Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols*

Roy Friedman Robbert van Renesse
Department of Computer Science
Cornell University
Ithaca, NY 14853.

July 7, 1995

Abstract

This paper compares the throughput and latency of four protocols that provide total ordering. Two of these protocols are measured with and without message packing. We used a technique that buffers application messages for a short period of time before sending them, so more messages are packed together. The main conclusion of this comparison is that message packing influences the performance of total ordering protocols under high load overwhelmingly more than any other optimization that was checked in this paper, both in terms of throughput and latency. This improved performance is attributed to the fact that packing messages reduces the header overhead for messages, the contention on the network, and the load on the receiving CPUs.

*This work was supported by ARPA/ONR grant N00014-92-J-1866

1 Introduction

One of the main sources of difficulty and complexity in the design of asynchronous distributed systems is the fact that messages may be delivered in different order at different process. Consequently, distributed protocols that assume a service which delivers all messages in the same order to all processes are much more simple and elegant than protocols that do not make this assumption [3, 4, 5, 18, 19]. However, providing total ordering of messages in asynchronous environments usually incurs a considerable cost. As a result, developers of distributed systems tend to avoid total ordering protocols and resort to more complex solutions. The latter approach has two drawbacks: (a) it increases the development time of the entire system, and (b) due to the complexity of the system, it is much harder to verify its correctness.

Recent studies suggest that if the performance of a total ordering service is not too far from the performance of a network that does not provide this guarantee, then a system which uses the total ordering service may actually achieve better performance than a system which was built on top of the unordered network [18]. The reason for the improved over all performance of systems that use total ordering protocols is that these systems do not need to implement locks for conflicting actions. For example, by using total ordering protocols, doing transactions in a serializable manner is simple and fast. On the other hand, the common approach to serializable transactions in an unordered environment is to use a 2PL protocol, which is typically more expensive than a total ordering protocol.

In this paper, we investigate the throughput and latency of four protocols for providing total ordering. Each of the protocols that we check uses a different strategy to order messages, and uses different optimizations. For two of these protocols we have added a message packing option, and measured their performance with and without this optimization. This included in one case buffering messages for a short period of time, slightly less than their minimal expected delay, then packing all the messages that have been buffered and sending them as one packed message. It turned out that packing messages improves both the latency and throughput of the protocols by two order of magnitudes, and is therefore overwhelmingly more important for the performance than any other optimization that we used. Moreover, protocols that pack messages were able to sustain much higher loads than protocols that did not pack messages.

We believe that this improved performance of the packing protocols can be attributed to the following three reasons, which are similar to the ones that make public transportation systems more efficient than using private cars: Packing messages reduces the total number of bytes being sent, since for each packed message we need to add only one header instead of a header per message. Similarly, it reduces the contention on the Ethernet and the load on the receiving CPUs.

The rest of this paper is organized as follows: Section 2 describes the system that we used for taking our measurements. Section 3 describes the different protocols that we measured. Section 4 presents the results of our measurements. Other protocols that were not checked by us are described in Section 5, and we conclude with a discussion in Section 6.

2 The System

All measurements were taken on a cluster of 3 Sparc 10s and 3 Sparc 20s, connected by a 10Mbps Ethernet, running Horus [21] over SunOS 4.1.3, and using UDP/IP with IP-Multicast for broadcasts. We dedicate the rest of this section to describing Horus.

Horus is a layered architecture for reliable group communication in asynchronous distributed environments. That is, applications in Horus are made of several layers, each of them implementing a specific and well-defined concern, such that together these layers provide the needed semantics. Each layer must provide the same interface, known as the Horus Uniform Group Interface, to higher and lower layers, which allows for maximum reuseability of code. For example, all applications that need reliable communication use the same nak layer, and do not have to implement a nak protocol themselves. Being able to reuse existing code has three main advantages: (a) it shortens the development time of new applications, (b) code that was written specifically to do a certain task by someone who specialized in it is likely to be more efficient for this specific task than code that was written as part of a bigger application, and (c) an existing code that was already used in several places is likely to be better debugged and tested than code that was just written as part of a new application.

The complexity of asynchronous distributed environments makes these advantages even more significant. In particular, the ability to implement a specific well-defined problem, e.g., total delivery, without having to deal with all the other problems that arise in an asynchronous distributed environment, e.g., failures and unreliable communication media, greatly simplifies the code, and makes the task of verifying the code much simpler.

As a system for group communication in asynchronous distributed environments, Horus includes a *virtually synchronous* (VS) layer, which *simulates* a fail-stop model for higher layers. The VS layer of Horus simulates the fail-stop model by passing **view** events, which include estimates on who the reachable and live processes in the system are, to the higher layers. The VS layer guarantees that between every two consecutive **view** events V_1 and V_2, all the processes that appear in both these views will receive the same set of messages, that every message sent between V_1 and V_2 will be delivered before V_2, and that no message from a process which is not included in V_1 will be delivered between V_1 and V_2.

Whenever the VS layer suspects that a certain process has failed, or that a process that was not included in the most recent view wishes to join the view, it generates a **flush** event to the higher layers. This event informs the higher layers that the VS layer is now executing the flush protocol, which is responsible for delivering all the messages that were sent since the last **view** event to all live processes. A new **view** event is finally generated when the flush protocol terminates. Layers that sit on top of the VS layer are not allowed to submit new messages to the VS layer during the flush protocol, i.e., between a **flush** event and the following **view** event.

One important aspect of Horus is the way in which messages are handled. Messages in Horus consist of an optimized data-structure that can be passed between layers with zero copying, similar to the way *fbufs* are handled [14]. In order to send a message, the application creates a message structure, adds the data to this message, and passes a pointer to this message structure to the lower layer. Each layer can add some headers to the message, if so required to provide the intended semantics, and pass the message to the lower layer. At the receiving side, the bottom most layer creates a message structure to hold the data that is received from the network. Each layer strips off the headers that were added by the corresponding layer at the sender side, and passes a pointer to this message structure to the higher layer, until it reaches the application.

More details about the architecture of Horus, the virtually synchronous layers of Horus, and the implementation of the flush protocol can be found in [15, 20, 21]. We explain how the VS layer helps in providing fault tolerance to the total ordering protocols in Section 3.1.

3 The Protocols

We have chosen four different protocols, nicknamed **Totem**, **Total**, **Dynseq**, and **Dynord**. For two of these protocols we have created a version which includes packing of messages. We have nicknamed these optimized version as **Tomfc** and **Dysfc**.

Totem

This protocol is used by the Totem project [2]. It rotates a token among a list of participating processes. A process that wishes to send a message, must wait until it receives the token, and has to buffer its messages until then. Whenever a process receives the token, it must send all the messages that it have buffered so far, and then pass the token to the next process on the list.[1] If the token holder had messages to send, then the token is piggybacked on its last message. Otherwise, the token holder must generate a specific message for the token.

Totem is designed to give extremely high throughput for applications in which everyone is constantly sending messages to everyone. On the other hand, the maximum and average theoretical latency of messages depend explicitly on the number of processes in the system: Assuming n processes in the system, a message is blocked at the sender for $n/2$ token passes on the average until the sender can send it, while in the worst case a message may be blocked for $n-1$ token passes, even if there is only one sender in the system. In practice, however, this potentially large delay is somewhat balanced by the fact that once a message is sent, it is sent without any contention on the network.

[1] In fact, in the Totem project, a process can only send a limited number of messages when it receives the token. If this process has more messages to send, these messages must wait for the next token arrival in order to be sent.

Tomfc

This protocol is the same as **Totem**, except that if the token holder has messages to send, then it packs all these messages and send them as one packed message. In this case, the token is also piggybacked on the message.

Total

This protocol is an optimized version of **Totem** for scenarios in which only a small subset of the processes are sending messages, while the rest of the processes are only receiving messages. That is, in order to send messages, a process must first obtain the token, and has to buffer its messages until it gets the token. In order to obtain the token, the process has to broadcast a token request. A token holder that is aware of token requests, chooses one of the processes that sent these requests and passes the token to this process. When a process which requested the token receives the token, it packs all of its buffered messages, and send them as one packed message. If, at this time, it is aware of other token requests, it piggybacks the token on this message, together with a new token request. Otherwise, it keeps the token, and is allowed to send messages whenever the application program requests it to. To further optimize the protocol, a process that sent a token request, remembers this and will not send any more token requests until it receives the token.

The optimization that a token holder which sends messages piggybacks a token request on the message that passes the token to someone else is intended to exploit the locality principle. That is, it is assumed that if the application at a process requested to send some messages, then there is a good chance that it will also need to send more messages soon after. This way, if there is a set of processes that is constantly sending messages, no special messages needs to be sent for token requests, since all the token requests will be piggybacked on existing traffic. The net effects is that the token is rotating among the senders only. In the special case when there is only one sender, no token is passed around at all.

The theoretical latency of messages in **Total** depends only on the number of actual senders. That is, if there are n processes but only $k < n$ senders, each message will be blocked only for $k/2$ token passes on the average, and no more than $k-1$ token passes in the worst case. **Total** also has the advantage that once a message is sent, there is little or no contention on the network. The only possible contention in this protocol can come from token requests which are sent at the same time, or together with a broadcast by the token holder. However, there can be at most $k-1$ such token requests at the same time, and even this is highly unlikely due to the optimizations used for token requests.

Dynord

In this protocol, a process that wishes to broadcast a message, broadcasts the message immediately. However, when processes receive a broadcast message, they are not allowed to deliver it immediately. Instead, one process in chosen to be the *orderer*, and is responsible to broadcast, every now and then, an `order` message that specifies an order for the messages received by the orderer so far. When a process receives an `order` message, it delivers the messages ordered by the `order` message in the order specified by this message. Messages are always ordered in the order they are received by the orderer, which allows the orderer to deliver messages that it receives immediately. Moreover, for improved performance, an `order` message is piggybacked on every regular message broadcast by the orderer. Finally, if a process other than the orderer becomes much more active than the orderer, the job of being the orderer is passed to this active process.

Dynord was used by ISIS [7, 8], although the actual implementation in ISIS included explicit code for dealing with unreliable communication, failures, and stability detection. The theoretical latency of messages in **Dynord** depends on the one-way latency of regular messages, the one-way latency of an `order` message, and the elapsed time between every two consecutive `order` messages. In practice, however, since process are allowed to send messages freely, the one-way latency of messages is expected to be higher than optimal due to contention on the network. This contention is also expected to degrade the throughput of the protocol.

Dynseq

In this protocol, a special process is chosen to be the *sequencer* of the system. Hence, whenever a process wants to broadcast a message, it sends this message to the sequencer. The sequencer, on its part, broadcasts the message to everyone, and delivers it locally. Of course, when the sequencer wants to broadcast a message, it just broadcasts this message to everyone, and delivers it locally. Also, whenever a process other than the sequencer becomes very active, the job of being the sequencer is passed to this process.

Dynseq was used by the Amoeba project [16], although the actual implementation in Amoeba had to explicitly deal with unreliable communication, failures, and stability detection. The theoretical latency of **Dynseq** is the best, since each message requires only two one-way delays. However, in practice, this protocol also suffers from high contention on the network, which increases the actual one-way delays and degrades the throughput. Also, the use of a sequencer that has to send all the messages it receives reduces the theoretical bandwidth of the system by at least a factor of two.

Dysfc

This protocol is essentially the same as **Dynseq**, except that here processes are not allowed to send their messages all the time. Instead, messages are buffered and every l millisecond they are packed and sent as one packed message. In this case, we have chosen l to be one millisecond, since it is less than the minimal expected one-way user-to-user latency. Thus, the idea is that we delay each message, but only for a very short period, and by that we increase the throughput dramatically. By sending only one (packed) message every l milliseconds, the contention on the network and the load on the receiving processors are substantially decreased, which result in a lower user-to-user latencies than those achieved when messages are sent without this intentional blocking.

3.1 Fault Tolerance and Stability Detection

Since all the above protocols were implemented in Horus, and were assumed to run over the virtually synchronous layer, the fault tolerant part of these protocols is fairly simple, and is basically the same in all of them. That is, whenever a **flush** event is delivered from the lower layers, the protocol first sends all of its buffered messages (if there are any) as `unordered` messages, and from now on buffers for broadcast new messages that the application wishes to send. In the meantime, the virtually synchronous layer of Horus runs the flush protocol, which guarantees that every message that was broadcast after the last previous **view** is delivered by all the live processes. Messages that are delivered during the flush protocol and are labeled as `unordered` are buffered for delivery, while other messages are treated as regular protocol messages.

When the flush protocol terminates, a **view** event is delivered from the lower layers. At this point, the protocol at each process knows that it received all messages that were sent since the last view. In particular, it knows that all live processes have the same set of `unordered` messages buffered for delivery. Hence, some deterministic rule is used by all live processes to deliver their buffered `unordered` message. In our case, we have chosen to deliver messages according to the rank of their invoking process in the membership list, where messages by the same process are delivered in the order in which they were sent by that process. This rule seemed to be the most natural one, and was the easiest to implement. However, other deterministic rules can also be used.

Finally, after delivering all the `unordered` messages, the **view** event is passed to the upper layers (the application), and normal execution is resumed. Of course, application messages that were buffered for broadcast during the flush protocol are sent before any new application message can to be sent.

Note that all the protocols we have chosen either need a token holder, or some special process to order messages. Here, again, some deterministic rule which is based on the ranks of the processes in the membership list is used to choose this member at the end of each flush protocol. In our case, following a **view** event the token holder/orderer/sequencer is always chosen to be the member with the smallest rank. Thus, this "election" can take place without any additional message passing or delays.

Reliable delivery of messages is typically achieved by maintaining copies of messages that were sent and received, so these messages can be retransmitted to members that have failed to receive them. In order to make sure that the buffers that are used to store these messages do not grow too much, a stability detection mechanism must be used. That is, whenever a process learns that a message that it stored has been received by everyone, it declares this message stable and can discard it.

Many protocols for total delivery that are implemented directly on top of an unreliable network have to explicitly deal with stability detection and providing reliable delivery. In our case, there are other Horus layers whose job is to guarantee reliable delivery and stability detection. In particular, there are currently two different layers for detecting stability in Horus. One layer, nicknamed STABLE, piggybacks stability information on existing traffic, and may sometimes generate specific stable messages when there is not enough existing traffic. The other layer, nicknamed PINWHEEL, rotates a token around the members which collects the stability information. Each of these layers performs differently under different circumstances. However, each of the total ordering protocols described in this paper can be run with any of these layers, so the performance of the system can be adjusted at runtime to fit the anticipated communication pattern of the system.

4 Simulation Results

As described before, all of our measurements were taken on a cluster of 3 Sparc 10s and 3 Sparc 20s, connected by a 10Mbps Ethernet, running Horus over SunOS 4.1.3, and using UDP/IP with IP-Multicast for broadcasts. All measurements were taken with the same Horus stack (except for the total ordering protocol which was different), namely, "protocol-name:MBRSHIP:FRAG:NAK:COM". (The COM layer is responsible for the actual interface with udp; the NAK layer uses a negative acknowledgment protocols to provide link reliability; the FRAG layer fragments and reassembles large messages; the MBRSHIP layer is responsible for keeping track of membership changes and executing the flush protocol, as discussed in Section 2.)

Our measurements were obtained using two similar test programs. The first program generated rounds of messages, such that in each round every process sends a burst of m message, where m is a parameter of the test, and waits until it received all messages broadcast by all other processes in the system before it can start the next round. In the second program, only one process sends the bursts of messages, and then waits for an acknowledgment from all other processes before it can start a new round of messages.

All measurements were run late at night, when both the workstations and the network were fairly idle. For each of the protocols and each value of m that was tested, we have run the test programs several times, each of them consisting of 10 rounds, and calculated the the average of the results that were obtained. These results are summarized in the graphs in Figures 1, 2, 3 and 4.

Figures 1 and 2 show the throughput and latency-per-round for various burst lengths when all processes are broadcasting messages, while Figures 3 and 4 show the throughput and latency-per-round for the same burst lengths when only one process is broadcasting the messages. Note that our graphs show the latency-per-round, and not the average (one) message latency. We assume that usually when an application generates a burst of messages, the important issue is the delivery time of the entire burst, and not of a single message. In any

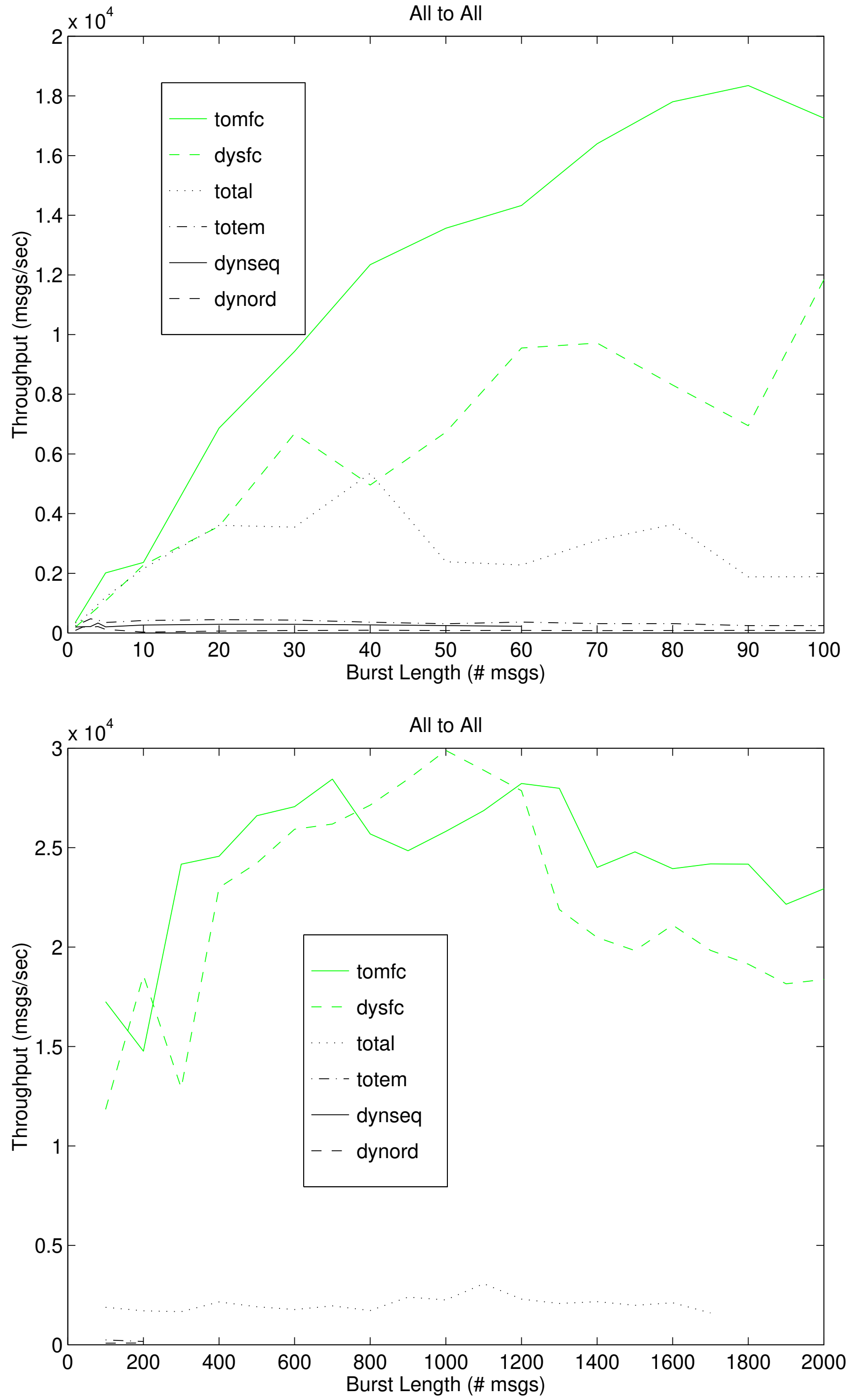
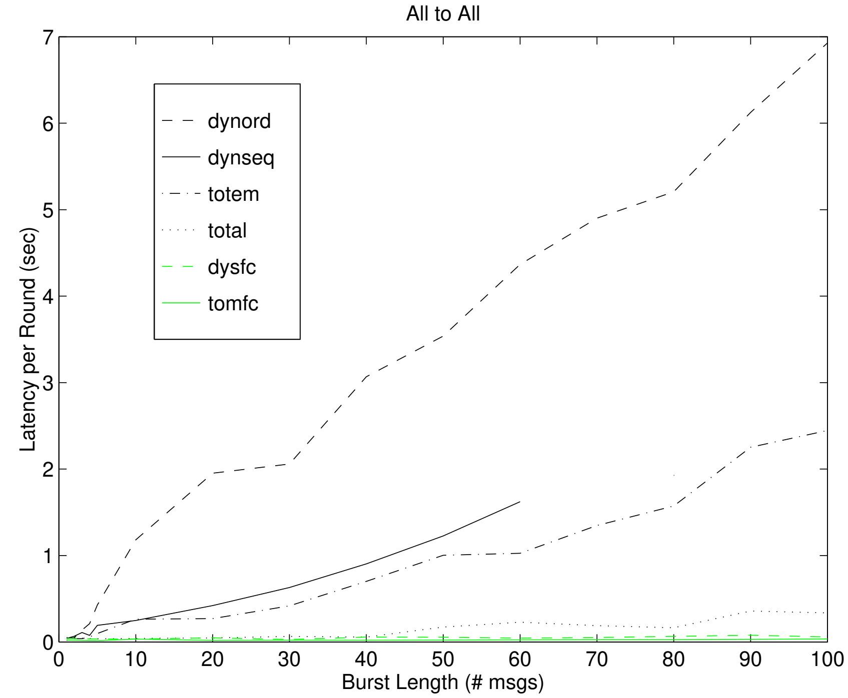


Figure 1: Message throughput for all-to-all communication



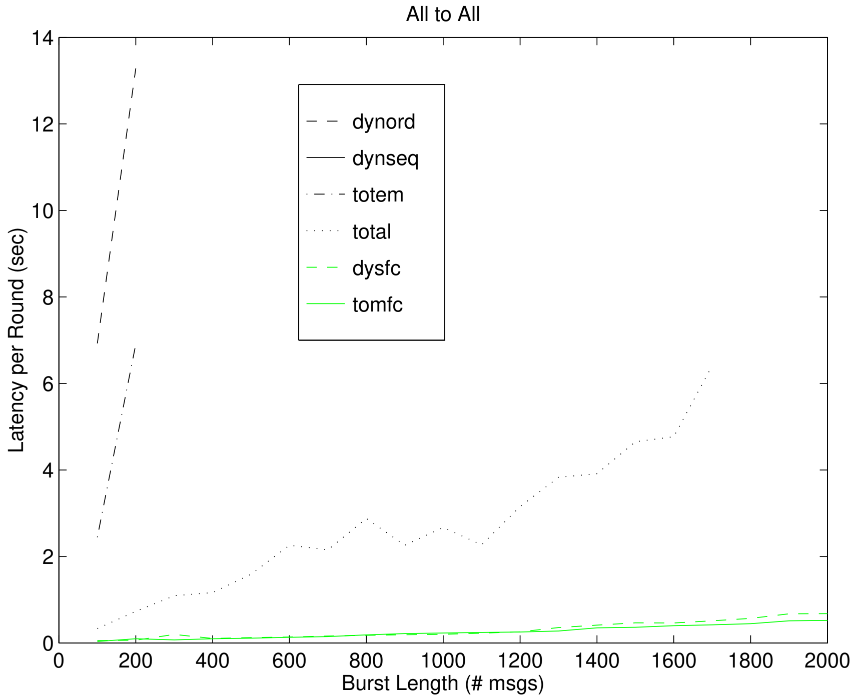


Figure 2: Round latency for all-to-all communication

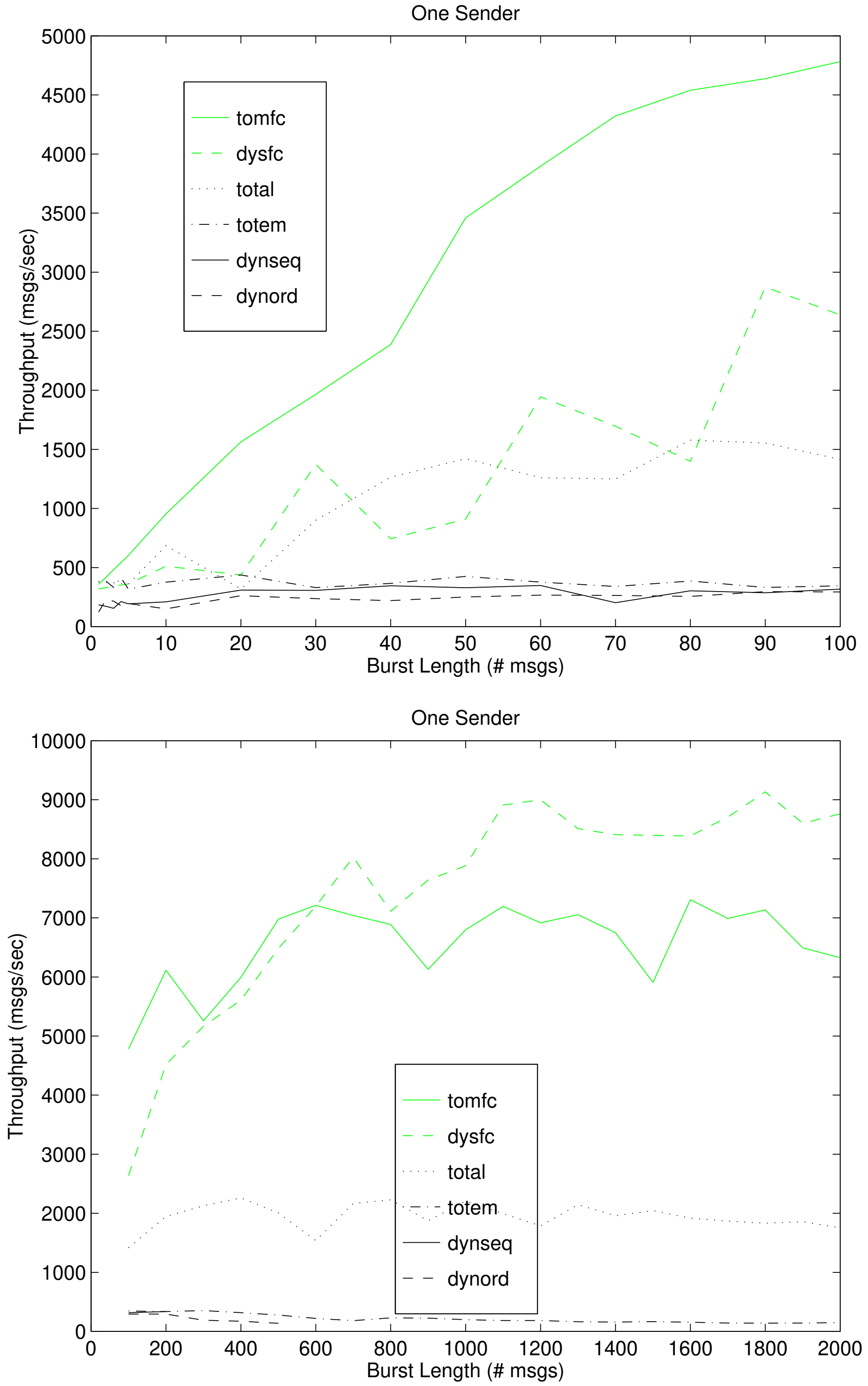


Figure 3: Message throughput for one sender

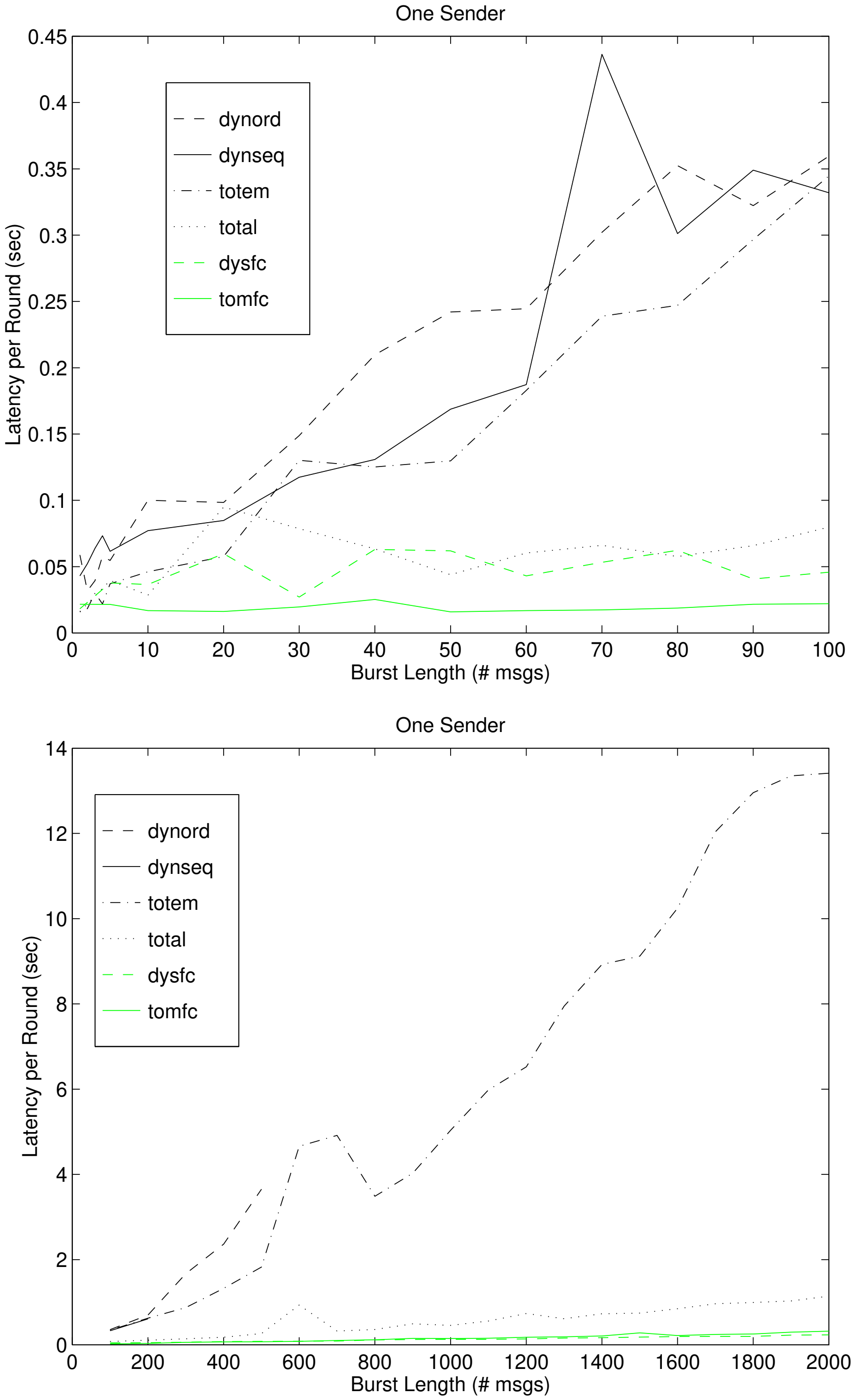


Figure 4: Round latency for one sender

case, the layers of Horus that were chosen implement asynchronous communication, in which control is returned to the sender as soon as possible, even if the message cannot be sent immediately. Hence, the latency-per-round is a reasonable estimate for the maximum latency of one message, and is probably not too far from the average latency as well.

As can be seen, already with bursts of length 10, the performance of **Dysfc** and **Tomfc** is an order of magnitude better than the performance of **Dynseq** and **Totem**, and for bursts larger than 100, the performance of **Dysfc** and **Tomfc** is already two order of magnitudes better than the performance of **Dynseq** and **Totem**. Also, **Dynseq** and **Dynord** were not able to sustain bursts of more than 60 and 200 messages, respectively, in the all-to-all case, and were not able to sustain bursts of more than 200 and 500 messages, respectively, in the one sender case. On the other hand, **Dysfc** continued to operate even at bursts of more than 2000 messages. In the middle we find **Total** that was able to sustain bursts of up to 1700 messages in the all-to-all case, but was not able to sustain longer bursts of messages.

Total's performance is somewhat expected, when considering it's packing policy. At fairly low loads, it packs all (or at least most) of the messages, and therefore performs as well as **Dysfc** and **Tomfc**. However, once the bursts becomes very long, the performance of **Total** degrades since the percentage of messages it can pack becomes smaller. In particular, in the single sender case, **Total** does not do much packing, since once the sender receives the token, it continues to hold it. Thus, for a large m, each burst involves m actual messages, while **Totem** and **Dynseq** generate only a few packed messages.

Among the non-packing protocols, **Totem** exhibits the best behavior. This was expected for the all-to-all case, but came as a surprise in the one sender case. It seems that when enough messages are sent, the lack of contention on the Ethernet outweighs the cost of rotating the token. Note that according to our measurements, for small messages, **Dynseq** behaves better than **Dynord**. This corresponds to the findings of Cristian, Beijer, and Mishra regarding these protocols that were obtained by simulation [12].

We believe that the superior performance of the packing protocols, i.e., **Tomfc** and **Dysfc**, can be attributed to three factors:

1. Horus has a substantial headers' overhead associated with each message. If we denote this byte overhead by h, then by packing m application messages as one message, the headers' overhead for these messages becomes only h, instead of $h \times m$ which is required without packing.

2. By packing messages, less messages need to gain access to the Ethernet, so there is less contention, and even the existing contention can be solved much faster.

3. By packing k messages, a receiving CPU must handle only one interrupt for these k messages, instead of k interrupts which are required without packing.

We believe that the humps in the graphs of **Dysfc** represent crossing a multiple of an IP-multicast packet size. That is, each time we slightly pass a multiple of an IP packet size, the throughput drops due to the need to send another packet. As the utilization of this new packet increases, so does the performance, until the next time we pass a multiple of an IP packet size.

5 Related Work

The Chang and Maxemchuk protocol [10], aka CM, is similar to **Dynord**, except that the job of being the orderer is passed after each `order` message to another process. Unlike **Dynord**, in CM, a sender of a message m has to block until it receives an `order` message for m. Also, since CM has to explicitly deal with failures and unreliable communication, a message can only be delivered after l orderers have seen it, where l is a parameter of the protocol.

The Train protocol [11] is somewhat similar to **Totem** in the sense that there is a token which constantly rotates among the members. However, in Train, the token also carries the messages themselves. That is, a process that receives the token has to deliver all the messages which are carried by it, and then adds all the messages it wishes to send to the token. This is why a token is also called a "train" in this protocol.

The Tandem protocol [9] is somewhat similar to **Dynseq**, except that a sender is not allowed to send more than one message as a time. A slightly optimized version of Tandem, called The Positive Acknowledgment protocol, aka PA, was studied in [12]. In PA, messages are also sent to the sequencer. Whenever the sequencer receives an out of order message from some member, it buffers this message until all previous messages from that member are received by it. When the sequencer receives an in order message from a member, it sends this message and all other buffered messages from the same process that are now in order to all other processes. Also, every member that receives a message from the sequencer has to acknowledge this message. Note that this protocol has to explicitly deal with unreliable communication, since it was not developed to run on top of a system that provides these guarantees.

The Pinwheel protocol [13] is similar to CM. However, in Pinwheel, negative acknowledgements are used, so a sender does not have to block until its messages are acknowledged by the token holder. Also, in Pinwheel the next token holder is always the next member in the membership list. An improved version of Pinwheel that uses a *newsmonger* was also introduced in [13]. In this improved version of the protocol, if no process sends any messages for a while, a newsmonger starts a round of information gathering which helps to reduce the delivery and stability times.

The On-Demand protocol [1] is similar to **Total**, except that the token holder is allowed to keep the token for a while even if it does not have any messages to send and there are already other token requests. This is done in compliance with the locality principle, which suggests that if a process had something to send, then there is a good chance that it will have more things to send in the near future. On the other hand, in the On-Demand protocol, a token-request is not generated automatically when the token is passed, as done in **Total**.

All the total ordering protocols that we have discussed so far are asymmetric in the sense that at each given moment there is some process which has a different role than the others. However, there are also fully symmetric protocols, e.g., [4]. In these protocols, processes timestamp their messages using Lamport clocks [17]. To deliver a message, a process must have received all the previous and concurrent messages in Lamport's time. When this happens, messages are delivered according to their lamport's time, breaking symmetry using the ranks of their invoking processes. Of course, in order to ensure progress, processes that have no real messages to send need to broadcast a dummy message every now and then. Fully symmetric protocols usually exhibit high latencies and low throughput.

Recently, the use of Isotach networks as a tool for providing total ordering was suggested [18]. In Isotach networks, the *logical delivery time* of messages is always the same as the number of hops the message has to travel to its destination. Some deterministic rule which must maintain the per process order is used to order messages which are received at the same logical time. Thus, it is possible to guarantee total ordering by making sure that all copies of the same message will be delivered to all its targets at the same logical time, and that no message will be delivered in logical time before all previous messages by the same sender. This is done by delaying messages for different periods of time, based on how far their destinations are. For example, if one destination p of a message m is located 2 hops away, and another destination q of m is located 5 hops away, then the copy of m that is intended for q can be sent immediately, while the copy which is designated for p would be buffered for 3 logical clock pulses.

Cristian, Beijer, and Mishra have compared the Train, PA, Amoeba, and ISIS protocols in [12]. (Recall that Amoeba is similar to **Dynseq**, but includes the details of maintaining stability and failure recovery, and ISIS is the same as **Dynord**, again with explicit stability detection and failure handling.) This comparison was carried out by simulation, and assumed that the process-to-process latency does not depend on the load of the system and the number of senders. In contrast, our paper compared the protocols in a real system, where the number of messages sent influences their latencies and throughput in two ways: the contention on the network and the load on the receiving CPUs. For example, the handling time of interrupts that arrive at a rate of k interrupts per second is larger than multiplying the handling time of one interrupt in an idle state by k. The advantage of using a simulator for comparison is that the results are free from operating system related effects, which gives a better insight on the behavior of the protocols themselves. On the other hand, network contention and CPU load are hard to simulate accurately, but are definitely part of what a user of the system would experience.

6 Discussion

Total ordering protocols can greatly simplify the design of many distributed applications. However, in order to be useful, these protocols must also provide good performance. In this paper we have compared the throughput and latency of four different types of protocols, with different optimizations. Our conclusion is that at least for small messages, packing messages is the most important optimization that a protocol can offer, even if this means buffering messages for a short period of time before sending them, in order to have more messages to pack.

Our measurements were taken for messages of size 1 byte, although due to padding done by Horus, they are valid for 4 byte messages as well. Most applications that can benefit from total ordering protocols, e.g., control applications, monetary systems, and brokerage systems, need to send only small messages. However, messages sent by these application are usually slightly bigger than what we have measured. We intend to continue our measurements in order to find the largest size of messages for which packing would still give a dramatic improvement in performance. We believe that these tests will show that for most practical applications that require total ordering of messages, packing is the most important optimization that can be used.

Also, our measurements were taken over Ethernet. By definition, this system is bound to suffer from collisions. It would be interesting to investigate the effect of message packing in point-to-point networks, e.g., ATM. Some random tests that we have done with ATM using the U-net direct interface, recently developed in Cornell University [6], indicate that message packing improves the performance for these type of networks as well. However, more systematic measurements should to be taken in order to determine the exact effect of packing on message latencies and throughput in ATM networks.

Finally, in this paper we have not checked the behavior of protocols during crashes. This is an important question that we intend to investigate in the future.

References

[1] G. Alvarez, F. Cristian, and S. Mishra. On-Demand Asynchronous Atomic Broadcast. Technical Report CS95-416, Dept. of Computer Science, University of California, San Diego, 1995.

[2] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. In *Proc. of the 13th International Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[3] H. Attiya and R. Friedman. A Correctness Condition for High-Performance Multiprocessors. In *Proc. of the 24th ACM Symp. on the Theory Of Computing*, pages 679–690, May 1992. Revised version: Technical Report #767, Department of Computer Science, The Technion. Submitted for publication.

[4] H. Attiya and J. Welch. Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[5] H. Bal, F. Kaashoek, and A. Tanenbaum. ORCA: A Language for Parallel Programming of Distributed Systems. *IEEE Transaction on Software Engineering*, 18(3):190–205, March 1992.

[6] A. Basu, V. Buch, W. Vogels, and T. von Eiken. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. Technical report, Department of Computer Science, Cornell University, April 1995.

[7] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pages 123–138, December 1987.

[8] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[9] R. Carr. The Tandem Global Update Protocol. *Tandem Systems Review*, June 1985.

[10] J. Chang and N. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[11] F. Cristian. Asynchronous Atomic Broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, February 1991. Also: 1st IEEE Workshop on Management of Replicated Data, Houston, TX, November, 1990.

[12] F. Cristian, R. Beijer, and S. Mishra. A Performance Comparison of Asynchronous Atomic Broadcast Protocols. *Distributed Systems Engineering Journal*, 1(4):177–201, 1994.

[13] F. Cristian and S. Mishra. The Pinwheel Asynchronous Atomic Broadcast Protocols. In *Proc. of the 2nd International Symposium on Autonomous Decentralized Systems*, Phoenix, AZ, 1995. Also: Technical Report CSE93-331, Department of Computer Science & Engineering, University of California, San Diego.

[14] P. Druschel and L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pages 189–202, December 1993.

[15] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical Report TR95–1491, Department of Computer Science, Cornell University, March 1995.

[16] F. Kaashoek, A. Tanenbaum, S. Hummel, and H. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[17] L. Lamport. Time, Clocks and the Ordering of Event in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[18] P. Reynolds, C. Williams, and R. Wagner. Empirical Analysis of Isotach Networks. Technical Report 92-19, University of Virginia, Dept. of Computer Science, June 1992.

[19] A. Setubal and G. Gerber. Fault-Tolerant Distributed Database for Supervisory Control System. In *Proc. of Workshop Verteilte Datenbaksysteme*, Karlsruhe University, Karlsruhe, Germany, September 1991.

[20] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A Framework for Protocol Composition in Horus. In *Proc. of the 14th ACM Symposium on Principles of Distributed Computing*, August 1995.

[21] R. van Renesse, K. P. Birman, and T. M. Hickey. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report 94-1442, Cornell University, Dept. of Computer Science, August 1994.