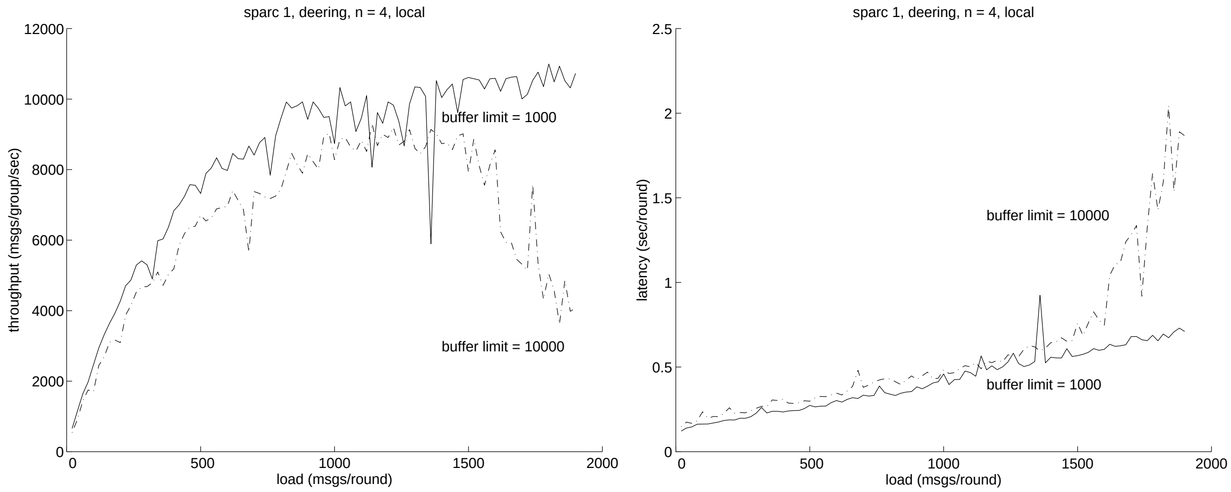

Figure 3: The obtainable throughput and latency for the FC stack with window-based flow control and message packing. Back pressure (buffer limit of 1000) maintains optimal throughput and low latency.

in system queues such as the run queue and wait queues, and a state descriptor containing information such as a program counter and a stack pointer. While systems such as Horus do not place a limit on the number of threads in principle, the system will start paging when too many threads are created. Also, if many threads are active, they will together consume a lot of CPU time.

One way to cope with this is to use a *back pressure* mechanism. When an application requests to send more messages than the system can currently handle, the system blocks the application until it completes handling of enough messages already in the system. This way the system allows ongoing communication to complete first before allowing more work, and this avoids paging. When to admit a message depends on the amount of resources that the system has available. If buffered messages use up most of the memory, or if unacknowledged messages are taking up threads close to what the system can handle, then the system should prevent the application from sending more messages.

Note that the window scheme and the back pressure scheme are related. While back pressure regulates the flow of messages from the application to the system, the window mechanism regulates the flow of messages from the system to the network and the other systems. Optimally, these flows should be matched.

We demonstrate the effectiveness of the back pressure mechanism by incorporating memory usage information into flow control. In particular, we impose a limit on the number of messages buffered for transmission. While the number of buffered messages is not the same as the amount of memory used by buffered messages, this is a reasonable approximation as long as all the messages are of approximately the same size. In our experiment, we kept the size of all the messages the same. Systems in which the size of messages vary greatly must keep track of the number of bytes used by buffered messages, in addition to the number of buffered messages, to use the back pressure mechanism effectively.

To test the back pressure technique, we varied the system load, and observed throughput with and without the back pressure mechanism in the FC layer. We fixed the group size to be four, the message size to be one byte, the window size to be two, and acknowledge each message separately. We only show our results for the local window algorithm. Figure 3 demonstrates that the back pressure technique prevents paging. Each curve in the figure is labeled with a buffer limit. This is the maximum number of messages that the system allowed to be buffered at any time. The back pressure mechanism showed its effectiveness with the smaller limit value of 1000, but not with the larger limit value of 10,000 (but it would at a higher load) for a group of size four. In both cases, throughput increases as we impose a higher load on the system. The throughput reaches a maximum value of about 10,000 messages per group per second at a load of 1000 messages per round. In both cases, throughput stays at the peak value until a load of 1500 messages per round is reached. However, without the back pressure mechanism a further increase in load causes a sharp decline in throughput. We stopped our measurement at a load of 2000 messages per round, when messages would start to get lost without the back pressure mechanism. With the back pressure mechanism, throughput stays at this peak value, and was maintained on any high load we tested. We carried out a systematic study to up to 3000 messages per round, and the throughput kept increasing (but at slower rate).

The degradation of performance above a load of 1500 messages per round without the back pressure mechanism is also reflected in a sharp increase in latency. The back pressure mechanism does not suffer this sharp increase in latency, but shows instead a gradual increase. The increase in latency with the back pressure mechanism is due to blocking messages, and is bounded if the system does not admit messages above its saturation point. This makes the performance of the system significantly more predictable.

Figure 4 shows that the back pressure technique helps sustain a high throughput for larger groups, but its effectiveness diminishes as the group size gets large and the load very high. We believe that for large groups reservation is needed to arbitrate sending within a group (see Section 7).

Picking the proper point to start blocking the sender is crucial. If the back pressure mechanism is too conservative, blocking the application too early and too often, it will result in an unnecessarily long response time. In our testing environment, this will happen with a buffer limit value much smaller than 1000. If the back pressure is too optimistic, however, the algorithm might fail to prevent the system from paging (as was the case for the buffer limit value of 10,000 that we described). The buffer limit is a function of the amount of memory the system has, and can be calculated for a given system.

6 Related Work

Flow control is a classic problem in networking. Many existing networks use window-based, point-to-point flow control [8]. For this, Jacobson developed heuristics to improve congestion control of TCP/IP communication [5]. The basic idea is to adjust the window size based on the current performance of the environment. The protocol first starts with a small window size and slowly increases the window size until the (estimated) network saturation point.

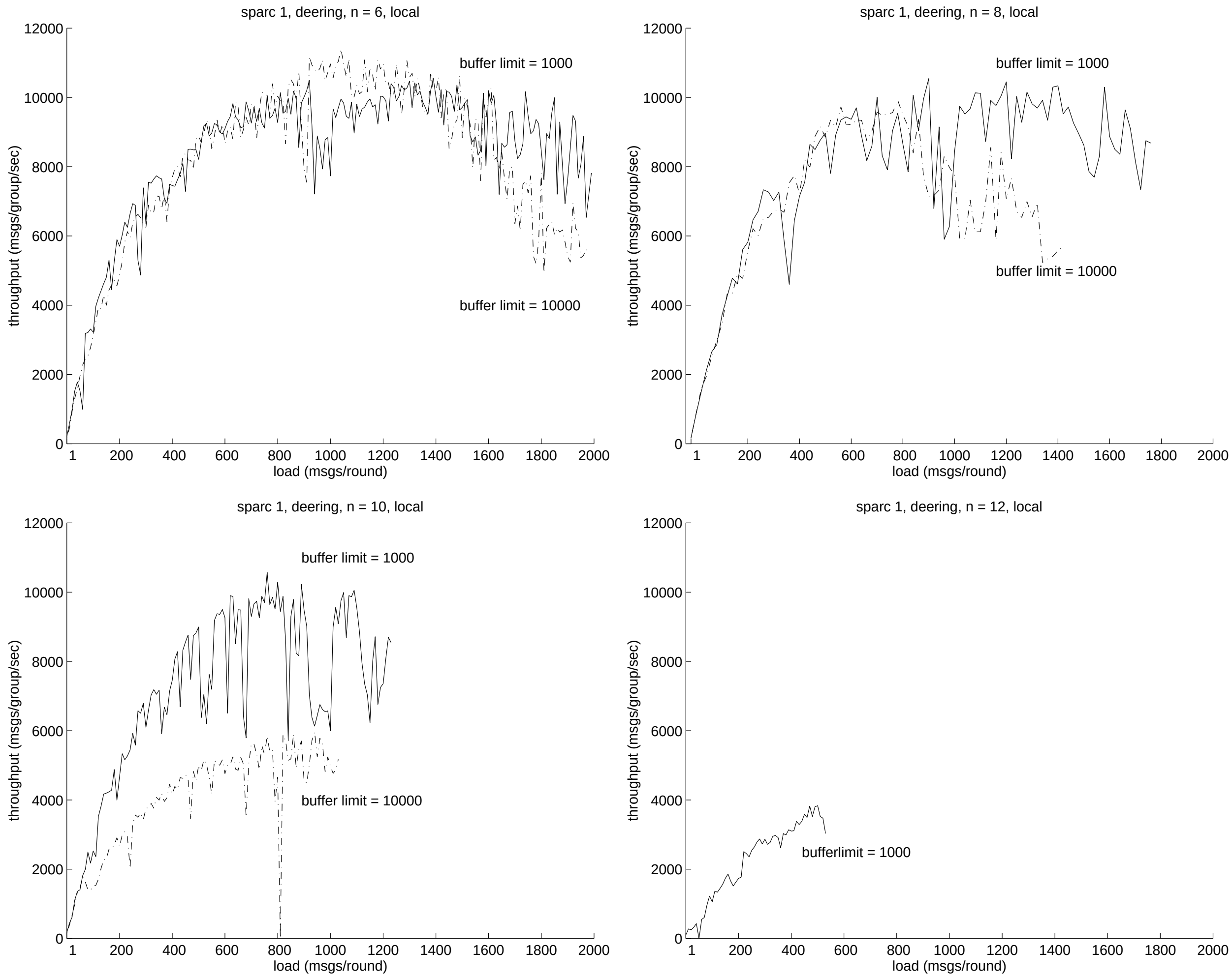

Figure 4: Throughput for larger groups. For large groups with high load, the mechanisms we used are still not sufficient.

When there is a congestion and messages get dropped, the window size is cut in half, and the protocol retransmits dropped messages at exponentially slower rates.

Extending the point-to-point window-based flow control to a window protocol suitable for multicast is not a new idea, and was done in Isis [3], Transis [1], and MUTS [9]. Our work is a systematic study of these extensions. While existing systems assume that global information is useful, we showed that this may not necessarily be the case. Our results indicate that an algorithm which uses only local information works for wider ranges of parameters and achieves slightly better performance.

The approach of reserving system resources for message passing has been used in real-time systems (*e.g.*, [2]). In such systems, applications must complete in a guaranteed amount of time. Realtime systems employ various techniques to meet these deadlines. Prior to committing to any request, the system compares the resource requirements of the request with the availability of system resources for the request. Only if the system has enough resources to execute the request in the required amount of time, it grants the request.

So-called “soft” real-time applications have desired time limits for tasks, but they can tolerate some tasks not meeting their deadline. Although the system makes some effort in meeting the desired time limits, it does not have to guarantee that it meets the deadlines for every task. This way, the system can be more optimistic in deciding whether to commit a resource to a task or not. Currently, our approach is more suited to these types of systems, which includes many multi-media systems. It uses system resource information as a hint to managing message traffic, but it does not know a priori the exact performance it will achieve by controlling the traffic.

7 Discussion and Future work

In this paper, we showed that incorporating system resources information into flow control is useful in upcall-based distributed systems. We started our investigation with variants of window-based flow control suitable for distributed systems. Our experiment suggests that approximate global information is not necessarily useful (and, in fact, can be harmful). Next, we showed message packing as a technique for increasing throughput. By passing many small messages in a single packet, we can amortize the cost of passing them and gain a significant increase in performance.

Finally, we showed that a back pressure mechanism which (indirectly) incorporates system resource information into flow control helps maintain the peak throughput on high load. The number of physical resources ultimately determines how much load a system can handle. Sharing of resources incurs extra overhead. By sending more messages than the receiver can handle, the ratio of overhead to useful work gets prohibitively high, and the system starts paging and stops doing any useful work. The back pressure mechanism prevents systems from entering this state by regulating the amount of resources consumed by the flow of messages. We also showed that the back pressure mechanism keeps the increase in the latency to a reasonable amount. We postulated that the response time is bounded if we reject rather than block messages above the saturation point. Thus, we can provide predictability by accounting high level resources.

The combination of the window scheme together with the back pressure mechanism is an end-to-end technique. The receiver stops acknowledging messages when it reaches its thread and/or memory capacity. This results in an increase of the number of unacknowledged messages. When the number of unacknowledged messages reaches the window limit, the sender stops sending. This in turn results in more buffered messages at the sender. When the number of buffered messages reaches some threshold, the back pressure mechanism kicks in and prevents the application from sending more messages.

There is still much work to be done in flow control in distributed systems. So far we only showed the usefulness of incorporating system resource information into flow control. This was done by varying the number of messages floating in the system (i.e., sent but not received by an application) and using different flow control techniques. We have not yet formalized how to pick suitable values for these techniques given a situation. We need to have the memory and thread modules communicate with the flow control module, so that the flow control module can use this information to adjust its parameters automatically.

We want to separate mechanism from policy. A separate policy module could pick flow control parameters based on various information sources. A policy module could also provide a priori reservation of resources. What we are proposing is essentially a scheduler for communication. Much of the work in distributed systems to date have been devoted to enabling communication among entities and providing transparency of distribution. Much less work has been done on efficient sharing in distributed systems. A communication scheduler could regulate the sharing of distributed systems by taking systems resources into account.

Acknowledgments

We are grateful to Fred Schneider and Ken Birman for discussions which have shaped this work. We thank Werner Vogels and Roy Friedman for their comments on an early draft of this paper.

References

[1] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication subsystem for high availability. In *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[2] David Anderson. Metascheduling for continuous media. *ACM Computing Surveys*, 11(3):226–252, August 1993.

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

[4] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[5] Van Jacobson. Congestion avoidance and control. In *Communications architectures and protocols*, pages 314–329, August 1988.

[6] Larry L. Peterson, Norm Hutchinson, Sean O'Malley, and Mark Abbott. RPC in the x-Kernel: evaluating new design techniques. In *Proceedings of the Twelveth ACM Symposium on Operating Systems Principles*, pages 91–101, Los Angeles, California, November 1989. ACM SIGOPS.

[7] Jerome Saltzer, David Reed, and David Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[8] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, second edition, 1988.

[9] Robbert van Renesse, Kenneth P. Birman, Robert Cooper, Brad Glade, and Patrick Stephenson. Reliable multicast between microkernels. In *Proceedings of the USENIX workshop on Micro-Kernels and Other Kernel Architectures*, Seattle, Washington, April 1992.

[10] Robbert van Renesse, Takako M. Hickey, and Kenneth P. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report TR 94-1442, Cornell University, August 1994.