High Throughput Total Order Broadcast for Cluster Environments

Rachid Guerraoui
IC EPFL, Switzerland
CSAIL MIT, USA

Ron R. Levy
IC EPFL,
Switzerland

Bastian Pochon
IC EPFL,
Switzerland

Vivien Quéma
INRIA, France
Univ. di Roma 1, Italy

Abstract

Total order broadcast is a fundamental communication primitive that plays a central role in bringing cheap software-based high availability to a wide array of services. This paper studies the practical performance of such a primitive on a cluster of homogeneous machines.

We present FSR, a (uniform) total order broadcast protocol that provides high throughput, regardless of message broadcast patterns. FSR is based on a ring topology, only relies on point-to-point inter-process communication, and has a linear latency with respect to the total number of processes in the system. Moreover, it is fair in the sense that each process has an equal opportunity of having its messages delivered by all processes.

On a cluster of Itanium based machines, FSR achieves a throughput of 79 Mbit/s on a 100 Mbit/s switched Ethernet network.

1 Introduction

Motivation. As an ever increasing number of critical tasks are being delegated to computers, the unforeseen failure of a computer can have catastrophic consequences. Unfortunately, the observed increase of computing speed as predicted by Moore's law has not been coupled with a similar increase in reliability. However, because of rapidly decreasing hardware costs, ensuring fault tolerance through replication is gaining in popularity. The key to making replication work is a well designed software layer that hides all the difficulties behind replication from the application developer and renders it transparent to the clients [24].

At first glance, the idea is simple. Each process maintains a single copy of the object (representing a software service) that is to be replicated. All invocations are broadcast to all processes (i.e., replicas), which perform them on their copies[1]. A key underlying ordering mechanism ensures that all processes perform the same operations on their copies in the same order, even if they subsequently fail. This mechanism is encapsulated by a communication abstraction called *total order broadcast* (TO-broadcast) [25]. We consider the uniform variant that guarantees consistency for processes that subsequently fail. This abstraction ensures in particular the following properties for all messages that are broadcast: (1) *Agreement*: if a process TO-delivers a message m, then all correct processes eventually TO-deliver m; (2) *Total order*: if some process TO-delivers some message m before message m', then no process TO-delivers m' before m.

Clearly, the *throughput* of a TO-broadcast protocol is crucial to the throughput of the associated replication mechanism. It captures the number of requests that can be handled by the replicas under high load.

The problem addressed in this paper is that of devising a high-throughput TO-broadcast protocol for a cluster of homogeneous machines interconnected by a fully switched LAN. Even though it should also be efficient in arbitrarily large clusters, it has to be optimized for relatively small clusters (less than 15 machines), because in practice it is not very useful to replicate the same state on a lot of machines. Similarly, performance should be measured for fairly static environments with few failures where only few machines leave or join the system. These kinds of environments are common for e-commerce applications such as fault-tolerant J2EE clusters [37].

Modeling. The first step in reasoning about the throughput of a communication abstraction is to determine a model to precisely represent such throughput.

Various models have been proposed to reason about message passing complexity. Nevertheless, none of them is adequate for modeling clusters of homogeneous machines interconnected by fully switched LANs. Either they assume that processes can receive several messages at the same time [30], or they do not assume the existence of a broadcast primitive [15, 5].

In this paper, we propose to analyze protocols using a slightly modified version of the popular round-based message passing communication model [30]. The model we propose assumes that processes can send a message to one

[1]In practice, invocations that do not change the state of the replicated object do not need to be broadcast and can be performed in parallel.

or more processes at the start of each round and can receive a single message sent by other processes at the end of the round.

Throughput can thus be defined as the average number of *completed* TO-broadcasts per round. A complete TO-broadcast of message m meaning that all processes TO-delivered m. We consider that a TO-broadcast algorithm is *throughput efficient* if its throughput is higher than or equal to 1. This means that on average all processes TO-deliver one message per round.

Throughput. Numerous TO-Broadcast protocols have been published [17]. Protocols relying on communication history [35, 31, 19, 34, 32] and destination agreement [10, 7, 29, 21, 2] do not have good throughput as they rely on a quadratic number of messages and an underlying consensus sub-protocol. Protocols relying on a fixed sequencer also inherently have low throughput [26, 3, 9, 22, 8, 41]. While requiring fewer messages than the previously mentioned class of protocols, they still exhibit bad throughput because the sequencer becomes a bottleneck. Protocols using moving sequencers [12, 40, 27, 14] have been proposed to overcome the limitation of fixed sequencer protocols. While significantly improving the throughput, these protocols do nevertheless not achieve higher throughput than 1 due to the impossibility of piggy-backing acks in certain broadcast patterns (e.g. 1-to-n meaning that a single process TO-broadcasts to all other processes). Finally, a class of TO-broadcast protocols, called privilege-based protocols [20, 13, 18, 1, 23], uses a ring topology of processes and a token passed among processes to grant the privilege of broadcasting. These protocols provide high throughput in the 1-to-n and n-to-n case (all processes TO-broadcasting to all other processes), but not in the k-to-n case ($k \neq 1, n$). For instance, when two processes simultaneously want to broadcast messages, for fairness reasons, the token is constantly passed from one sender to the other, which reduces the throughput.

Contributions. In this paper we present FSR, a uniform total order broadcast protocol that relies on point-to-point communication channels between processes. FSR is hybrid and uses both a fixed sequencer and a ring topology (hence the name). Similarly to the train protocol [13], each process only sends messages to the same single process. Unlike the train protocol however, messages in FSR are sequenced by a fixed process in the ring. These two characteristics ensure throughput efficiency and fairness, regardless of the type of traffic. In our context, fairness conveys the equal opportunity of processes to have their TO-broadcast messages eventually TO-delivered. Moreover, FSR has linear latency with respect to the number of processes.

We give a careful analysis of FSR fault-tolerance, scalability and fairness, as well as describe the performance of its implementation.

Roadmap. Section 2 reviews existing TO-broadcast protocols and compares them to FSR. Section 3 describes the system model. Section 4 describes FSR in detail. Section 5 provides performance results and analysis. Section 6 concludes the paper.

2 Related Work

The five following classes of TO-broadcast protocols were identified in [17]: fixed-sequencer, moving sequencer, privilege, communication history and destination agreement. In this section, we only survey time-free protocols, for these are comparable to FSR as they do not assume synchronized clocks.

2.1 Fixed Sequencer

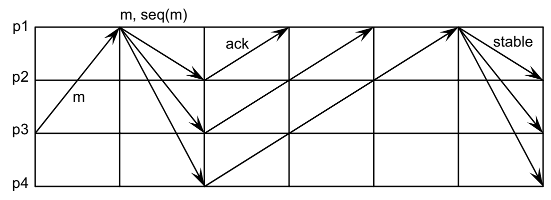

Figure 1. Fixed sequencer-based TO-broadcast.

In a fixed sequencer protocol [26, 3, 9, 22, 8, 41] (Figure 1), a single process is elected as the sequencer and is responsible for the ordering of messages. The sequencer is unique, and a new sequencer is elected only in the case the previous sequencer fails. Three variants of the fixed sequencer protocol exist [4], each using a different communication pattern. Fixed sequencer protocols exhibit linear latency with respect to n [16], but poor throughput. The sequencer becomes a bottleneck because it must receive the acknowledgments (acks) from all processes[2] and also receive all messages to be broadcast. Note that this class of protocols is popular for *non*-uniform total order broadcast protocols since these do not require all processes to send acks back to the sequencer, thus providing much better latency and throughput.

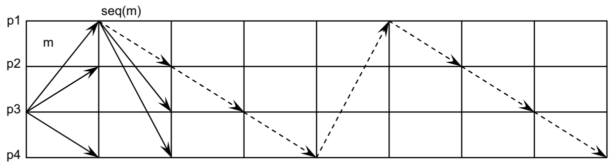

Figure 2. Moving sequencer-based TO-broadcast.

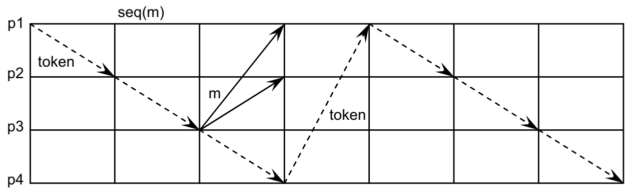

Figure 3. Privilege-based TO-broadcast.

2.2 Moving Sequencer

Moving sequencer protocols [12, 40, 27, 14] (Figure 2) are based on the same principle as fixed sequencer protocols, but allow the role of the sequencer to be passed from one process to another, even if no failures occur. This is achieved through a token that carries a sequence number and constantly circulates among the processes. The motivation is to distribute the load among sequencers, thus avoiding the bottleneck caused by a single sequencer. When a process p wants to broadcast a message m, p sends m to all other processes. Upon receiving m, the processes store it into a receive queue. When the current token holder q has a message in its receive queue, q assigns a sequence number to the first message in the queue and broadcasts that message together with the token. For a message to be delivered, it has to be acknowledged by all processes. Acks are gathered by the token. Moving sequencer protocols have a latency that is worse than that of fixed sequencer protocols [17]. On the other hand, they achieve better throughput, although not higher than 1. Figure 2 shows a 1-to-n broadcast of one message. It is clear from the figure that it is impossible for the moving sequencer protocol to deliver one message per round. The reason is that the token must be received at the same time as the broadcast messages and the protocol thus cannot achieve high throughput. Note that fixed sequencer protocols are often preferred to moving sequencer protocols because they are much simpler to implement [17].

2.3 Privilege

Privilege-based protocols [20, 13, 18, 1, 23] (Figure 3) rely on the idea that senders can broadcast messages only when they are granted the privilege to do so. The privilege to broadcast (and order) messages is granted to only one process at a time, but this privilege circulates from process to process in the form of a token. When a process wants to broadcast a message, it must first wait until it receives the token. As explained in [14], there is a trade off between privilege-based protocol performance and fairness. To see why, consider the case where two processes located at opposite sides of the ring simultaneously broadcast bursts of messages. Either one of the processes keeps the token, which is unfair, or the token is constantly passed from one sender to the other one, which drastically reduces the throughput.

[2]Acknowledgments in the fixed sequencer can only be piggy-backed when all processes broadcast messages all the time [16].

2.4 Communication History

As in privilege-based protocols, communication history-based protocols [35, 31, 19, 34, 32] use sender-based ordering of messages. They differ however by the fact that processes can send messages at any time. Messages carry logical clocks that allow processes to observe the messages received by the other processes in order to learn when TO-delivering a message does not violate the total order. Communication history-based protocols have poor throughput because they rely on a quadratic number of messages exchanged for each message to be TO-broadcast.

2.5 Destination Agreement

In destination agreement-based protocols, the delivery order results from an agreement between destination processes. Many such protocols have been proposed [10, 7, 29, 21, 2]. They mainly differ by the subject of the agreement: message sequence number, message set, or acceptance of a proposed message order. These protocols have relatively bad performance because of the high number of messages that are generated for each broadcast. Indeed, they rely on consensus that in a way is modular, but which is very expensive in terms of latency and message complexity.

Note that hybrid protocols, combining two different ordering mechanisms have also been proposed [19, 36, 39]. Most of these protocols are optimized for large scale networks, using multiple groups or optimistic strategies.

3 Model

We consider a system with n processes which have access to a failure detection module, which implements a *Perfect* failure detector P [11]. Using P we implement a virtually synchronous communications (VSC) [6] layer which ensures consistent message delivery while allowing processes to join and leave during the execution of the protocol.

Processes communicate through point-to-point channels. Moreover, we assume a fully connected network, where each pair of processes is connected. The network is full-duplex, by which we mean that each node can simultaneously send and receive messages. There are also separate collision domains: process p_1 can send messages to p_2 without interfering with process p_3 sending messages to p_4.

Evaluating the performance of a communication abstraction requires a performance model. Some models only address point-to-point networks, where no native broadcast primitive is available [15, 5]. A recent model [38] proposes to evaluate total order broadcast protocols, assuming that a process cannot simultaneously send and receive a message. This does clearly not capture modern network cards, which provide full duplex connectivity. Round-based models [30] are in that sense more convenient as they assume that a process can send a message to one or more processes at the start of each round, and can receive the messages sent by other processes at the end of the round. Whereas this model is well-suited for proving lower bounds on the latency of protocols, it is however not well suited for making realistic predictions about the throughput. In particular, it is not realistic to consider that several messages can be simultaneously received by the same process.

In this paper, we propose to analyze protocols using a slightly modified version of the round-based model. More specifically, we define rounds as follows: in each round r, every process p_i is supposed to: (1) compute the message for round r, $m(i, r)$, (2) unicast (or best effort broadcasts) $m(i, r)$ and (3) receive a single message sent at round r unless the sending process has crashed.

4 Protocol

Our FSR protocol guarantees uniform total order message delivery despite the failure of t processes with $t < n$, where n is the total number of processes in the system. The performance of FSR is optimized for failure free periods.

More specifically, the performance of FSR was designed for high throughput in various kinds of high-load traffic scenarios. These scenarios include a single process TO-broadcasting, several processes TO-broadcasting a steady stream of messages at the same time, several processes TO-broadcasting bursts of messages simultaneously and all processes TO-broadcasting a steady stream of messages. Not only does FSR provide the same throughput in all these cases, it also provides the same reasonable latency to all processes. Interestingly, fairness is inherently part of the protocol such that if several processes want to TO-broadcast messages at the same time, then they will TO-broadcast the same number of messages during a given time-frame. FSR does not enforce a trade off between performance and fairness.

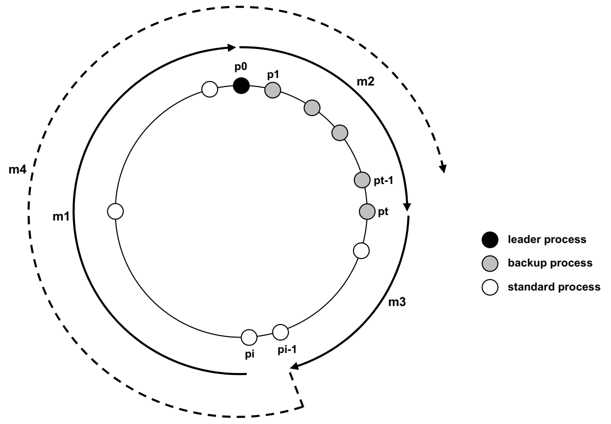

Figure 4. FSR protocol illustration.

4.1 Overview

In short, the idea underlying FSR is to combine a fixed sequencer for ordering, with a ring topology for dissemination. The main advantage of the ring topology is that it is simple to implement and at the same time provides high throughput. However, the ring is not only used for message dissemination but also for sequencing. Contrary to traditional fixed sequencer protocols, processes do not send messages directly to the sequencer but only to their direct successors.

All messages circulate clockwise in the same direction. Even though there is only a single fixed sequencer, this process is not a bottleneck since it only needs to append a small sequence number to the message and then forward it: the sequencer receives and sends the same number of messages as all other processes. The sequencer is followed in the ring by t *backup* processes which have the role of keeping a copy of all messages and sequence numbers that have not yet been delivered by all processes.

The FSR protocol is illustrated in Figure 4. Two cases are interesting to highlight:

1. The case of a standard process broadcasting a message illustrated in Figure 4. When a process p_i TO-broadcasts a message m, p_i forwards m to its successor p_{i+1} (message m_1), which in turn forwards m to its successor and so on until the message reaches the leader p_0. As in any sequencer based protocol, the leader assigns monotonically increasing sequence numbers to messages, therefore imposing a total order on their delivery. The message and sequence number pair (m_2) is then forwarded by the leader until it has reached t backup processes (process p_t). The leader

and backup processes do not yet TO-deliver the message (except for the last backup p_t). From process p_t the message with sequence number m_3 is forwarded until process p_{i-1}. Processes p_t to p_{i-1} TO-deliver m upon receiving m_3. Process p_{i-1} then sends an acknowledgment m_4 which is forwarded until process p_{t-1}. All processes can TO-deliver m upon receiving m_4.

2. When a backup process p_b ($0 < b \leq t$) TO-broadcasts a message m, it is forwarded until the leader p_0 (this first message is obviously omitted if the leader initiates the TO-broadcast). The message and sequence pair is forwarded until process p_{b-1}. From there on an ack is circulated until process p_t. Contrary to the previous case, none of the backup processes can yet TO-deliver m. Only when processes receive the ack sent from p_t can they TO-deliver m.

There are several tricky issues that need to be handled in order for the protocol to be efficient and fair. Although in the protocol described above a message goes around the ring more than once, in order to guarantee high throughput, the actual message to be TO-broadcast only goes around *once*. The rest of the generated messages only contain an identifier. Since these messages are small they can be piggy-backed on other messages when the load is high. However when the load is low these messages are not piggy-backed in order to keep a low latency. Also, because of the ring dissemination topology, uniform message size is necessary in order to avoid that large messages stall the smaller messages. This can be achieved by segmenting large messages into several smaller ones.

Ensuring fairness means that if more than one process TO-broadcasts messages then each process should be able to broadcast the same number of messages during the same amount of time. By carefully deciding when a process can start a new TO-broadcast, it is possible to provide this fairness.

4.2 Protocol Details

Our FSR protocol is built on top of a group communication system which provides virtually synchronous communications (VSC) [6]. According to the virtual synchrony programming model, processes are organized into groups. Processes can join and leave the group using the appropriate primitives. Faulty processes are excluded from the group after crashing. Upon a membership change, processes agree on a new view by using a view change protocol.

4.2.1 Group Membership Changes

When a process joins or leaves the group, a $view\_change$ event is generated by the VSC layer and the current view v_r is replaced the new view v_{r+1}. This can happen when a process crashes or when a process actively wants to leave or join the group. As soon as a new view is installed it becomes the basis for the new ring topology. There are several cases to consider when a $view\_change$ event occurs. When v_{r+1} is installed, the processes execute the following procedures depending on their role in v_{r+1}:

- All processes TO-broadcast any message in view v_{r+1} that they have TO-broadcast in the view v_r but not yet TO-delivered in v_r.
- The new leader (in v_{r+1}) must resend the following messages: (1) all message and sequence number pairs that have not yet been TO-delivered, (2) an ack of the latest TO-delivered message.

4.2.2 Optimizations

The acknowledgment messages sent within FSR are very small messages that just contain an identifier of the message that they acknowledge. Consequently, these messages can be piggy-backed on normal messages sent by other TO-broadcasts. When all acks are piggy-backed, each TO-broadcast effectively only sends each message around the ring once, thus enabling FSR to achieve high throughput.

4.2.3 Fairness

Fairness captures the very fact that each process has an equal opportunity of having its messages eventually TO-delivered by all processes. Intuitively, the notion of fairness means that no single process has priority over other processes when broadcasting messages. For instance, when two processes TO-broadcast large numbers of messages, then each process should have approximately the same number of messages TO-delivered by all processes.

Fixed sequencer protocols surveyed in Section 2 are inherently fair: each process that TO-broadcasts a message sends it directly to the sequencer which will handle incoming messages on a first come, first served basis. If a lot of messages arrive at the sequencer at the same time then it will serve them in a round-robin fashion. In our FSR protocol, messages to be TO-broadcast are not sent directly to the sequencer, but rather forwarded to the successor. If all processes want to TO-broadcast messages, then at each round a process can either start a new TO-broadcast by sending a message to its successor, or forward messages from its predecessor.

Ensuring fairness in FSR is achieved by having a specific mechanism to decide whether a process can initiate a new broadcast or whether it must first forward messages stored in its incoming buffer. Intuitively, each process maintains a list *forward* of the processes for which it has forwarded

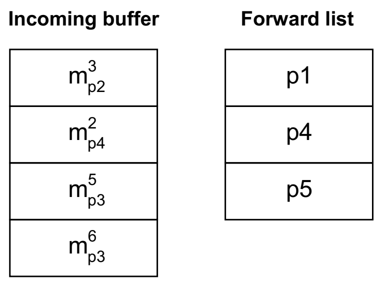

Figure 5. Incoming buffer and forward list of a process initiating a TO-broadcast.

messages since its last broadcast. When a process initiates a TO-broadcast, it first forwards messages that are in its incoming buffer and that have been sent by processes not in the *forward* list. Figure 5 illustrates the incoming buffer and *forward* list of a process p_i wishing to initiate a TO-broadcast. Before sending its own message m, process p_i forwards messages m^3_{p2} and m^5_{p3}. This simple mechanism ensures that no process will prevent others from TO-broadcasting their own messages.

4.3 Analytical Performance

4.3.1 Latency

The latency of TO-broadcasting is defined as the largest number of rounds that are necessary from the initial TO-broadcast of a single message m until the round where the last process TO-delivers m. The latency is measured in a newly initialized system when no other messages are TO-broadcast: it is obvious that latency increases when a lot of messages are TO-broadcast simultaneously. The latency of FSR can be expressed as follows for all processes: $L(i) = 2n + t - i - 1$, where i is the position of the TO-broadcasting process in the ring with respect to the leader at position 0. We can observe the following:

- The latency is linear with respect to the number of processes n, implying that FSR scales well.
- The latency is also linear with respect to the number of tolerated failures t.
- The position of the TO-broadcasting process in the ring has an influence on the latency. In order to evenly distribute the latency for all processes, the role of the leader can be periodically moved to the next process in the ring. This can be done by periodically executing a *leave* followed by a *join* at the current leader process. It is also possible to transfer the role of the leader without having the leader *leave* and *join* the group, but for space reasons that discussion is left out of this paper.

4.3.2 Throughput

The throughput of FSR is at least equal to one. This means that on average at least one TO-broadcast is completed during each round (a complete TO-broadcast of message m meaning that all processes TO-delivered m). In more detail:

- The throughput is independent from the number of processes that TO-broadcast at the same time. If only one process continuously TO-broadcasts it is obvious that it can TO-broadcast a new message every round since every broadcast message goes round the ring only once. After an initial latency of $2n + t - i - 1$ rounds the first message has been TO-delivered by all processes and in the consecutive rounds one message is TO-delivered every round. With multiple senders the same argument holds. Because of the fairness described in Section 4.2, each round a message TO-broadcast by a different process is TO-delivered.
- The throughput of FSR is independent of the number n of processes in the system.
- The throughput of FSR is independent of the number t of processes that can crash.

5 Performance

This section describes the various experiments that we conducted to evaluate the performance of FSR. We implemented FSR using DREAM [28], a Java-based component library dedicated to the construction of communication middleware. Dream enables the development of various forms of message-oriented middleware (e.g. publish/subscribe, event/reaction and group communication protocols) by component assembly. The library contains a wide array of components, including message queues, channels (socket wrappers), routers, etc.

5.1 Benchmark Description

We ran benchmarks on a cluster of machines with dual 900MHz Itanium-2 processors, 3GB of RAM and a Fast Ethernet adapter, running Linux kernel 2.4.21. The raw latency and bandwidth over IP between two machines were measured with Netperf [33] and displayed in Table 1.

The benchmarks test k-to-n TO-broadcasts, k ranging from 1 to n. All processes know a priori the number of messages they expect from other processes (each sender sends the same number of messages). A barrier is used to synchronize the experiment start-up. Each process takes a local timestamp and starts sending its messages. When the last expected message from a sender is received, an acknowledgment is sent back to the sender. This allows stopping the

Protocol	Bandwidth
TCP	94 Mb/s
UDP	93 Mb/s

Table 1. Raw network performance measured using Netperf.

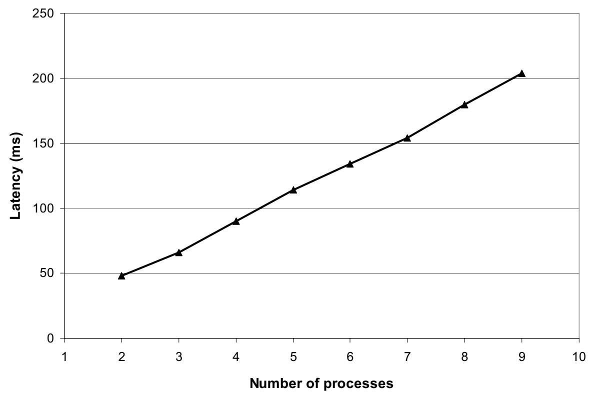

Figure 6. Latency as a function of the number of processes.

timer at each sender. Then, each sender calculates the time between the first broadcast message sent and the acknowledgment message received by the last process receiving all the senders' messages. For each sender, we calculate the throughput by using the time between the start message and the acknowledgment of the last message. We ensure that the acknowledgment latency is negligible compared to the overall experience time. We also perform the same experiment but with only one sender and one message. Repeating this experiment several times gives us the average latency in the contention-free case.

5.2 Latency Evaluation

Figure 6 plots the latency without contention as a function of the number of processes. The experiments consisted in n-to-n TO-broadcasts of 100KB messages. The represented latency is the average of the latencies observed at each sender. The graph shows that the latency is linear with respect to the number of processes, which confirms the theoretical analysis.

Figure 7 plots the latency as a function of the throughput. The experiments consisted in n-to-n TO-broadcasts of 100KB messages between 5 processes. The results were obtained by throttling the senders to a given sending rate and reporting the corresponding average latency and throughput. This graph shows that the latency is almost constant until the maximum throughput is reached. Then, unprocessed messages are stored in local queues at each process, which explains the important increase of the observed latency.

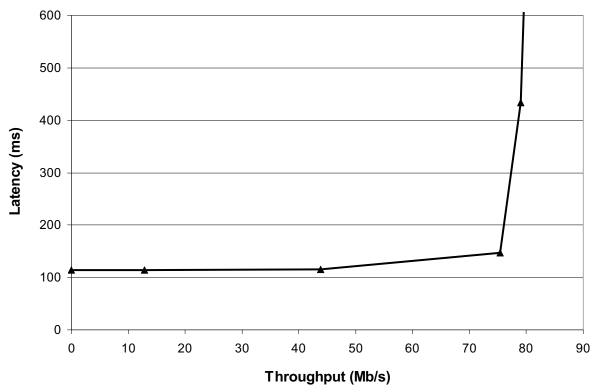

Figure 7. Latency as a function of the throughput.

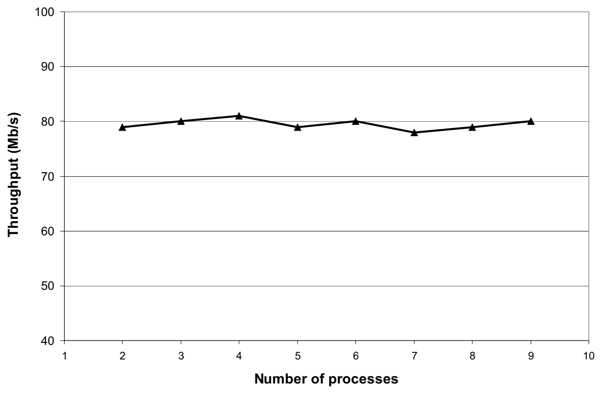

Figure 8. Throughput as a function of the number of processes.

5.3 Throughput Evaluation

Figure 8 plots the maximum throughput as a function of the number of processes. The experiments consisted in n-to-n TO-broadcasts of 100KB messages. The graph shows that FSR achieves a throughput of 79 Mbit/s on a 100 Mbit/s switched Ethernet network. Moreover, it shows that the achieved throughput is independent of the number of processes in the ring, which confirms our analysis.

The last experiment consisted in varying the number of senders in the ring. The experiment consisted in k-to-5 TO-broadcasts (k ranging from 1 to 5) of 100KB messages. The graph shows that the performance of FSR does not depend on k. This means that FSR reaches the maximal throughput, whichever the number of sender is.

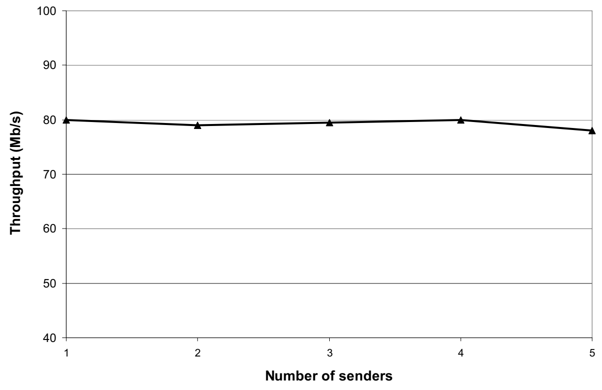

Figure 9. Throughput as a function of the number of senders.

6 Summary

This paper presents FSR, a uniform total order broadcast protocol that can be used at the main communication block of a replication scheme to achieve software-based fault-tolerance.

FSR is the first uniform total order broadcast protocol that consistently provides high throughput whether one or several processes continuously TO-broadcast messages. In short, high throughput captures the ability to deliver the largest possible number of messages broadcast, regardless of message broadcast patterns. This notion is precisely defined in a round-based model of computation which captures message passing interaction patterns over clusters of homogeneous machines interconnected by fully switched LANs. We believe that the model is interesting in its own right and can be used to evaluate the performance of other protocols.

FSR is based on a ring topology, only relies on point-to-point inter-process communication, and has linear latency with respect to the number of processes. FSR is also fair in the sense that each process has an equal opportunity of having its messages delivered by all processes.

References

[1] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.

[2] E. Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, Washington, DC, USA, 1997. IEEE Computer Society.

[3] S. Armstrong, A. Freier, and K. Marzullo. Multicast transport protocol. RFC 1301, IETF, 1992.

[4] R. Baldoni, S. Cimmino, and C. Marchetti. A Classification of Total Order Specifications and its Application to Fixed Sequencer-based Implementations. *to appear in Journal of Parallel and Distributed Computing*, June 2006.

[5] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. *Mathematical Systems Theory*, 27(5):431–452, 1994.

[6] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP'87)*, pages 123–138, New York, NY, USA, 1987. ACM Press.

[7] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[8] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1993.

[9] R. Carr. The tandem global update protocol. *Tandem Syst. Rev. 1*, pages 74–85, jun 1985.

[10] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[11] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[12] J.-M. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, 1984.

[13] F. Cristian. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, 1991.

[14] F. Cristian, S. Mishra, and G. Alvarez. High-performance asynchronous atomic broadcast. *Distrib. Syst. Eng. J.*, 4(2):109–128, jun 1997.

[15] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[16] X. Défago, A. Schiper, and P. Urbán. Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. on Information and Systems*, E86-D(12):2698–2709, 2003.

[17] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[18] R. Ekwall, A. Schiper, and P. Urban. Token-based atomic broadcast using unreliable failure detectors. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*, pages 52–65, Washington, DC, USA, 2004. IEEE Computer Society.

[19] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: a fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, Washington, DC, USA, 1995. IEEE Computer Society.

[20] T. Friedman and R. V. Renesse. Packing messages as a tool for boosting the performance of total ordering protocls. In *Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC '97)*, Washington, DC, USA, 1997. IEEE Computer Society.

[21] U. Fritzke, P. Ingels, A. Mostefaoui, and M. Raynal. Consensus-based fault-tolerant total order multicast. *IEEE Trans. Parallel Distrib. Syst.*, 12(2):147–156, 2001.

[22] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Trans. Comput. Syst.*, 9(3):242–271, 1991.

[23] A. Gopal and S. Toueg. Reliable broadcast in synchronous and asynchronous environments (preliminary version). In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, pages 110–123, London, UK, 1989. Springer-Verlag.

[24] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[25] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. *Distributed systems (2nd Ed.)*, pages 97–145, 1993.

[26] F. Kaashoek and A. Tanenbaum. An evaluation of the amoeba group communication system. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, Washington, DC, USA, 1996. IEEE Computer Society.

[27] J. Kim and C. Kim. A total ordering protocol using a dynamic token-passing scheme. *Distrib. Syst. Eng. J.*, 4(2):87–95, jun 1997.

[28] M. Leclercq, V. Quéma, and J.-B. Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs. *IEEE Distributed Systems Online*, 6(9), September 2005.

[29] S. Luan and V. Gligor. A fault-tolerant protocol for atomic broadcast. *IEEE Trans. Parallel Distrib. Syst.*, 1(3):271–285, 1990.

[30] N. A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.

[31] L. Malhis, W. Sanders, and R. Schlichting. Numerical performability evaluation of a group multicast protocol. *Distrib. Syst. Eng. J.*, 3(1):39–52, march 1996.

[32] L. Moser, P. Melliar-Smith, and V. Agrawala. Asynchronous fault-tolerant total ordering algorithms. *SIAM J. Comput.*, 22(4):727–750, 1993.

[33] Netperf. http://www.netperf.org/.

[34] T. Ng. Ordered broadcasts for large applications. In *Proceedings of the 10th IEEE International Symposium on Reliable Distributed Systems (SRDS'91)*, pages 188–197, Pisa, Italy, 1991. IEEE Computer Society.

[35] L. Peterson, N. Buchholz, and R. Schlichting. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, 1989.

[36] L. Rodrigues, H. Fonseca, and P. Verissimo. Totally ordered multicast in large-scale systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, Washington, DC, USA, 1996. IEEE Computer Society.

[37] Sun. *Java 2 Platform, Enterprise Edition (J2EE)*. http://java.sun.com/j2ee/.

[38] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proceedings of 9th IEEE International Conference on Computer Communications and Networks (IC3N 2000)*, pages 582–589, 2000.

[39] P. Vicente and L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, Washington, DC, USA, 2002. IEEE Computer Society.

[40] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, pages 33–57, London, UK, 1994. Springer-Verlag.

[41] U. Wilhelm and A. Schiper. A hierarchy of totally ordered multicasts. In *Proceedings of the 14TH Symposium on Reliable Distributed Systems*, Washington, DC, USA, 1995. IEEE Computer Society.