A Multicast Flow Control Protocol

Mark Hayden Zhen Xiao*

April 6, 1999

Abstract

This paper describes the goals, approach, implementation, and performance of MFLOW, a multicast flow control protocol. Our work focuses on the issues that arise from the increase in acknowledgments that occur as the size of the group grows. MFLOW makes use of bulk acknowledgments to substantially reduce the number of acknowledgments per multicast message. It also staggers acknowledgments from different destinations to avoid storms of messages. We demonstrate that staggering can both decrease the number of acknowledgments needed for effective flow control as well as increase the robustness of the protocol to changes in the environment. We analyze both message and computation efficiency of MFLOW. The protocol is computationally efficient and performs well in our experiments.

1 Introduction

A classical problem with point-to-point communication in distributed systems is that of flow control, or controlling the sender's transmission so that it will not congest the network or overflow recipients' buffers. This may occur when the sender is running on a lightly loaded computer while the recipient is running on a heavily loaded one so that the recipient is unable to handle all the messages that arrive, and, consequently, some of them may be dropped. This paper describes an approach to flow control in multicast (1-to-many) communication settings where there is a similar problem, except that in this case there is more than one recipient. Multicast communication is widely used in group communication systems, where groups of processes that act together for increased reliability, reduced response time, or better availability.

Two common approaches to flow control in point-to-point settings are rate-based and sliding window flow control [Tan96] [Guo98]. The approach with rate-based flow control is to put a restriction on the rate of the sender to slow it down sufficiently so that it will not swamp the receiver. However, getting a suitable rate takes some tuning: if the rate is too conservative, it wastes network bandwidth. The approach with window-based flow control is to have the receiver provide feedback to the sender. At a certain point, the sender must wait

*This work was supported by DARPA/ONR contracts N0014-91-1-10014 and ARPA/RADC F30602-96-1-0317.

for such feedback before sending more messages, thus giving the receiver a chance to keep up. In such protocols, the sender and the receiver each maintain a flow control window. The sender's window determines the amount of data it is permitted to send, and the receiver's window determines the set of messages it is permitted to receive. The sender only advances its window when an acknowledgment from the receiver arrives, thus effectively preventing the sender from sending more messages than the receiver can buffer. Acknowledgments are often merged, thus a single *ack* packet can acknowledge many messages.

The direction we take for multicast flow control is to generalize the sliding window protocol from pairs of processes to groups. In doing so, a couple of issues arise from acknowledgments as the size of the group increases. In point-to-point communication, only one acknowledgment is needed for the sender to forward its window. In multicast setting with n processes, $n-1$ acknowledgments are needed. The number of acknowledgment messages grows linearly with the size of the group, which means the system does not scale well. Another issue is that if all receivers respond at the same time, the burst of acknowledgments from multicast destinations may potentially drown the sender's buffer. Not only may some of the acknowledgments be dropped, but also the sender is periodically disturbed by bursts of acknowledgments.

We present a multicast flow control protocol, MFLOW, which addresses these problems. MFLOW uses bulk acknowledgments to substantially reduce the number of acknowledgments. The idea is that, just as in point-to-point communication where the acknowledgments are often merged, the acknowledgments in multicast applications can also be merged. Our protocol also explicitly staggers acknowledgments from multicast destinations. Our focus is on how to use the information available in group communication system (where the sender knows the buffer size at the receivers') to develop deterministic and efficient algorithms. It does not use any randomization or timeouts. In many applications such deterministic behavior is desirable. And it is computationally much simpler, which enhances its performance.

Multicast flow control is inherently more difficult than point-to-point flow control because more processes need to acknowledge messages. As the group size increases, if all destinations periodically reply with acknowledgments at the same time, the sender and the network can be flooded with occasional storms of acknowledgments. To prevent this, our protocol staggers over time the acknowledgments from different multicast destinations. This prevents the sender from being periodically disturbed by bursts of acknowledgments. Bursty behavior is bad in its own right, and can also be a cause of buffer overflow.

What is more, experiments show staggered acknowledgments make the protocol open to a wider settings of parameters, which makes it more flexible and suggests a greater robustness to changes in the environment than the non-staggered protocols. This appears to occur because the staggered protocol spreads acknowledgments out over time instead of sending them in bursts. In many situations the congestion in the network or the malfunction of some network routers or links may affect multiple receivers. Thus the omission of an acknowledgment from one receiver may imply other receivers have similar problems due to the current network condition. If all receivers respond at the same time, the sender can only detect such conditions periodically (when it finds that several destinations do not send acknowledgments) and may not be able to take appropriate action in a timely manner. In contrast, in a staggered protocol, the acknowledgments from different receivers are staggered over time and the lack of any single acknowledgment will eventually slow down the sender. Hence the sender can be constantly "informed" of the current state of the network and adjust its actions correspondingly.

Our protocol is efficient both in terms of computation and messages. We use an optimized implementation which achieves amortized $O(1)$ processing time on each message send and receive. Our protocol remains inactive when no application in the group is sending messages. Thus it does not waste network bandwidth when the group is computing or doing other tasks which do not involve communication. When applications do communicate, we show, for a given system configuration, how to precompute the optimal parameters in order to minimize message traffic overhead induced by the protocol. Finally, we present the performance of our protocol.

Our protocol is implemented as a protocol layer in the Ensemble system[Hay98]. Ensemble is an extensively layered and highly reconfigurable group communication system developed at Cornell University. It provides a library of protocols which can be dynamically linked into protocol stacks in various ways to meet the needs of different applications. Different layers may be substitutable for one another and may have different behavior under different workloads, which facilitates the comparison of different flow control techniques. The underlying principles of our design, however, can be applied to other architectures as well.

2 Goals and Approach

The goal of our research is to implement a multicast flow control protocol which effectively addresses the issues that arise from acknowledgments as the group size increases.

The MFLOW protocol uses credits[KM] to measure the available buffer space at the receivers. Each sender maintains a *window* which is used to bound the number of unacknowledged multicast messages a process can send. Initially, each sender keeps some amount of credit in stock. For each message it sends, the process deducts a certain amount of credit based on the size of the message. Multicast messages are transmitted only if the sender has enough credit for every destination. Otherwise, messages are buffered without being sent.

Every receiver keeps track of the amount of unacknowledged data it has received from each sender. When the amount exceeds a pre-specified threshold, the *ack_thresh*, it sends an acknowledgment to that process. On receipt of acknowledgment messages, the sender recalculates the amount of send credit, and buffered messages are sent based on the new credit. The protocol attempts to avoid situations where all receivers send acknowledgments at the same time, so that a sender is not flooded with acknowledgment messages.

Our approach has the following characteristics described in the sections below.

2.1 Independent streams

Every sender in the group is handled separately. For each of the senders, our protocol manages the flow from the sender to its multicast destinations. In a group of n processes, the protocol should be viewed as n different instances. The data structures that are used for managing the flow control for the different instances are all distinct. Effectively, the layers at all the processes are implementing the same "one sender, $n-1$ receivers" protocol for each of the processes in the group.

This may turn out to be a mixed blessing, however. On the one hand, it facilitates the analysis of the protocol because they are all independent. When analyzing the protocol, we only need consider the case of one sender. On the other hand, it ignores other information global to the group. Such information may be useful to optimize the performance. For example, some processes in the group may send more messages than others. Thus it may be reasonable to assign more credit to them. However, this would complicate our design without necessarily providing significantly better performance.

2.2 Modular implementation

In some systems, flow control is implemented along with other protocol features. For instance, the TCP protocol implements flow control, reliable transmission, and failure detection. This approach has several problems associated with it. The flow control aspects of the protocol become intertwined with the others. This in turn makes it difficult to change how flow control is managed without dealing with other issues that arise from the other parts of the protocol.

The Ensemble system, of which MFLOW is a part, uses layered protocol structure in order break large monolithic protocols into small micro-protocols. Protocols can be composed in a large number of ways to achieve differing sets of properties needed by different applications. Our experience has been that when high-level properties such as reliable multicast, failure detection, and flow control are decomposed into smaller layers, each of which implements several simple properties, they can be developed and tested more easily than a large, monolithic one. It also allows each protocol to be used in combination with a variety of other protocols, and facilitates experimentation with new communication properties and incremental extension of the system.

Our protocol requires reliable, FIFO multicast and point-to-point properties from underlying protocol layers. The "acknowledgment" messages it uses are not for reliability purposes (the reliable multicast protocol has its own acknowledgments in addition). They are only used to inform the sender about how much free buffer space the recipient has at the moment.

2.3 Weighted flow control

In making flow control decisions, this protocol attaches a weight to each message according to the size of the application data portion of the message (plus a fixed amount to represent the additional overhead of Ensemble protocol headers). The sender deducts the amount of credit according to this weight. The receivers keep track of the total size of multicast messages received from each sender. An acknowledgment is sent when the amount exceeds the acknowledgment threshold. Since larger messages consume more buffer space at the receiver's side than smaller ones, it makes sense to deduct more credit for them.

2.4 Event driven

Unlike many other flow control protocols, this protocol does not use timeouts for flow control. In other words, all actions of the MFLOW protocol are activated as the result of message send and receive events that are passed to this layer. This gives the protocol a relatively "deterministic" behavior. Also, the MFLOW protocol is inactive when there are no application messages being sent. Thus it does not waste network bandwidth when the group is computing or doing some other tasks which do not involve communication.

3 Message Efficiency

The additional acknowledgment messages from the multiple destinations in a multicast setting creates the concern that the acknowledgments may swamp the sender and the network, so it is useful to formally analyze the protocol to ensure its efficiency. The measure of efficiency we use is calculated in terms of the number of additional (non-application generated) messages the protocol introduces. Additional messages entail processing and communication overhead, which in turn slows down protocol stacks and impedes achieving high performance. Here we present such a formal analysis of the overhead and show how the protocol parameters can be set to achieve a balance between message efficiency and buffer size.

We define message efficiency to be the fraction of the total number of messages that were generated by the application. Although the analysis we present is calculated based on rates (msgs/sec) of message traffic, if you counted over a period of time *nmsgs* application messages and *acks* acknowledgment messages (the only additional messages that MFLOW introduces), the efficiency under our definition would be:

$$msg\_eff = \frac{nmsgs}{acks + nmsgs} \tag{1}$$

For instance, a protocol with no acknowledgments has efficiency 1; a naive protocol in which all receivers acknowledge every multicast message, the efficiency is $1/(n-1)$ (where n is the number of processes in the group), which is quite inefficient.

Our definition of efficiency is somewhat conservative because acknowledgment messages are usually smaller than application messages and therefore consume less network bandwidth and processing time. Thus, with an alternative definition of efficiency that considered bytes of acknowledgments verses application data would normally give higher levels of efficiency than presented here.

In analyzing the protocol, we make some assumptions about the behavior of the application and the network. We assume that the application is multicasting fixed size messages at a fixed rate. In addition we assume that the network has a bound on the typical message propagation delay. This analysis necessarily abstracts away some of the details of the system. However, not all of these assumptions need be exactly met by the system for the analysis to still hold most of the time. For instance, if some messages occasionally take longer than the expected delay time, then this would cause the protocol to not behave as predicted for a short period of time surrounding the aberrations.

The key to efficiency is to amortize the overhead of acknowledgments across many application messages. The MFLOW protocol uses the *ack_thresh* parameter to set the number of bytes a destination will receive before replying with an acknowledgment. A large *ack_thresh* increases the effectiveness of the amortization. The trouble with message efficiency is that it involves a tradeoff with the size of message buffers. In particular, the buffer size of the destinations (set by the *window* parameter) must be set at least as large as *ack_thresh*. The more frequent acknowledgments are sent, the smaller the buffers that are needed to maintain a particular rate of traffic. The opposite is also true, flow control protocols can be made more efficient through less frequent acknowledgments but with larger buffers.

In analyzing the protocol, we make use of the following parameters:

- *size*: the size of application messages.
- *rate*: the sender's rate of multicasting messages.
- n: the number of processes in the group.
- ϵ: the typical propagation delay of messages plus the interrupt and service time.
- *ack_thresh*: the number of bytes after which a destination sends an acknowledgment.
- *window*: the number of unacknowledged bytes a sender will transmit.

With these definitions, the number of *acks* in our protocol can be calculated as follows:

$$acks = \frac{nmsgs \cdot size \cdot (n-1)}{ack\_thresh} \tag{2}$$

Substituting Equation 2 into Equation 1 leads to:

$$\begin{aligned} msg\_eff &= \frac{ack\_thresh}{ack\_thresh + size \cdot (n-1)} && (3) \\ &= 1 - \frac{size \cdot (n-1)}{ack\_thresh + size \cdot (n-1)} && (4) \end{aligned}$$

To avoid introducing delays, the *ack_thresh* should be set so that the acknowledgments from multicast destinations can be expected to get back before the sender's window fills up, as shown in Figure 1 (we only depict one of the receivers).

After sending the message which exceeds the *ack_thresh*, the sender may continue for a time:

$$\frac{window - ack\_thresh}{size \cdot rate}$$

at which point it must stop if the acknowledgments have not come back yet[1]. Hence, in order for the acknowledgments to come back before they are needed, the *ack_thresh* should satisfy:

$$\frac{window - ack\_thresh}{size \cdot rate} \geq 2\epsilon \tag{5}$$

[1] The discussion here is simplified in that, after sending the message which exceeds the *ack_thresh*, the sender's remaining credits are a little less than $window - ack\_thresh$, depending on the size of that message.

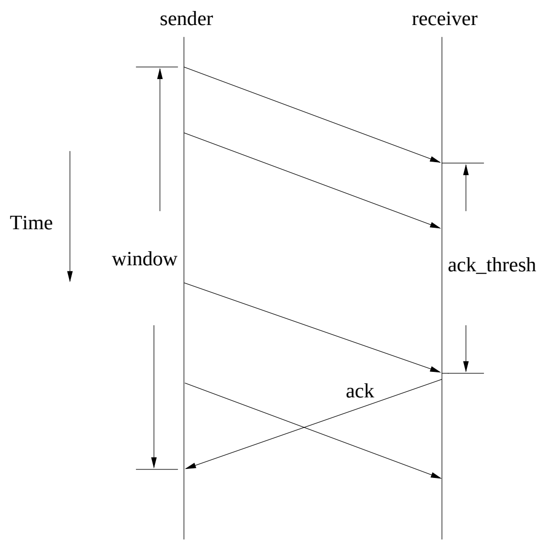


Figure 1: *Maximum value of ack_thresh without introducing unnecessary latency. The ack_thresh should be set so that the acknowledgments from multicast destinations are supposed to get back before the sender's window fills up. In the absence of message loss on the network or processing delay, this will prevent the MFLOW protocol from unnecessarily delaying messages.*

Solving this inequality yields:

$$ack\_thresh \leq window - 2\epsilon \cdot size \cdot rate \tag{6}$$

Combining Equation 4 and 6, we have:

$$msg\_eff \leq 1 - \frac{size \cdot (n-1)}{window - 2\epsilon \cdot size \cdot rate + size \cdot (n-1)} \tag{7}$$

This is the maximum message efficiency we can get without introducing unnecessary latency. Figure 2 (a) depicts the growth of efficiency when the window size increases for a group of 10 processes with $size = 1K$ bytes, $\epsilon = 2$ milliseconds, and $rate = 1000$ msg/sec.

Also, from Equation 3 we can see that our protocol is more efficient for smaller messages than for larger ones.

4 Avoiding storms of acknowledgments

As mentioned previously, a concern in multicast flow control protocols is that storms of acknowledgments may overflow a sender's buffer if all receivers acknowledge at the same time. In addition to overflowing the buffer, the sender will be periodically perturbed by receiving all acknowledgments at once.

Figure 3 (a) depicts such a protocol in a group of 4 processes with $ack\_thresh = 15K$ bytes and $window = 30K$ bytes. One process in the group keeps multicasting messages at

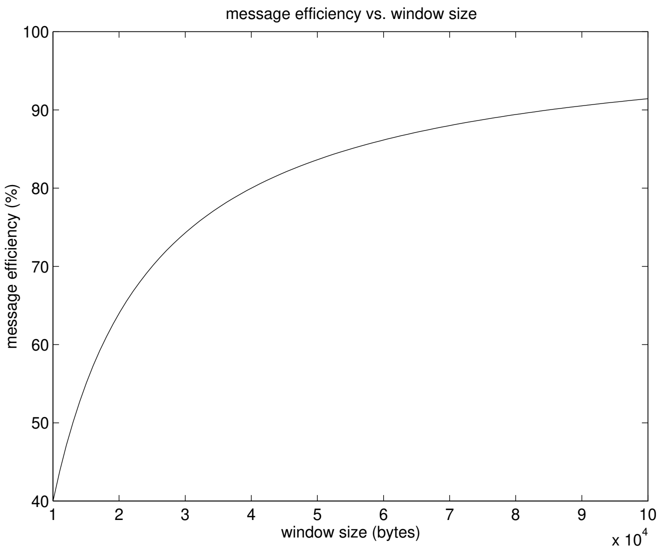


Figure 2: *The maximum message efficiency versus window size for a group of* 10 *processes with size* $= 1K$ *bytes,* $\epsilon = 2$ *millisecond, and rate* $= 1000$ *msg/sec.*

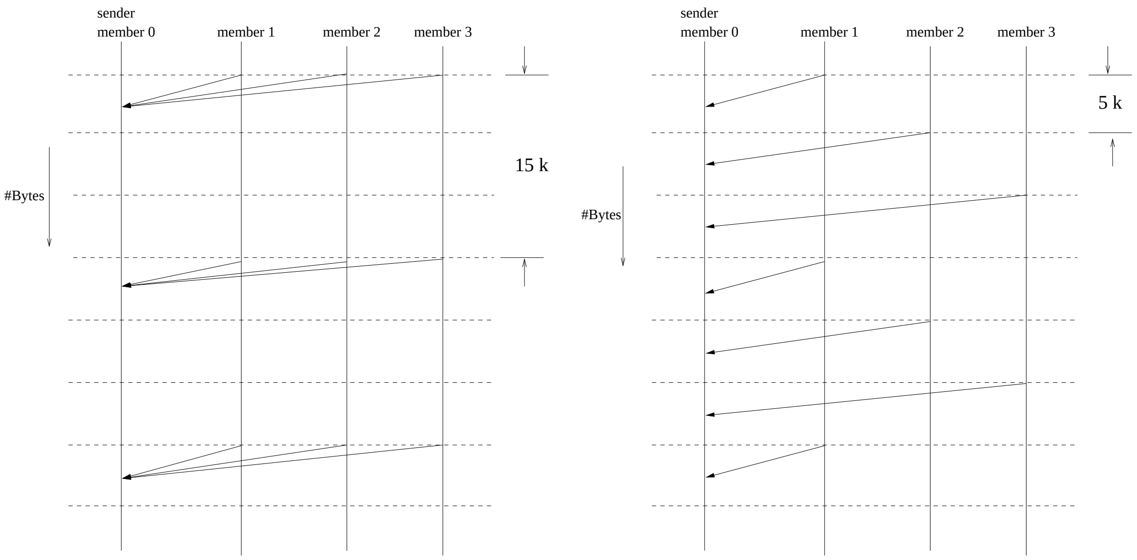
a fixed rate. The horizontal direction represents different processes in the group, and the vertical direction represents the number of bytes of multicast messages received so far. The arrow denotes the acknowledgment messages. As can be seen from the figure, there is a storm of acknowledgments every $15K$ bytes.

Ideally, the acknowledgments from multicast destinations should be distributed over time. Not only does this make buffer overflow less likely, but also prevents the sender from being periodically disturbed by bursts of acknowledgments. Figure 3 (b) depicts such a protocol, where any two adjacent acknowledgments are $5K$ bytes apart.

There are other advantages to staggering the acknowledgments. In many situations, the congestion in the network or the malfunction of some network routers or links may affect multiple receivers. Thus the omission of an acknowledgment from one receiver may imply other receivers have similar problems due to the current network condition. However, if all receivers respond at the same time, the sender can only detect such conditions periodically (when it finds several destinations do not send acknowledgments). Thus it may not be able to take appropriate actions in a timely manner. In contrast, in our algorithm the acknowledgments of different receivers are spread over time and the lack of any acknowledgment can slow down the sender. Hence the sender can be constantly informed of the current state of the network and adjust its action correspondingly. We empirically demonstrate this effect in our tests.

4.1 Staggering Algorithm

Our protocol staggers acknowledgments by initializing the values of received credits, the *recv_credit* array, for different processes in the group to some distinct negative values so that different receivers reach the acknowledgment threshold at different times. In the above



(a) Without Staggering (b) Staggering Acknowledgments

Figure 3: *The effect of staggering. (a) Without staggering, all receivers reach their acknowledgment threshold at the same time. The storm of acknowledgments will disturb the sender and may overflow the sender's buffer. (b) With staggering, the acknowledgments from different receivers are interleaved.*

example, the *recv_credit* arrays for different processes are initialized as shown in Table 1 (a).

As can be seen from this table, any two adjacent acknowledgments from different receivers are staggered $ack\_thresh/(n-1)$ apart. For example, in the first column of the array, process 1, 2, and 3 are the receivers, and their initial credits are $0K$, $-5K$, and $-10K$, respectively, which are $5K$ apart.

However, to stagger acknowledgments it is not sufficient just to set the initial credits for different receivers to distinct values. We also need ensure that their credits do not synchronize after some rounds of communication. In order to guarantee that the amount of credit in stock for different receivers are always $ack\_thresh/(n-1)$ apart, when a receiver acknowledges previous credits, it does not simply send back all of them. Rather, the amount

rank	0	1	2	3
process 0	NA	0 k	0 k	0 k
process 1	0 k	NA	-5k	-5k
process 2	-5k	-5k	NA	-10k
process 3	-10k	-10k	-10k	NA

(a) *recv_credit* array

rank	0	1	2	3
process 0	NA	30k	35k	40k
process 1	30k	NA	35k	40k
process 2	30k	35k	NA	40k
process 3	30k	35k	40k	NA

(b) *send_credit* array

Table 1: *The credit arrays for a group of* 4 *processes with* $ack\_thresh = 15K$ *bytes and* $window = 30K$ *bytes.*

sent back is always multiples of *ack_thresh*. Thus the remaining credits for different receivers are distinct.

This raises another issue, however. Since receivers do not send back all the credits they received, at a certain point the sender will stop sending messages due to lack of credits, even though actually there is enough buffer space at the receivers' side. To avoid the situation where the sender stops sending messages prematurely, we adjust the amount of the sender's initial credits for different receivers to compensate their initial negative values. For the above example, the sender's credit array would be initialized as shown in Table 1 (b).

5 Implementation Issues

A naive implementation of the MFLOW protocol involves operations with $O(n)$ (where n is the number of processes in the group) processing time on most message sends and receives. Most other protocols in the Ensemble system have constant processing time for normal case message sends and receives, thus making the MFLOW protocol a potential bottleneck of the system as the group size increases. Hence an important consideration in our implementation is how to achieve constant processing time, which we describe here.

All the operations for the *recv_credit* array can be easily implemented in constant time. For the *send_credit* array, however, the following two operations are on the critical path :

- *take*: Reduce the amount of available credit for each receiver when the sender sends a message.

- *check*: Check whether enough credits are available for all receivers. If so, buffered messages are sent as allowed by the amount of credit.

A straightforward way to implement the above two operations is to use a *send_credit* array as described in previous sections. Whenever the sender sends a message, it checks each entry in the array to see whether it has enough credits and then deducts a certain amount of credit from each entry in the array. This simple implementation has complexity $O(n)$.

We can improve the above approach by introducing the following two variables :

- *msg_len*: The total length of messages the sender has multicasted so far.

- *waitfor*: The rank of the process with the fewest amount of credit.

When the sender sends a message, instead of deducting a certain amount of credit from each entry of the *send_credit* array, it adds that amount into *msg_len*. For the *check* operation, it just compares *send_credit*[*waitfor*] with *msg_len* and sends messages if the former is greater. The essence of this approach is to maintain the invariant that *waitfor* always points to the process with the fewest amount of credit. This can be implemented using a heap or a priority queue with complexity $O(\log n)$.

However, we can do better than that. There are two properties in our protocol:

- The initial values in the *send_credit* array satisfy:

$$send\_credit[i] < send\_credit[i+1] \text{ for } i = 0, \ldots, n-2$$

- The amount of acknowledgment credit is equal to *ack_thresh*, which satisfies:

$$ack\_thresh \; > send\_credit[n-1] - send\_credit[0]$$

With these two properties, the process with the least amount of credit must be the one with the lowest rank among those who have received the fewest number of acknowledgment messages. Thus we use the following algorithm:

> Associate an acknowledgment queue with each entry in the *send_credit* array. Upon receipt of an acknowledgment from a process for whom we are not currently waiting, we put the acknowledgment into the corresponding queue for later processing — the *send_credit* array is not changed at this moment. When the sender receives an acknowledgment from the process for whom it is waiting, it adds the amount of credit into the *send_credit* array and forwards *waitfor* to the next entry in the array, wrapping to the front if necessary. If the acknowledgment queue of that process is not empty, remove an acknowledgment from the queue and increase the corresponding entry in the array. Repeat the above until we finally reach a process with an empty acknowledgment queue. Then that process must be the one with the least amount of credit.

The above algorithm achieves an amortized complexity of $O(1)$ because each acknowledgment is processed exactly once and the amount of processing time per acknowledgment is constant.

6 Testing

We present three sets of experiments with the MFLOW protocol. The first demonstrates that staggering of acknowledgments greatly increases the inter-arrival times of acknowledgment messages at senders. The second test demonstrates that we can use a larger window size and hence use fewer acknowledgment messages by staggering the acknowledgments. The third test demonstrates the bandwidth achievable with the MFLOW protocol for a range of group sizes. We compare performance with that of the TCP protocol with 2 processes (which can be viewed as a ceiling for the available bandwidth of the system) and show that MFLOW is only slightly slower than TCP for 2 processes and then decreases slowly as more processes are added.

All tests involve an application program in which one process in the group multicasts messages of a particular size at a specified rate for some period of time. The application program introduces minimal processing overhead and the networks are otherwise lightly loaded, thus the tests demonstrate MFLOW's behavior in a controlled setting to allow us to draw meaningful comparisons. The tests do not demonstrate MFLOW's behavior in heavily loaded scenarios.

The tests we present are on 3 different platforms[2]. The first set of tests are on an 8 node IBM SP2 supercomputer, the second set are on 4 Sparc 20 workstations, and the third set are on a network of PCs running Windows NT.

6.1 Staggering

6.1.1 Interleaving acknowledgments

We tested our staggering algorithm on an 8 processor IBM SP2 machine running the AIX operating system over a 10 Mbit Ethernet. Our test program has only one process in the group multicasting messages of a specified size at a specified rate for some period of time. Our test measures how closely acknowledgments arrive at the multicast sender. We calculate the interval time between the $i-1$ and ith acknowledgment, and that between the ith and $i+1$ one. Then we choose the minimum of the two as the ith value in the interval times. We compare the results with and without staggering.

Figure 4 presents the results for a group of 8 processes with $1K$ byte messages. The sender's rate is 100 msg/sec. The *window* and *ack_thresh* of MFLOW are set to be $50K$ and $25K$ bytes, respectively. The x-axis is the interval time calculated as above in milliseconds; the y-axis is the fraction of intervals which are less than the corresponding amount of time. Note the scales in the x-axis of the two figures are different.

As can be seen in Figure 4, without staggering, above 93% of the interval times are less than 0.2 milliseconds. In contrast, with staggering all the interval times are more than 10 milliseconds. We conclude that our staggering algorithm avoids the storm of acknowledgments from multiple destinations and thus enhances the scalability of the system.

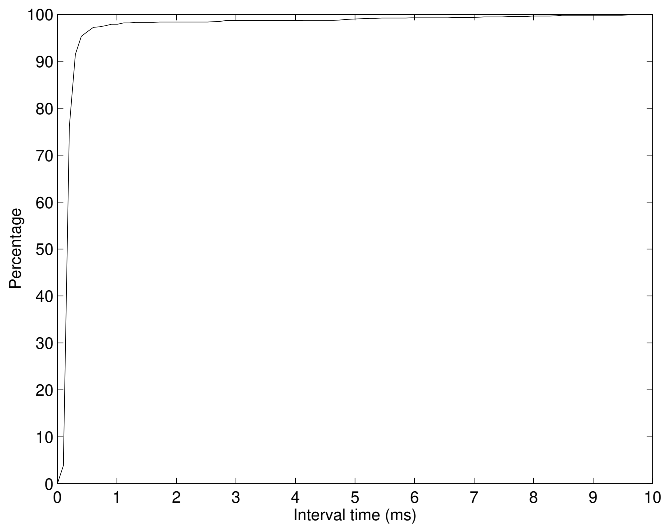
6.1.2 Increasing window size

We tested the effect of staggering on the growth of the sender's window on a group of 4 Sparc 20s running SunOS and connected by 10Mbit Ethernet. Our test program consists of one member in the group multicasting messages of a specified size of a specified rate for some period of time. The sender's rate is always faster than what the network can transmit. We vary the acknowledgment threshold and the sender's window size and measure how the bandwidth changes when the window increases. We compares the results with and without staggering. Figure 5 presents the results for a group of 4 members with $7K$ byte messages[3]. The x-axis is the sender's window size, and the y-axis is the bandwidth calculated as the average number of messages got through per second. The underlying network provides hardware support for multicast groups. The acknowledgment threshold is always $10K$ bytes smaller than the sender's window.

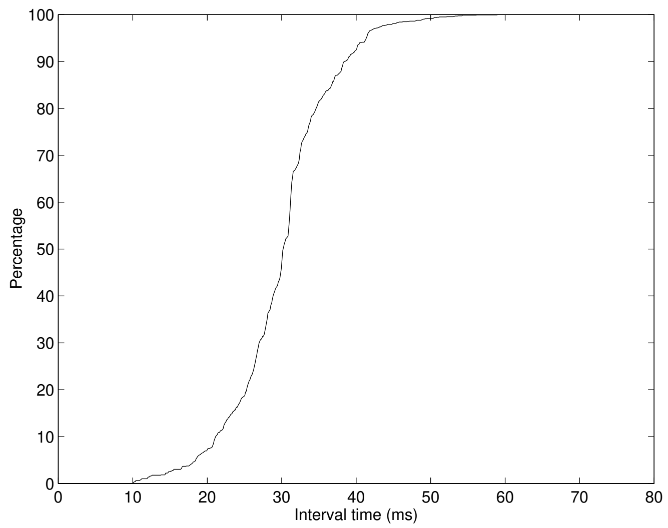
As can be seen from this figure, for $40K$ windows, both protocols (with and without staggering) have more or less the same performance. As the window size increases, the

[2]This came about because the tests were taken over a period of several months and during this period the configurations of computers available to us changed several times. While the use of different platforms does not affect the conclusions we draw, for consistency we intend to run the tests again on a single platform

[3]Messages larger than $7K$ bytes are fragmented by Ensemble and we wish to keep the effects of fragmentation out of our measurements.



(a) Without Staggering



(b) With Staggering

Figure 4: *The effect of staggering for a group of* 8 *processes with window* $= 50K$*bytes, ack_thresh* $= 25K$*bytes, size* $= 1K$*, rate* $=$ 100 *msg/sec. The x-axis is the interval time in milliseconds; the y-axis is the fraction of intervals which are less than the corresponding amount of time. Note that the scale in the x-axis of the two figures are different. As shown in the figure, without staggering, above* 93% *of the interval times are less than* 0.2 *milliseconds. In contrast, under our staggering algorithm, all the interval times are more than* 10 *milliseconds.*

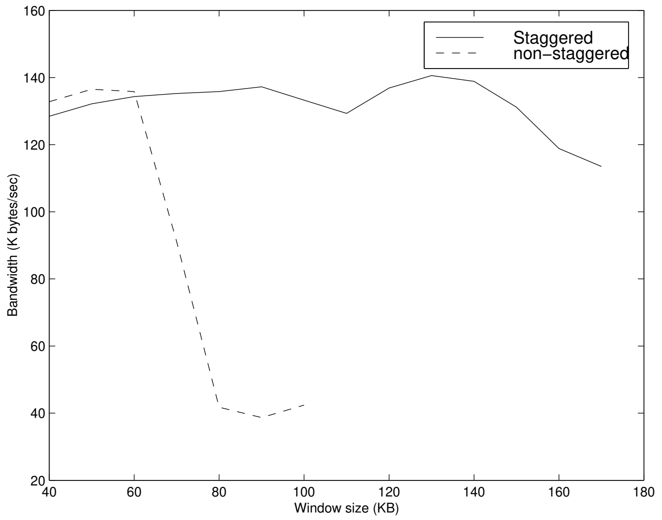


Figure 5: *The effect of staggering on bandwidth when the sender's window increases for a group of* 4 *members with* $7K$ *byte messages. The x-axis is the sender's window size; the y-axis is the bandwidth measured as the average bytes got through to the receiver's side per second.*

performance of the non-staggering protocol degrades rapidly, while the staggering protocol maintains its rate. In particular, when the window size increases from $40K$ to $100K$ bytes, the bandwidth of the non-staggering protocol drops from $929K$ bytes/sec to $296K$ bytes/sec. On the other hand, the staggering protocol maintains a steady bandwidth of about $918K$ bytes/sec when the window size grows from $40K$ to $170K$ bytes. This demonstrates that with staggering we can use a larger window and hence use fewer acknowledgment messages while maintaining the same bandwidth.

Also note that in order to maintain a comparable bandwidth, the maximum window size for the non-staggering protocol is $60K$ bytes, which is about 1/3 of the maximum size for the staggering protocol. There are 3 receivers in this case, and this may imply that with staggering, the window can be almost n times the window for the non-staggering protocol when there are n receivers in the group[4].

Using large windows can have significant influence on improving performance. Thus it is desirable to transmit more messages before we have to stop to wait for acknowledgments to come back.

6.2 Performance

The final test is done on a group of 200 MHZ Pentium PC machines running Windows NT and connected by a $10M$ bit Ethernet. Our test program consists of one sender in the group multicasting messages of a specified size of a specified rate for some period of time. The sender's rate is always faster than what the network can transmit. We measure how the bandwidth of MFLOW changes for different group sizes.

Figure 6 presents the results for groups of 2, 3, ... 9 processes with $7K$ byte messages. The underlying network provides hardware support for multicast, so Ensemble uses IP multicast when there are more than 2 processes in the group. The x-axis is the size of the group, and the y-axis is the bandwidth calculated as the average bytes delivered per second. The *window* and *ack_thresh* for MFLOW are set to be $50K$ and $25K$, respectively.

As can be seen from this graph, MFLOW achieves a bandwidth of $1028K$ bytes/sec for a group of 2 processes. When the group size increases, the bandwidth drops slowly. In particular, it maintains a bandwidth of $745K$ bytes/sec for a group of 9 processes. In other words, the bandwidth only drops 27.5% when the group size increases from 2 to 9, which demonstrates its good performance.

We also measured the bandwidth of TCP in the same environment for groups of 2, 3 and 4 processes. For a group of 2 processes, TCP achieves a bandwidth of $1071K$ bytes/sec, which is 4.18% higher than that of MFLOW. We believe that the superior performance of TCP is due to the following causes:

- TCP is implemented inside the kernel, while MFLOW is outside the kernel. Consequently, TCP can perform some operations more effectively than MFLOW and has more control over buffering operations. In particular, MFLOW is subject to buffer overflow in the kernel, whereas TCP implementations can avoid this.

[4]We have not yet verified this claim in a wider range of tests due to lack of suitable environments.

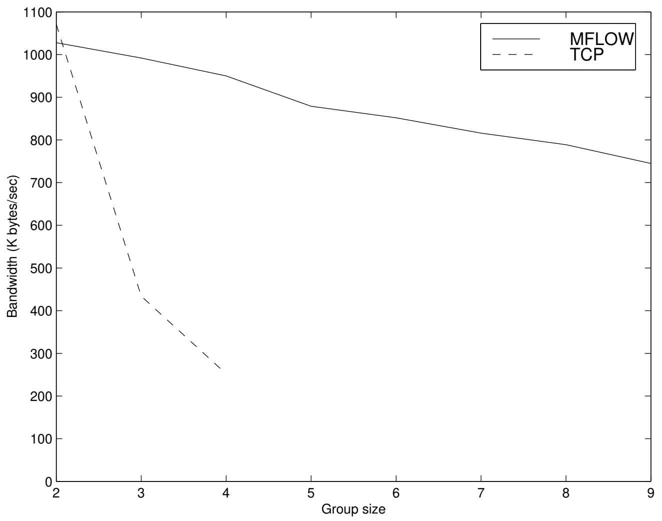


Figure 6: *Changes of the bandwidth for TCP and MFLOW when the group size increases. The size of messages used in this test is* $7K$ *bytes. The x-axis is the size of the group, from* 2 *through* 9*. The y-axis is the bandwidth calculated as the average bytes got through to the receiver's side per second.*

- MFLOW is implemented as a protocol layer in Ensemble. As a toolkit for building complex distributed applications, Ensemble normally has 16 to 20 protocol layers in the protocol stacks. For every application message sent, each layer adds its header, which is processed and popped off by the corresponding layer at the receiver's side. Although some optimization methods are used to compress protocol headers, this overhead can be substantial.

When the group size increases, the bandwidth of TCP drops significantly. In particular, the bandwidth is $434K$ bytes/sec for a group of 3 processes and $252K$ bytes/sec for a 4 process group. In other words, the bandwidth drops 76.5% when the group size increases from 2 to 4. This is not surprising because TCP does not use IP multicast and thus the messages are actually sent as unicasts through the network. We stopped at 4 processes since the trend is already obvious.

7 Conclusion

In this paper, we presented a multicast flow control protocol, its formal analysis, its implementation issues, and performance tests demonstrating its effectiveness. In multicast settings, a message is sent simultaneously to multiple destinations and the acknowledgments from these destinations impose overhead that grows with the size of the group. The effective handling of the acknowledgment messages is essential to multicast flow control algorithms. We have shown that the idea of bulk acknowledgments can be extended into multicast setting to reduce protocol overhead. In addition, staggering of acknowledgments can be used to effectively avoid a flurry of acknowledgments that otherwise would occur and produce more stable performance.

Acknowledgments

We are grateful to Robbert van Renesse, Srinivasan Keshav, and Ken Birman for discussions which have shaped this work. Mihai Budiu and Takako Hickey also made useful suggestions.

References

[Guo98] Katherine Hua Guo. *Scalable Message Stability Detection Protocols.* PhD thesis, Cornell University, May 1998.

[Hay98] Mark Hayden. *The Ensemble System.* PhD thesis, Cornell University, January 1998.

[KM] H.T. Kung and R. Morris. Credit-based Flow Control for ATM Networks.

[Tan96] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, 1996.

A Pseudo Code

A.1 MCREDIT Module

The MCREDIT modules manipulates the credit data structure.

[**got_credit t rank credit**] receive [credit] credit from process [rank].

[**got_msg t rank int**] receive a message of length [int] from process [rank]. It returns the total amount of credit received from that process so far.

[**set_credit t rank int**] reset the *recv_credit* array after having acknowledged previous credits from process [rank].

[**check t**] check if enough credits are available for everybody.

[**take t credit**] remove [credit] credits from all processes.

A.2 MFLOW Module

The MFLOW module decides the policy of flow control and handles event passing in the protocol stack.

State variables :

[**window**] size of the window

[**ack_thresh**] acknowledgment threshold

[**send_buf**] message buffer. Application messages are buffered here if not enough credits are available for everybody.

[**credit**] credit information, which includes *send_credit* array, *recv_credit* array, etc.

Event handlers :

[**dn event_type msg_hdr**] pass an event down of the given type with the given message header.

[**up event_type**] pass an event up of the given type.

```
upon recv ev msg hdr do
(* Increase the amount of credit to pass back to the sender.
 * If the amount of credit is beyond the threshold,
 * send an acknowledgment.  Finally, deliver the message.
 *)
  | ECast, Data ->
      let len = msg_len msg in
      let current_credit = Mcredit.got_msg credit origin len in
      if current_credit > ack_thresh then (
        let remainder = current_credit mod ack_thresh in
        let credit = current_credit - remainder in
        dn (sendToRank origin) (Ack(credit)) ;
        Mcredit.set_credit credit origin remainder
      ) ;
      up ev msg

(* Some credit are sent back.  If messages are waiting, check whether
 * enough credits are now available.  If so, buffered messages are
 * sent as allowed by the amount of credits.
 *)
  | ESend, Ack(credit) ->
      Mcredit.got_credit credit origin credit ;
      while (not Queue.empty send_buf)
         && (Mcredit.check credit)
      do
        let ev,msg = Queue.take send_buf in
        Mcredit.take credit (msg_len msg) ;
        dn ev msg Data   (* send down *)
      done

upon xmit msg do
(* Multicast a message to each destination if enough credit is
 * available.  Otherwise, the message is buffered.
 *)
  | ECast ->
      if Mcredit.check credit then (
        Mcredit.take credit (msg_len msg) ;
        dn msg Data
      ) else (
        Queue.add (ev,msg) send_buf
      )
```