Throughput Optimal Total Order Broadcast for Cluster Environments[1]

RACHID GUERRAOUI, RON R. LEVY, BASTIAN POCHON
School of Computer and Communication Sciences, EPFL
and
VIVIEN QUÉMA
Centre National de la Recherche Scientifique

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

We present LCR, the first throughput optimal uniform total order broadcast protocol. LCR is based on a ring topology. It only relies on point-to-point inter-process communication and has a linear latency with respect to the number of processes. LCR is also fair in the sense that each process has an equal opportunity of having its messages delivered by all processes.

We benchmark a C implementation of LCR against Spread and JGroups, two of the most widely used group communication packages. LCR provides higher throughput than the alternatives, over a large number of scenarios.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Network communications*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.2.4 [**Computer-Communication Networks**]: Local and Wide-Area Networks—*Network operating systems*; D.4.4 [**Operating Systems**]: Communications Management—*Network communication*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*

General Terms: Algorithms, Performance, Reliability

Additional Key Words and Phrases: software fault-tolerance, replication, total order broadcast, cluster computing

1. INTRODUCTION

1.1 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. On the other hand, because of rapidly decreasing hardware costs, ensuring fault tolerance through state machine replication [Schneider 1990] is gaining in popularity. Roughly speaking, state machine replication is about maintaining several copies of the same software object on different machines (also called replicas or processes), such that, if one or more replicas fail, enough replicas remain to guarantee accessibility to the object. The key to making this technique work is a well designed software layer that hides all the difficulties behind maintaining replica consistency from the application developer

[1]This paper is an extended version of [Guerraoui et al. 2006].

and renders it transparent to the clients.

In a replicated database for instance [Cecchet et al. 2004], all processes perform the same write queries (INSERT and UPDATE) in the same order. Read queries (SELECT) do not change the state of the replicated database and do not have to be performed by all replicas. It is crucial to guarantee however uniformity, that is, replicas should not execute any write query before making sure that all other replicas will also execute it. To illustrate this, consider the simple case where a replica executes a write request w_1, subsequently answers a client's read request r_1 and fails. If the other replicas do not execute w_1, the value of r_1 returned to the client will not be consistent with the replicated database. The ordering mechanism should ensure that all replicas perform the same operations on their copy in the same order, even if they subsequently fail. This mechanism is encapsulated by a communication abstraction called *uniform total order broadcast* (UTO-broadcast) [Hadzilacos and Toueg 1993]. This abstraction ensures the following for all messages that are broadcast: (1) *Uniform agreement*: if a replica delivers a message m, then all correct replicas eventually deliver m; (2) *Strong uniform total order*: if some replica delivers some message m before message m', then a replica delivers m' only after it has delivered m.

Clearly, even though UTO-broadcast is very useful, not all applications need the strong guarantees that it provides. Some applications might only need reliable or even best-effort broadcast. We will show however that there are no weaker broadcast protocols that can obtain higher throughput than the protocol described in this paper. In a sense, the strong uniform total order guarantees provided by our protocol are free.

1.2 Latency vs. Throughput

Historically, most total order protocols have been devised for low broadcast latency [Kaashoek and Tanenbaum 1996; Armstrong et al. 1992; Carr 1985; Garcia-Molina and Spauster 1991; Birman and van Renesse 1993; Wilhelm and Schiper 1995]. Latency usually measures the time required to complete a single message broadcast without contention. (It is typically captured by a number of rounds in the classical model of [Lynch 1996]: in that model, at the start of each round a process can send a message to one or more processes. It receives the messages sent by other processes at the end of the round.) On the other hand, very few protocols have been designed for high throughput. Throughput measures the number of broadcasts that the processes can complete per time unit. In some high load environments, e.g. database replication for e-commerce, throughput is often as important, if not more, than latency. Indeed, under high load, the time spent by a message in a queue before being actually disseminated can grow indefinitely. A high throughput broadcast protocol reduces this waiting time.

Maybe not surprisingly, protocols that achieve low latency often fail to provide high throughput. To illustrate this, consider the example depicted in Figure 1:

(1) In algorithm $\mathcal{A}$, process p_1 first sends a message to p_2. In the next step p_2 forwards the message to p_4 and at the same time p_1 sends the message to p_3.

(2) In algorithm $\mathcal{B}$, process p_1 first sends a message to p_2. In the next step p_2 forwards the message to p_3 and finally p_3 forwards the message to p_4.

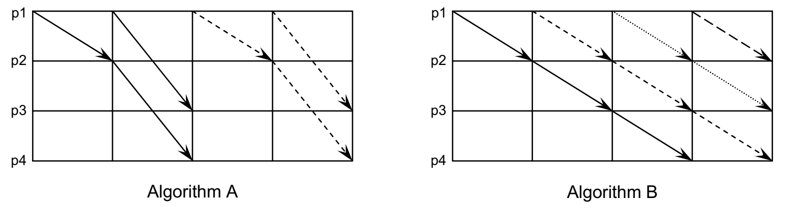


Fig. 1. Latency vs. Throughput: comparison of two broadcast algorithms A and B. Process p_1 initiates the broadcast. Broadcast latency is 2 rounds for A and 3 rounds for B. However, B has higher throughput than A: in algorithm B one broadcast is completed every round, while in algorithm A only one broadcast is completed every 2 rounds.

Algorithm $\mathcal{A}$ has a latency of 2 whereas algorithm $\mathcal{B}$ a latency of 3. Latency wise, $\mathcal{A}$ is better than $\mathcal{B}$. If we look at the throughput however, we see that $\mathcal{A}$ can only start a new broadcast every 2 time units, while $\mathcal{B}$ can broadcast a new message every time unit. Therefore even though the latency of $\mathcal{B}$ is higher than that of $\mathcal{A}$, the maximal throughput of $\mathcal{B}$ is twice that of $\mathcal{A}$.

1.3 Contributions

In this paper we present LCR, a uniform total order broadcast protocol that is throughput optimal in failure-free periods. LCR relies on point-to-point communication channels between processes. It uses logical clocks and a ring topology (hence the name). The ring topology ensures that each process always sends messages to the same process, thus avoiding any possible collisions. To eliminate bottlenecks, messages in LCR are sequenced using logical vector clocks instead of a dedicated sequencer. Furthermore, the ring topology allows all acknowledgement messages to be piggy-backed, reducing the message overhead. These two characteristics ensure throughput optimality and fairness, regardless of the type of traffic. In our context, fairness conveys the equal opportunity of processes to have their broadcast messages delivered.

We give a full analysis of LCR's performance and fairness. We also report on performance results based on a C implementation of LCR that relies on TCP channels. The implementations are benchmarked against Spread and JGroups on a cluster of 9 machines and we show that LCR consistently delivers the highest throughput. For instance, with 4 machines, LCR achieves throughput of up to 50% higher than that of Spread and up to 28% higher than that of JGroups.

1.4 Roadmap

Section 2 introduces the relevant system and performance models. Section 3 gives an overview of the related work. We describe LCR in Section 4, then we give an analytical evaluation of it in Section 5, and we report on our experimental evaluation in Section 6. Section 7 concludes the paper with some final remarks.

2. MODEL

2.1 System Model

Our context is a small cluster of homogeneous machines interconnected by a local area network. In our protocol, each of the n machines (or processes) creates a

TCP connection to only a single process and maintains this connection during the entire execution of the protocol (unless the process fails). Because of the simple communication pattern, the homogeneous environment, and low local area network latency, it is reasonable to assume that, when a TCP connection fails, the server on the other side of the connection failed [Dunagan et al. 2004]. We thus directly implement the abstraction of a *perfect* failure detector (P) [Chandra and Toueg 1996a] to which each process has access.

2.2 Performance Model

Analyzing the performance of a communication abstraction requires a precise distributed system model. Some models only address point-to-point networks, where no native broadcast primitive is available [Culler et al. 1993; Bar-Noy and Kipnis 1994]. The model of [Urbán et al. 2000], which was recently proposed to evaluate total order broadcast protocols, assumes that a process cannot simultaneously send and receive a message. This does clearly not capture modern network cards for these provide full duplex connectivity. Round-based models [Lynch 1996] 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 appropriate 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 analyze protocols using the model used in [Guerraoui et al. 2006; Guerraoui et al. 2007]. This model assumes that processes can send one message to one (unicast) or more processes (multicast) at the start of each round, but can only receive a single message sent by other processes at the end of the round. If more than one message is sent to the same process in the same round, these messages will be received in different rounds. The rationale behind this model is that machines in a cluster are each connected to a switch via a twisted pair ethernet cable. As modern network cards are full-duplex, each machine can simultaneously send and receive messages. Moreover, as the same physical cable is used to send and receive messages, it makes sense to make the very same assumption on the number of messages that can be received and on the number of messages that can be sent in one round. The model we use can thus be described as follows: in each round k, every process p_i can execute the following steps:

(1) p_i computes the message for round k, $m(i,k)$,
(2) p_i sends $m(i,k)$ to all or a subset of processes,
(3) p_i receives at most one message sent at round k.

In a sense, our model is similar to the Postal model [Bar-Noy and Kipnis 1997] with the addition of a multicast primitive which is available on all current local area networks.

2.3 Throughput

Basically, throughput captures the average number of completed broadcasts per round. A complete broadcast of message m means that all processes delivered m.

In this model, a broadcast protocol is optimal if it achieves an average of one complete broadcast per round regardless of the number of broadcasters. Considering a cluster with n processes, we seek for an optimal throughput with k simultaneous broadcasters, k ranging from 1 to n. When evaluating throughput, we assume a variable number k of processes continuously sending messages, where $1 \leq k \leq n$. Using this assumption, we can accurately model bursty broadcast patterns. In the general case with random broadcast patterns, we can observe the following: as soon as the load on the broadcast system approaches the maximum throughput (the case we are interested in), processes will not be able to broadcast new messages immediately. This will result in the creation of a queue of messages at the sender which leads to sending a burst. Thus our model can accurately represent the general case in high load scenarios.

3. RELATED WORK

The five following classes of UTO-broadcast protocols have been distinguished in the literature [Défago et al. 2004]: fixed-sequencer, moving sequencer, privilege-based, communication history, and destination agreement. In this section, we only survey time-free protocols, i.e. protocols that do not rely on physical time, since these are the ones comparable to the LCR protocol. The reason is that the assumption of synchronized clocks is not very realistic in practice, especially since clock skew is hard to detect. We also do not discuss protocols with disk writes as in [van Renesse and Schneider 2004] for instance. Our goal (and that of most of the related work surveyed here) is to optimize the broadcasting of messages at the network level.

3.1 Fixed Sequencer

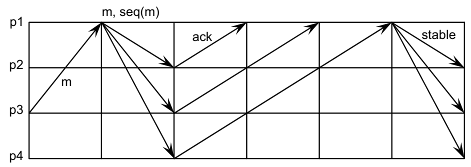


Fig. 2. Fixed sequencer-based UTO-broadcast.

In a fixed sequencer protocol [Kaashoek and Tanenbaum 1996; Armstrong et al. 1992; Carr 1985; Garcia-Molina and Spauster 1991; Birman and van Renesse 1993; Wilhelm and Schiper 1995], a single process is elected as the sequencer and is responsible for the ordering of messages (Figure 2). The sequencer is unique, and another process is elected as a new sequencer only in the case of sequencer failure. Three variants of the fixed sequencer protocol exist [Baldoni et al. 2006], each using a different communication pattern. Fixed sequencer protocols exhibit linear latency with respect to n [Défago et al. 2003], but poor throughput. The sequencer becomes a bottleneck because it must receive the acknowledgments (acks) from all processes[2] and must also receive all messages to be broadcast. Note that this class

[2]Acknowledgments in the fixed sequencer can only be piggy-backed when all processes broadcast messages all the time [Défago et al. 2003].

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.

3.2 Moving Sequencer

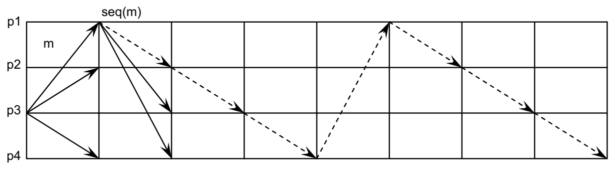


Fig. 3. Moving sequencer-based UTO-broadcast.

Moving sequencer protocols [Chang and Maxemchuk 1984; Whetten et al. 1994; Kim and Kim 1997; Cristian et al. 1997] (Figure 3) 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 in failure-free situations). This is achieved by a token which 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, it sends it to all other processes. Upon receiving m, 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 m 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 [Défago et al. 2004]. On the other hand, these protocols achieve better throughput, although not optimal. Figure 3 depicts 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. Thus, the protocol cannot achieve optimal throughput. Even in the n-to-n case, optimal throughput cannot be achieved because at any given time there is only one process which can send messages. Thus, the throughput when all processes broadcast cannot be higher than when only one process broadcasts (in Section 5.1 we will show that optimal throughput can only be achieved when all processes broadcast). Note that fixed sequencer protocols are often prefered to moving sequencer protocols because they are much simpler to implement [Défago et al. 2004].

3.3 Privilege-based Protocols

These protocols [Friedman and Renesse 1997a; Cristian 1991; Ekwall et al. 2004; Amir et al. 1995; Gopal and Toueg 1989] rely on the idea that senders can broadcast messages only when they are granted the privilege to do so (Figure 4). 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. As with moving

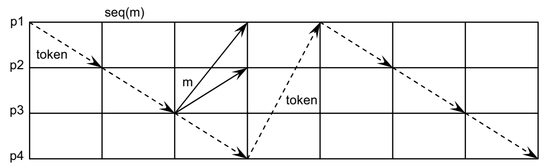


Fig. 4. Privilege-based UTO-broadcast.

sequencer protocols, the throughput when all processes broadcast cannot be higher than when only one process broadcasts.

3.4 Communication History-based Protocols

As in privilege-based protocols, communication history-based protocols [Peterson et al. 1989; Malhis et al. 1996; Ezhilchelvan et al. 1995; Ng 1991; Moser et al. 1993] use sender-based ordering of messages. Nevertheless, they differ by the fact that processes can send messages at any time. Messages carry logical clocks that allow processes to observe the messages received by other processes in order to learn when 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 that is broadcast.

3.5 Destination Agreement

In destination agreement protocols, the delivery order results from an agreement between destination processes. Many such protocols have been proposed [Chandra and Toueg 1996b; Birman and Joseph 1987b; Luan and Gligor 1990; Fritzke et al. 2001; Anceaume 1997]. 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.

Note that hybrid protocols, combining two different ordering mechanisms have also been proposed [Ezhilchelvan et al. 1995; Rodrigues et al. 1996; Vicente and Rodrigues 2002]. Most of these protocols are optimized for large scale networks instead of clusters, making use of multiple groups or optimistic strategies.

4. THE LCR PROTOCOL

LCR combines (a) a ring topology for high-throughput dissemination with (b) logical (vector) clocks for message ordering. It is a uniform total order broadcast (UTO-broadcast) protocol exporting two primitives, utoBroadcast and utoDeliver, and ensuring the following four properties:

—**Validity**: if a correct process p_i utoBroadcasts a message m, then p_i eventually utoDelivers m.

—**Integrity**: for any message m, any correct process p_j utoDelivers m at most once, and only if m was previously utoBroadcast by some correct process p_i.

—**Uniform Agreement**: if any process p_i utoDelivers any message m, then every correct process p_j eventually utoDelivers m.

—**Total Order**: for any two messages m and m', if any process p_i utoDelivers m without having delivered m', then no process p_j utoDelivers m' before m.

We detail our LCR protocol in the following. We assume a set $\Pi = \{p_0, \cdots, p_{n-1}\}$ of n processes. Processes are organized in a ring: every process has a predecessor and a successor in the ring: p_0 is before p_1, which is before p_2, etc. We call p_0 "the *first* process in the ring", and p_{n-1} "the *last* process in the ring". We first describe how we totally order messages in LCR. We then describe the behavior of the protocol in the absence of failures and then we describe what happens when there is a group membership change, e.g. a node failing or joining/leaving the system.

4.1 Total Order Definition

As we explain later in this section, to broadcast a message, a process sends it to its successor, which itself send its its successor, and so on until every process received the message. We define the total order on messages in LCR as the order according to which messages are received by the last process in the ring, i.e. process p_{n-1}. To illustrate this, consider any two messages m_i and m_j broadcast by processes p_i and p_j, respectively. Assume $i < j$, i.e. p_i is "before" p_j in the ring. In order to determine whether m_i is ordered before m_j, it is enough to know whether p_j had received m_i before sending m_j. This is achieved using vector clocks: process p_j is equipped with a logical vector clock $\mathcal{C}_{p_j} = (c_k)_{k=[0..n-1]}$. At any time, the value $\mathcal{C}_{p_j}[i]$ represents the number of broadcasts initiated by process p_i that p_j received so far. Before initiating the broadcast of message m_i, process p_i increments the number of broadcasts it initiated (stored in $\mathcal{C}_{p_i}[i]$) and timestamps m_i with its clock. Upon reception of message m_i, process p_j updates its logical clock to take into account message m_i: it increments the value stored in $\mathcal{C}_{p_j}[i]$. If later p_j sends message m_j, it will timestamp m_j with its logical clock that reflects the fact that it received m_i before sending m_j. Every process can thus order m_i and m_j by comparing their clocks. More precisely, let us note $\mathcal{C}_{m_i}$ (resp. $\mathcal{C}_{m_j}$) the logical clock carried by m_i (resp. m_j). Message m_i is ordered before m_j, noted $m_i \prec m_j$, if and only if $\mathcal{C}_{m_i}[i] \leq \mathcal{C}_{m_j}[i]$ when $i < j$, and $\mathcal{C}_{m_i}[i] < \mathcal{C}_{m_j}[i]$ when if $i = j$.

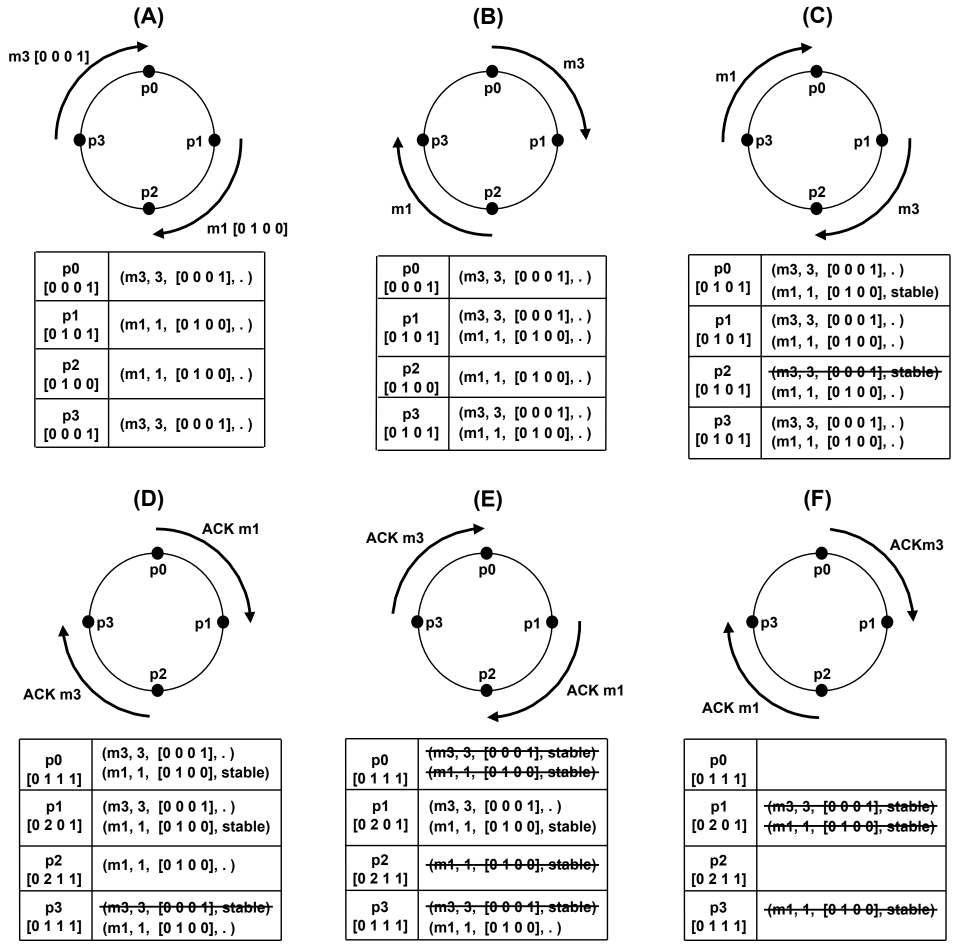
4.2 Failure-free Behavior

The pseudo-code of the LCR sub-protocol executed in the absence of failures is depicted in Figure 5. To broadcast a message m_i, process p_i sends m_i to its successor in the ring. The message is then successively forwarded until it reaches the predecessor of p_i. Processes forward messages in the order in which they receive them. To ensure *total order delivery*, each process ensures, before delivering message m_i, that it will not subsequently receive a message that is ordered before m_i (according to the order defined in the previous section). Moreover, for the sake of *uniform delivery*, each process ensures, before delivering m_i, that all other processes already received m_i (m_i is stable). These guarantees rely on a local list, denoted pending, used by every process to store messages before they are delivered. Messages in pending are totally ordered according to the order defined above. We now explain when messages stored in pending are delivered.

When the predecessor of p_i receives message m_i, it sends an ACK message along the ring. The ACK message is then successively forwarded until it reaches the predecessor of the process that sent it. Upon receiving this ACK message, each process knows (1) that m_i is stable, and (2) that it already received all messages that are ordered before m_i. The first point is obvious. The second point can be intuitively explained as follows. Consider a message m_j that is ordered before m_i. We know, by definition of the total order on messages, that process p_{n-1} will receive m_j before m_i. Provided that messages are forwarded around the ring in the order in which they are received, we know that the predecessor of p_i will receive m_j before sending the ACK for m_i. Consequently, no process can receive the ACK for m_i before m_j. Once a message has been set to stable, it can be delivered as soon as it becomes first in the pending list.

```
Procedures executed by any process $p_i$
1:  procedure initialize($initial\_view$)
2:    $pending_i \leftarrow \emptyset$                                   {pending list}
3:    $\mathcal{C}[1 \ldots n] \leftarrow \{0, \ldots, 0\}$                 {local vector clock}
4:    $view \leftarrow initial\_view$

5:  procedure utoBroadcast($m$)
6:    $\mathcal{C}[i] \leftarrow \mathcal{C}[i] + 1$
7:    $pending \leftarrow pending \cup [m, p_i, \mathcal{C}, \perp]$
8:    Rsend $\langle m, p_i, \mathcal{C} \rangle$ to $successor(p_i, view)$          {broadcast a message}

9:  upon Rreceive $\langle m, p_j, \mathcal{C}_m \rangle$ do
10:   if $\mathcal{C}_m[j] > \mathcal{C}[j]$ then
11:     if $p_i \neq predecessor(p_j, view)$ then
12:       Rsend $\langle m, p_j, \mathcal{C}_m \rangle$ to $successor(p_i, view)$     {forward the message}
13:       $pending \leftarrow pending \cup [m, p_j, \mathcal{C}_m, \perp]$
14:     else
15:       $pending \leftarrow pending \cup [m, p_j, \mathcal{C}_m, \mathsf{stable}]$           {m is stable}
16:       Rsend $\langle \text{Ack}, p_j, \mathcal{C}_m \rangle$ to $successor(p_i, view)$        {send an Ack}
17:       tryDeliver()
18:     $\mathcal{C}[j] \leftarrow \mathcal{C}[j] + 1$                        {update local vector clock}

19: upon Rreceive $\langle \text{Ack}, p_j, \mathcal{C}_m \rangle$ do
20:   if $p_i \neq predecessor(predecessor(p_j), view)$ then
21:     $pending[\mathcal{C}_m] \leftarrow [*, *, *, \mathsf{stable}]$                 {m is stable}
22:     Rsend $\langle \text{Ack}, p_j, \mathcal{C}_m \rangle$ to $successor(p_i, view)$        {forward the Ack}
23:     tryDeliver()

24: procedure tryDeliver()
25:   while $pending.first = [m, p_k, \mathcal{C}_m, \mathsf{stable}]$ do
26:     utoDeliver($m$)                                         {deliver a message}
27:     $pending \leftarrow pending - [m, p_k, \mathcal{C}_m, \mathsf{stable}]$
```

Fig. 5. Pseudo-code of the LCR protocol.



(A)

p0 [0 0 0 1]	(m3, 3, [0 0 0 1], .)
p1 [0 1 0 1]	(m1, 1, [0 1 0 0], .)
p2 [0 1 0 0]	(m1, 1, [0 1 0 0], .)
p3 [0 0 0 1]	(m3, 3, [0 0 0 1], .)

(B)

p0 [0 0 0 1]	(m3, 3, [0 0 0 1], .)
p1 [0 1 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], .)
p2 [0 1 0 0]	(m1, 1, [0 1 0 0], .)
p3 [0 1 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], .)

(C)

p0 [0 1 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], stable)
p1 [0 1 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], .)
p2 [0 1 0 1]	~~(m3, 3, [0 0 0 1], stable)~~ (m1, 1, [0 1 0 0], .)
p3 [0 1 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], .)

(D)

p0 [0 1 1 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], stable)
p1 [0 2 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], stable)
p2 [0 2 1 1]	(m1, 1, [0 1 0 0], .)
p3 [0 1 1 1]	~~(m3, 3, [0 0 0 1], stable)~~ (m1, 1, [0 1 0 0], .)

(E)

p0 [0 1 1 1]	~~(m3, 3, [0 0 0 1], stable)~~ ~~(m1, 1, [0 1 0 0], stable)~~
p1 [0 2 0 1]	(m3, 3, [0 0 0 1], .) (m1, 1, [0 1 0 0], stable)
p2 [0 2 1 1]	~~(m1, 1, [0 1 0 0], stable)~~
p3 [0 1 1 1]	~~(m3, 3, [0 0 0 1], stable)~~ (m1, 1, [0 1 0 0], .)

(F)

p0 [0 1 1 1]	
p1 [0 2 0 1]	~~(m3, 3, [0 0 0 1], stable)~~ ~~(m1, 1, [0 1 0 0], stable)~~
p2 [0 2 1 1]	
p3 [0 1 1 1]	~~(m1, 1, [0 1 0 0], stable)~~

Fig. 6. Illustration of a run of the LCR protocol with 4 processes.

To illustrate the behavior of LCR, consider the following simple example (Figure 6) assuming a system of 4 processes. For simplicity of presentation, we consider that the computation proceeds in rounds. The arrays in Figure 6 depict the state of the pending list stored at each process at the end of each round. At the beginning of round (A), processes p_1 and p_3 broadcast m_1 and m_3, respectively. Message m_1 is the first message broadcast by p_1 and the latter did not receive any message before broadcasting m_1. Therefore, $\mathcal{C}_{m_1} = [0, 1, 0, 0]$. Similarly, $\mathcal{C}_{m_3} = [0, 0, 0, 1]$. At the end of the round, p_1 and p_2 (resp. p_3 and p_0) have m_1 (resp. m_3) in their pending list. During round (B), p_0 (resp. p_2) forwards m_3 (resp. m_1). At the end of the round, the pending lists of p_1 and p_3 contain two messages: m_1 and m_3. Note that in both lists, m_3 is ordered before m_1. Indeed, processes know that m_3 is ordered before m_1 ($m_3 \prec m_1$) because $p_1 < p_3$ and $\mathcal{C}_{m_1}[1] > \mathcal{C}_{m_3}[1]$, which indicates that when p_3 sent m_3, it had not yet received m_1. During round (C), p_1 (resp. p_3) forwards m_3 (resp. m_1). At the end of the round, all processes have both m_1 and

m_3 in their pending list. Moreover, p_0 (resp. p_2) knows that message m_1 (resp. m_3) completed a full round around the ring. Indeed, p_0 (resp. p_2) is the predecessor of the process that sent m_1 (resp. m_3). Consequently, p_0 (resp. p_2) sets m_1 (resp. m_3) to stable. At that time, p_0 (resp. p_2) knows that it will no longer receive any message that is ordered before m_1 (resp. m_3). Indeed, if a message m is ordered before m_1 (resp. m_3), it means, by definition of the total order on messages, that p_3 will receive it before m_1 (resp. m_3). Consequently, p_0 (resp. p_2) will also receive m before m_1 (resp. m_3). Although process p_0 knows that m_1 is stable, it cannot deliver it. Indeed, it first needs to deliver m_3 (which is first in its pending list), which it cannot do. The reason is that it does not know yet whether m_3 is stable or not. Delivering m_3 could violate uniformity. In contrast, process p_2 can deliver m_3 because it is stable and first in its pending list. Process p_2 thus knows that the delivery of m_3 respects total order and uniformity. At the start of round (D), p_0 (resp. p_2) sends an ACK for m_1 (resp. m_3). These ACK messages are forwarded in rounds E and F until they reach the predecessor of the process which initiated the ACK message: for instance, the ACK for message m_1 was initiated by process p_0 in round D. It is forwarded until it reaches process p_3 in round F. Upon reception of these ACK messages (rounds D, E, F), processes set m_1 and m_3 to stable and deliver them as soon as they are first in their pending list.

4.3 Group Membership Changes

The LCR protocol is built on top of a group communication system [Birman and Joseph 1987a]: processes are organized into groups, which they can leave or join, triggering a view change protocol. Faulty processes are excluded from the group after crashing. Upon a membership change, processes agree on a new view. When a process joins or leaves the group, a *view_change* event is generated by the group communication layer and the current view v_r is replaced by a new view v_{r+1}. This can happen when a process crashes or when a process explicitly wants to leave or join the group. As soon as a new view is installed, it becomes the basis for the new ring topology.

The *view_change* procedure is detailed in Figure 7. Note that when a view change occurs, every process first completes the execution (if any) of all other procedures described in Figure 5. It then freezes those procedures and executes the view change procedure. The latter works as follows: every process sends its pending list to all other processes. Upon receiving this list, every process adds to its pending list the messages it did not yet receive. Then the processes send back an ACK_RECOVER message. Processes wait until they receive ACK_RECOVER messages from all processes before sending an END_RECOVERY message to all. When a process receives END_RECOVERY messages from all processes, it can deliver all the messages in its pending list. Thus, at the end of the view change procedure, all pending lists have been emptied, which guarantees that all messages from the old view have been handled.

4.4 Correctness

In this section, we prove that LCR is a uniform total order broadcast protocol. We proceed by successively proving that LCR ensures the four properties mentioned at the beginning of Section 4: validity, integrity, uniform agreement and total order.

```
Procedures executed by any process $p_i$
 1: upon view_change($new\_view$) do
 2:    Rsend $\langle$RECOVER, $p_i$, $pending\rangle$ to all $p_j \in new\_view$
 3:    Wait until received $\langle$ACK_RECOVER$\rangle$ from all $p_j \in new\_view$
 4:    Rsend $\langle$END_RECOVERY$\rangle$ to all $p_j \in new\_view$
 5:    Wait until received $\langle$END_RECOVERY$\rangle$ from all $p_j \in new\_view$
 6:    forceDeliver()
 7:    $view \leftarrow new\_view$

 8: upon Rreceive $\langle$RECOVER, $p_j$, $pending_{p_j}\rangle$ do
 9:    for each $[m, p_l, \mathcal{C}_m, *] \in pending_{p_j}$ do
10:       if $\mathcal{C}_m[l] > \mathcal{C}[l]$ then
11:          $pending \leftarrow pending \cup [m, p_l, \mathcal{C}_m, \perp]$
12:    Rsend $\langle$ACK_RECOVER$\rangle$ to $p_j$

13: procedure forceDeliver()
14:    for each $[m, p_k, \mathcal{C}_m, *] \in pending$ do
15:       utoDeliver($m$)                                   {deliver a message}
16:       $pending \leftarrow pending - [m, p_k, \mathcal{C}_m, *]$
17:       $\mathcal{C}[k] \leftarrow \mathcal{C}[k] + 1$                     {update local vector clock}
```

Fig. 7. Pseudo-code of the view_change procedure.

LEMMA 4.1 VALIDITY. *If any correct process p_i utoBroadcasts a message m, then it eventually utoDelivers m.*

PROOF. Let p_i be a correct process and let m_i be a message broadcast by p_i. This message is added to p_i's pending list (Line 7 of Figure 5). If there is a membership change, p_i being a correct process, it will be in the new view. Consequently, the view change procedure guarantees that p_i will deliver all messages stored in its pending list (Line 6 of Figure 7). It will thus deliver m_i. Let us now consider the case when there is no membership change. All processes (including p_i) will eventually set m_i to stable (Line 21 of Figure 5). This is due to the fact that m_i will be forwarded along the ring (because $\mathcal{C}_{m_i}[i]$ is higher than the i^{th} value stored in the clock of each process) until it reaches the predecessor of p_i. The latter marks m_i as stable (line 15 of Figure 5) and sends an ACK to its successor in the ring containing $\mathcal{C}_{m_i}$. Similarly to m_i, the ACK message is forwarded along the ring (line 22 of Figure 5) until it reaches the predecessor of the predecessor of p_i. Upon receiving the ACK message, each process marks m_i as stable. When p_i sets m_i to stable, its pending list starts with a (possibly empty) set of messages m such that $m \prec m_i$ and m has not been yet delivered by p_i. Let us call *undelivered* this set of messages. Let us first remark that this set cannot grow. Consider, for instance, the case of a message m_j sent by a process p_j that precedes m_i (i.e. $m_j \prec m_i$). We know that p_j sent m_j before receiving m_i. Consequently, the predecessor of p_i will receive m_j before receiving m_i, and thus before sending the ACK for m_i. As each process forwards messages in the order in which it receives them, we know that p_i will necessarily receive m_j before receiving the ACK for message m_i. Let us now consider every message in *undelivered*. As there is no membership change, the same reasoning as the one we did for m_i applies to messages in *undelivered*: every process will eventually set these messages to stable. Consequently, all messages in *undelivered* will be delivered. Message m_i will thus be first in *pending* and marked as stable. It will thus also be delivered by p_i. □

LEMMA 4.2 INTEGRITY. *For any message m, any process p_j* utoDelivers *m at most once, and only if m was previously* utoBroadcast *by some process p_i.*

PROOF. No spurious message is ever utoDelivered by a process as we assume only crash failures. Thus, only messages that have been utoBroadcast are utoDelivered. Moreover, each process keeps a vector clock $\mathcal{C}$, which is updated in such a way that we are sure that every message is only delivered once. Indeed, if there is no membership change, Lines 10 and 18 of Figure 5 guarantee that no message can be processed twice by p_j. Similarly, when there is a membership change, Line 10 of Figure 7 guarantees that process p_j will not deliver messages twice. Moreover, Line 17 of Figure 7 guarantees that p_j's vector clock is updated after the membership change, thus preventing the future delivery of messages that have been delivered during the *view_change* procedure. □

LEMMA 4.3 UNIFORM AGREEMENT. *If any process p_i* utoDelivers *any message m in the current view, then every correct process p_j in the current view eventually* utoDelivers *m.*

PROOF. Let m_k be a message sent by process p_k and let p_i be a process that delivered m_k in the current view. There are two cases to consider. In the first case, p_i delivered m_k during a membership change. This means that p_i had m_k in its pending list before executing line 15 of Figure 7. Since all correct processes exchange their pending list during the view change procedure, we are sure that all correct processes that did not deliver m_k before the membership change will have it in their pending list before executing line 6 of Figure 7. Consequently, all correct processes in the current view will deliver m_k. The second case to consider is when p_i delivered m_k outside of a membership change. The protocol ensures that m_k did a full round around the ring before being delivered by p_i: indeed p_i can only deliver m_k after having set it to stable, which either happens when it is the predecessor of p_k in the ring or when it receives an ACK for message m_k. Consequently, all processes stored m_k in their pending list before p_i delivered it. If a membership change occurs after p_i delivered m_k and before all other correct processes delivered it, the protocol ensures that all correct processes that did not yet deliver m_k will do it (Line 6 of Figure 7). If there is no membership change after p_i delivered m_k and before all other processes delivered it, the protocol ensures that an ACK for m_k will be forwarded around the ring, which will cause all processes to set m_k to stable. Each correct process will thus be able to deliver m_k as soon as m_k will be first in the pending list (Line 26 of Figure 5). The protocol ensures that m_k will become first eventually. The reasons are the following: (1) the number of messages that are before m_k in the pending list of every process p_j is strictly decreasing, and (2) all messages that are before m_k in the pending list of a correct process p_j will become stable eventually. The first reason is a consequence of the fact that once a process p_j sets message m_k to stable, it can no longer receive any message m such that $m \prec m_k$. Indeed, a process p_l can only produce a message $m_l \prec m_k$

before receiving m_k. As each process forwards messages in the order in which it received them, we are sure that the process that will produce an ACK for m_k will have first received m_l. Consequently, every process setting m_k to stable will have first received m_l. The second reason is a consequence of the fact that for every message that is utoBroacast in the system, the protocol ensures that an ACK will be forwarded around the ring (Lines 16 and 22 of Figure 5), implying that all correct processes will mark the message as stable. Consequently, all correct processes will eventually deliver m_k. $\square$

LEMMA 4.4 TOTAL ORDER. *For any two messages m and m', if any process p_i* utoDelivers *m without having delivered m', then no process p_j* utoDelivers *m' before m.*

PROOF. We prove the lemma by contradiction. Let m and m' be any two messages and let p_i be a process that utoDelivers m without having delivered m'. Consider a process p_j that utoDelivers m' before delivering m. Let us denote t_i the time at which p_i delivered m and t_j the time at which p_j delivered m'. Let us first note that the protocol ensures that at time t_i (resp. t_j), all processes have already received m (resp. m'). Indeed, when there is no membership change, a message can only be delivered after it is set to stable, which requires the message to have done a full round around the ring. When there is a membership change, the protocol ensures (by broadcasting messages stored in pending lists and waiting for all processes to have received all broadcast before delivering any message) that all processes have a consistent pending list before delivering messages (Lines 2 to 5 of Figure 7).

Case 1: $t_j \leq t_i$. It follows that at time t_i, process p_i had already received m'. Consequently, $m \prec m'$, otherwise, p_i would have delivered m' first. We will show that in that case, we are sure that p_j received m before m', thus contradicting the fact that it delivered m' before m. Let us note p_k the process that utoBroadcast m. Provided $m \prec m'$, we know that p_k sent m before receiving m'. There are two cases to consider: if there is no membership change before p_j delivers m', as each process forwards messages in the order in which it received them, we are sure that p_j will receive m before it can set m' to stable. The second case is when there is a membership change before p_j delivers m'. Process p_j could not receive m before the membership change, otherwise, it would not deliver m' before m (since we know that $m \prec m'$). Provided that p_i delivers message m, we know that this can only happen during the membership change. Indeed, the message can not have been delivered before, otherwise p_j would have received it. Consequently, at the beginning of the view change procedure, p_i has m in its pending list and will send it to all processes. Consequently, p_j will receive m during the view change procedure. In both cases, we are sure that p_j received m before delivering m', which is in contradiction with the fact that it delivered m' before m provided that $m \prec m'$.

Case 2: $t_i < t_j$. It follows that at time t_j, process p_j had already received m. Consequently, $m' \prec m$, otherwise, p_j would have delivered m first. With a similar reasoning as the one we did for case 1, we know that p_i received m' before delivering m, which is in contradiction with the fact that it delivered m without delivering m' provided that $m' \prec m$.

□

Theorem 4.5. *LCR is a uniform total order broadcast protocol.*

Proof. By lemmas 4.1, 4.2, 4.3, and 4.4, we can derive the very fact that the LCR protocol ensures validity, integrity, uniform agreement, and total order. Thus, it is a uniform total order broadcast protocol. □

5. THEORETICAL ANALYSIS

This section analyzes several key aspects of LCR's performance from a theoretical perspective. The performance of LCR is evaluated in failure free runs which we expect to be the common case. We prove that LCR is throughput optimal in such case. Then we discuss its fairness.

5.1 Throughput

In this section we show that the throughput of LCR is optimal and that no other broadcast protocol (even with weaker consistency guarantees) can obtain strictly higher throughput. We do this by proving an upper bound on the performance of any broadcast protocol and show that LCR matches this bound.

Theorem 5.1 Maximum throughput for any broadcast protocol. *For a broadcast protocol in a system with n processes in the round-based model introduced in Section 2.2, the maximum throughput μ_{max} in completed broadcasts per round is:*

$$\mu_{max} = \begin{cases} n/(n-1) & \textit{if there are } n \textit{ senders} \\ 1 & \textit{otherwise} \end{cases}$$

Proof. We first consider the case with n senders. Each broadcast message must be received at least $n-1$ times in order to be delivered. The model states that at each round at most n messages can be received. Thus, for n processes to broadcast a message, a minimum of $n-1$ rounds are necessary. Therefore, on average, at most $n/(n-1)$ broadcasts can be completed each round. In the case with less than n senders it is sufficient to look at a non sending process. Such a process can receive at most 1 message per round and since it doesn't broadcast any messages itself, it can deliver at most 1 message per round. Since the throughput is defined as the number of *completed* broadcasts per round, the maximum throughput with less than n senders is equal to 1. □

Determining the throughput of LCR is straightforward: processes receive one message per round and the acknowledgements are piggy-backed. Thus LCR allows each process to deliver one message per round if there is at least one sender. When there are n senders, each process can deliver one message per round broadcast by other processes in addition to its own messages. LCR thus matches the bound of Theorem 5.1 and is theoretically throughput optimal. Thus, from a throughput perspective, the strong uniform total order guarantees provided by LCR are free.

5.2 Fairness

Even though the throughput of LCR as described thus far is optimal, there is still a problem. Consider two processes p_1 and p_2 that are neighbors on the ring. If p_1 is continuously broadcasting messages and p_2 systematically forwards p_1's messages, then p_2 cannot broadcast its own messages. Consequently, the protocol is not *fair*.

Fairness captures the fact that each process has an equal opportunity of having its messages delivered by all processes. Intuitively, the notion of fairness means that in the long run no single process has priority over other processes when broadcasting messages. Said differently, when two processes want to broadcast a large number of messages during a time interval τ, then each process should have approximately the same number of messages delivered by all processes during τ.

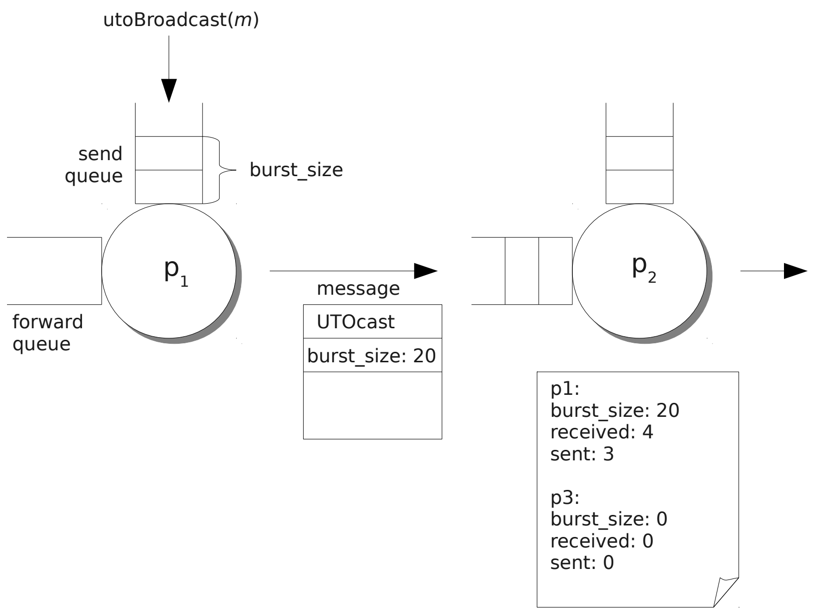


Fig. 8. Illustration of the fairness mechanism as implemented in LCR. Each process has two queues (send and forward) and uses the burst_nb, received, sent variables to determine whether to forward messages or send its own.

The mechanism for ensuring fairness in LCR acts locally at each process. If a process wishes to broadcast a new message, it must decide whether to forward a message received from its predecessor or to send its own. Figure 8 provides an illustration of the fairness mechanism as implemented in LCR. Processes have two queues: send and forward. Processes put broadcast requests coming from the application level in their send queue. Messages received from predecessors that need to be forwarded are buffered in the forward queue. When a process has a burst of messages to send (i.e. it has more than one message in its send queue), it piggybacks on the first message it sends an integer burst_size representing the number of messages currently stored in its send queue. Each process keeps a data structure which stores 3 integers per process in the ring: burst_size, received and sent. For each process p_i, the burst_size variable is updated every time p_i piggybacks a new burst_size value on a message it sends. The received variable keeps track of the number of messages that the process received from p_i since p_i's burst_size has been updated. The sent variable keeps track of the number of messages that the process sent since p_i's burst_size has been updated. When the received variable is equal to the burst_size, the three integers kept for process p_i are reset (i.e. the burst initiated by p_i is finished). When the process wants to send a message, it retrieves the first message in the forward list. Assume that this message has been sent by process p_j. The process only sends its own message if the received integer stored for p_j is higher or equal than the sent integer stored for p_j, which intuitively means that since p_j started its last burst, the process initiated less broadcasts than p_j.

5.3 Latency

The theoretical latency of broadcasting a single message is defined as the number of rounds that are necessary from the initial broadcast of message m until the last process delivers m. The latency of LCR is equal to $2n - 2$ rounds.

6. EXPERIMENTAL EVALUATION

This section compares the performance of LCR to that of two existing group communication systems: JGroups and Spread. Spread ensures uniform total order delivery of messages, whereas JGroups only guarantees *non-uniform* total order delivery. The experiments only evaluate the failure free case because failures are expected to be sufficiently rare in the targeted environment. Furthermore, the view change procedure that is used in LCR is similar to that of other total order broadcast protocols [Défago et al. 2004].

We first present the experimental setting we used in the experiments and then study various performance metrics: throughput, response time, fairness, and CPU consumption. In particular, we show that LCR always achieves a significantly better throughput than Spread and JGroups when all processes broadcast message. We also show that when only one process broadcasts messages, LCR outperforms Spread and has similar performance than JGroups. Regarding response time, we show that LCR exhibits a higher response time than Spread and JGroups when there is only one sender in the system. In contrast, it outperforms both protocols when all processes broadcast messages. Finally, we show that LCR and Spread are both fair and have a low CPU consumption. This contrasts with JGroups that is not fair and has a higher CPU consumption than Spread and LCR.

6.1 Experimental Setup

The experiments were run on a cluster of machines with a $1.66GHz$ bi-processor and $2GB$ RAM. Machines run the Linux $2.6.30-1$ SMP kernel and are connected using a Fast Ethernet switch. The raw bandwidth over IP is measured with Netperf [Jones 2007] between two machines and displayed in Table I.

Protocol	**Bandwith**
TCP	$93Mb/s$
UDP	$93Mb/s$

Table I. Raw network performance measured using Netperf.

The LCR protocol is implemented in C (≈ 1000 lines of code). It relies on the Spread toolkit to provide a group membership layer and uses TCP for communication between processes. As explained in [Dunagan et al. 2004], it is reasonable to assume, in an low-latency cluster, that when a TCP connection fails, the server on the other side of the connection failed. It is thus easy to implement the abstraction of a perfect failure detector [Chandra and Toueg 1996a]. Moreover, using TCP, it is not necessary to implement a message retransmission mechanism: a message sent from a correct process to another correct process will be delivered eventually.

The implementation of LCR is benchmarked against two industry standard group communication systems:

—*Spread.* We use Spread version 4.1 [Amir et al. 2004]. The message type was set to `SAFE_MESS` which guarantees uniform total order. Spread implements a privilege-based ordering scheme (see Section 3.3). A Spread daemon was deployed on each machine. All daemons belong to the same Spread segment. Spread was tuned for bursty traffic according to Section 2.4.3 of the Spread user guide [Stanton 2002]. Our benchmark uses the native C API provided by Spread.

—*JGroups.* We use JGroups version 2.7.0 [Ban 2007] with the Java HotSpot Server 1.6.0_16 virtual machine. We use the "`sequencer`" stack that contains the following protocols: `UDP, PING, MERGE2, FD_SOCK, FD_ALL, VERIFY_SUSPECT, BARRIER, pbcast.NAKACK, UNICAST, pbcast.STABLE, VIEW_SYNC, pbcast.GMS, SEQUENCER, FC, FRAG2, STATE_TRANSFER`. This stack provides *non uniform* total ordering. It implements a fixed-sequencer ordering scheme without acknowledgements (see Section 3.1).

All experiments we present in this section start with a warm-up phase, followed by a phase during which performance are measured. Finally, there is a cool-down phase without measurements. The warm-up and cool-down phases last 5 minutes. The measurement phase lasts 10 minutes.

6.2 Throughput

To assess the throughput of total order broadcast protocols, we use the following benchmark: k processes out of the n processes in the system broadcast messages at a predefined throughput (we call this experiment k-to-n broadcast). Each message has a fixed size, which is a parameter of the experiment. Each process periodically computes the throughput at which it delivers messages. The throughput is calculated as the ratio of delivered bytes over the time elapsed since the end of the warm-up phase. The plotted throughput is the average of the values computed by each process.

We first want to confirm our claim that LCR achieves optimal throughput. Figure 9 shows the results of an experiment with $n = 5$ processes. We vary the number k of broadcasting processes (X axis). The size of messages broadcast by processes is 10kB. Moreover, each broadcasting processes produce messages at the maximum throughput it can sustain. We first execute LCR without enabling the fairness mechanism described in Section 5.2. We observe that the throughput obtained by LCR is far from optimal: in practice, a theoretical throughput of 1 should be equal to the raw link speed between the processes, i.e. 93Mbit/s as shown in Table I. The reason why the throughput is not optimal is that when the fairness mechanism is disabled, senders can broadcast more messages than can be delivered, resulting in overflowing network buffers and the data structures maintained by each process.

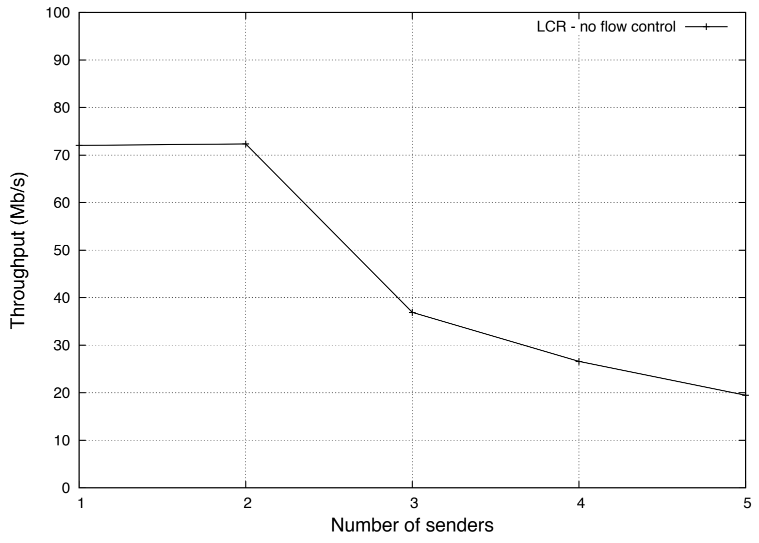


Fig. 9. LCR throughput with the fairness mechanism disabled in a system with 5 processes. Buffers are quickly saturated, which explains the low throughput.

Figure 10 depicts the throughput obtained by LCR when the fairness mechanism is enabled. The throughput clearly improves and is now close to optimal. The reason why the throughput improves is that the fairness mechanism throttles the senders, and does thus prevent them from injecting too many new messages into the ring. Note that the throughput with 5 senders is higher than with fewer senders, the reason being LCR's theoretical throughput of $n/(n-1)$ when all processes are senders. Finally, note that in all subsequent experiments, the fairness mechanism is enabled.

The next experiments we present (Figure 11 and 12) measure the impact of varying the message size on the throughput of LCR, Spread and JGroups for a system with 5 processes. In the experiment depicted in Figure 11, only one process broadcasts messages, whereas all processes broadcast messages in the experiment depicted in Figure 12. In both cases, we can observe that if the messages are too small, the throughput of all protocols suffers. This is due to the cost of ordering which remains constant despite a decrease in payload size. We can nevertheless observe that LCR achieves significantly better performance with small messages than Spread and JGroups. To improve performance, it is possible to batch small messages together into bigger messages when the load on the system is high as suggested in [Friedman and Renesse 1997b]. For all further experiments the message size is set to $10kB$, which is the optimal message size for the three protocols in both the 1-to-n and n-to-n cases.

Having studied the impact of message size, we now study how the throughput evolves as a function of the number of processes. Figure 13 plots the results of

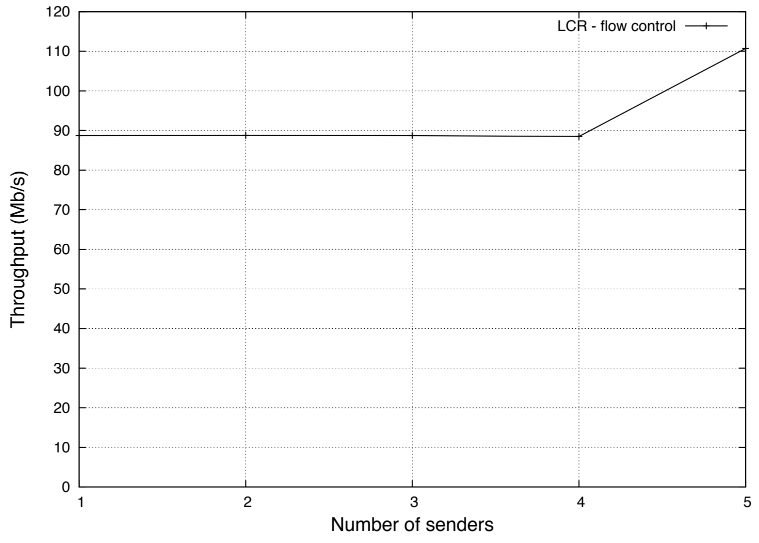


Fig. 10. LCR throughput with the fairness mechanism enabled in a system with 5 processes. Senders are throttled, which improves the throughput.

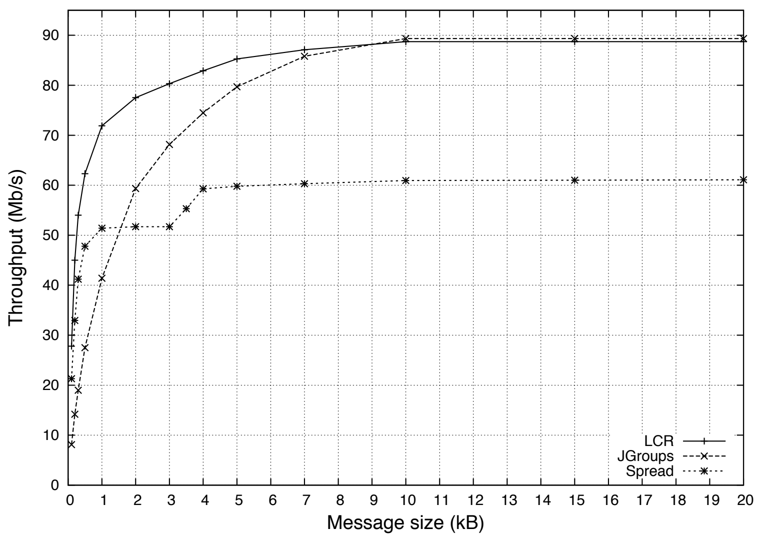


Fig. 11. Throughput with respect to message size for a system of 5 processes with one sender.

an experiment consisting in 1-to-n broadcasts of $10kB$ messages. The throughput achieved by LCR and JGroups is close to optimal and almost constant despite the increasing number of processes. Note that JGroups does not provide uniformity, and does thus implement a very simple communication pattern: the sender sends its messages to the sequencer, which multicast them to all other processes. Spread's throughput suffers a bit more from increasing the number of processes. This can be explained by the fact that Spread uses a token to order messages and ensure

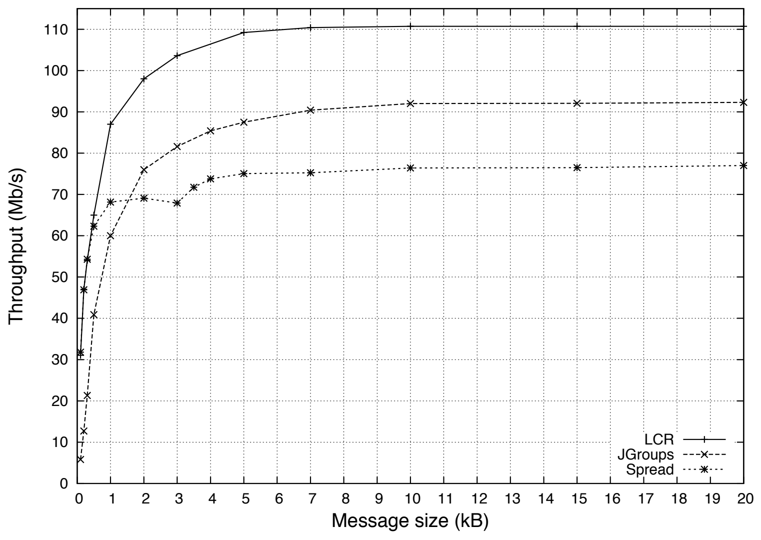


Fig. 12. Throughput with respect to message size for a system of 5 processes with 5 senders.

uniformity. Increasing the number of processes increases the time it takes for the token to circulate among all processes.

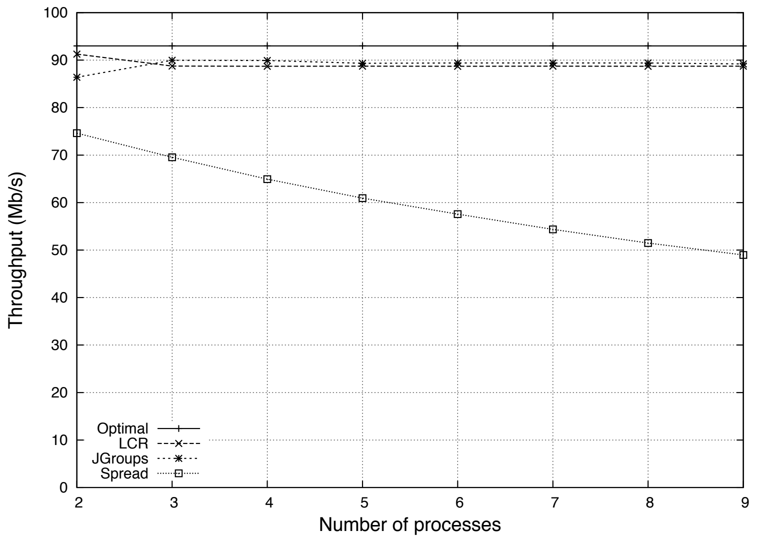


Fig. 13. 1-to-n throughput comparison. The optimal line is constant at $93Mb/s$.

Figure 14 plots the throughput as a function of the number of processes in a system where all processes broadcast $10kB$ messages. The optimal line for best effort broadcast ($n/(n-1)$ times the maximum link speed of $93Mb/s$) is plotted as a reference. We can first observe that the throughput of LCR is very close to

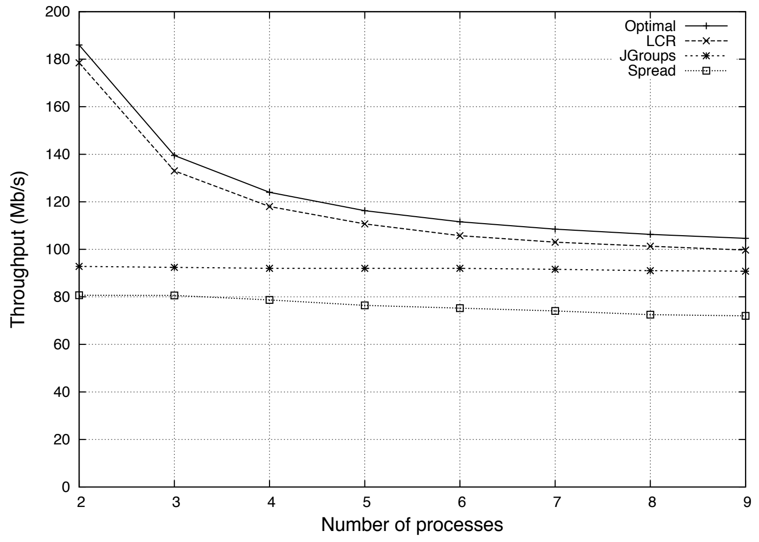


Fig. 14. n-to-n throughput comparison. The optimal line is calculated as $n/(n-1)*93Mb/s$. The LCR implementations closely follow the optimal line as does JGroups. JGroups however does not ensure uniformity or ordering. Spread's throughput is limited by the underlying privilege-based broadcast scheme.

optimal. Since the optimality line is calculated for best effort broadcast and LCR provides uniform total order broadcast, we can conclude that the ordering, reliability and uniformity properties of LCR are effectively almost free. The throughput of JGroups is almost constant (at $92Mb/s$) and only slightly better than the throughput achieved with only one sender (Figure 13). This can be easily explained by the fact that the throughput is limited by the throughput at which the sequencer can broadcast messages to other processes in the system (using IP multicast). The throughput achieved by Spread is better than in the 1-to-n case due to the fact that every process makes use of the token to broadcast the messages it produces. Nevertheless, as in the 1-to-n case, the throughput slightly decreases when the number of processes increases due to the increasing cost of ensuring uniformity (cost that JGroups does not have).

To summarize, we can say that LCR is the only protocol that fully exploits available network links when all processes broadcast messages, which we believe is the common case in many applications. It does thus sustain a significantly higher throughput than other protocols. For instance, the gain in throughput in a system with 4 processes is of about 28% compared to JGroups and of about 49% compared to Spread.

6.3 Response Time

In this section, we evaluate the response time of LCR, Spread and JGroups in the 1-to-n and n-to-n cases. We setup a system with 5 processes. We vary the throughput at which the sender processes inject new messages. The size of messages that are broadcast is $10kB$. During the measurement phase, for every message m it broadcasts, the sender evaluates the elapsed time between the broadcast and the delivery of m. For each protocol, we stop the curve when the injected load is higher than the throughput the protocol is able to sustain.

Figure 15 depicts the results obtained with one sender. In order to evaluate the general case where every process could be the sender, we do not colocate the sender and the sequencer in JGroups. We observe that Spread exhibits a consistently lower response time than JGroups and LCR ($3ms$ against $4ms$ for JGroups and $5ms$ for LCR when the input load is below $10Mb/s$). The fact that LCR exhibits higher response time in the 1-to-n case is not surprising provided that processes are organized in a ring topology. Interestingly, LCR's response time does not degrade when the input load increases. This is in contrast with the response time of JGroups which increases when the input load is greater than $10Mb/s$ (a similar behavior is observed in the n-to-n case). Finally, a last remark we can make is that, although providing uniform delivery of messages, Spread achieves better response time than JGroups. This is due to the fact that Spread uses a token. Once the sender process obtains the token, it can send messages in burst, thus decreasing the average response time. In JGroups, the sender always needs to first send the message to the sequencer, which will then multicast the message to other processes. Finally, note that the fact that the sender in Spread can send multiple messages in burst when it owns the token also explains why the response time slightly decreases when the load gets higher (between $20Mb/s$ and $40Mb/s$).

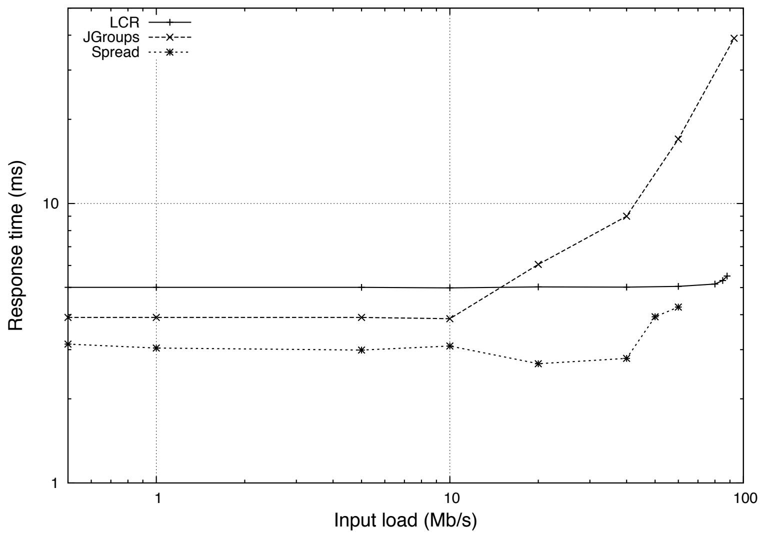


Fig. 15. 1-to-n response time comparison.

Figure 16 depicts the results obtained with n senders (note that the scale used on the Y axis is different than in Figure 15). LCR and JGroups exhibit a lower response time than Spread ($5.2ms$ for JGroups against $5.4ms$ for LCR and $7ms$ for Spread when the input load is below $10Mb/s$). The reason is that with Spread, senders must wait to have the token before sending their messages. As every process has messages to send, it takes a longer time to obtain the token. Note that the response time of Spread is almost constant, and only slightly increases when the maximum delivery throughput is reached. The same remark applies to LCR for which the response time only slightly increases when the input load is higher than $80Mb/s$ but remains low until the maximum delivery throughput is reached (up to $108Mb/s$, the response time is below $7.5ms$). In contrast, the response time of JGroups starts degrading when the input load is higher than $10Mb/s$. It becomes high for input loads higher than $70Mb/s$. This is probably due to the fact that the sequencer simultaneously receives messages from all other processes, which fills up its network buffers and increases the time it takes for a message to be sequenced.

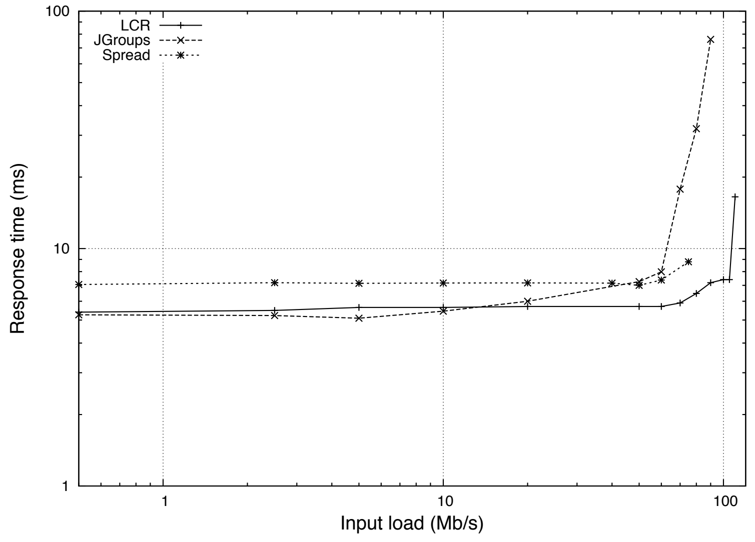


Fig. 16. n-to-n response time comparison.

To summarize, LCR exhibits a higher response time than other protocols when there is only one sender. When there are multiple senders, LCR's response time equals or outperforms those of other protocols. More precisely, it exhibits better response time than Spread and JGroups (when the load is higher than $20Mb/s$), and similar response time than JGroups (when the load is below $20Mb/s$), but contrarily to the latter, it ensures uniform delivery of messages and the response time does not degrade when the input load increases.

6.4 Fairness

We evaluate the fairness of LCR, JGroups and Spread as follows: we setup a system with 5 processes, of which 3 are senders. Each sender continuously broadcasts messages. At the end of the measurement phase, every process computes the percentage of messages it delivered that were issued by each of the 3 senders (called p_1, p_2 and p_3). In the case of JGroups, process p_1 is also the sequencer. The results are depicted in Figure 17. We can first observe that both LCR and Spread are fair: each process delivers 33% of messages from each sender. Concerning LCR, this is due to the fact that it implements the fairness mechanism described in Section 5.2.

Concerning Spread, this is due to the fact that, on average, each sender owns the token for the same amount of time. We can also observe that JGroups is not fair. This comes from the fact that processes p_2 and p_3 first need to send their messages to the sequencer p_1, whereas the latter can directly broadcast the messages it produces. This induces a significant unbalance: 50% of the messages that are delivered have been broadcast by p_1.

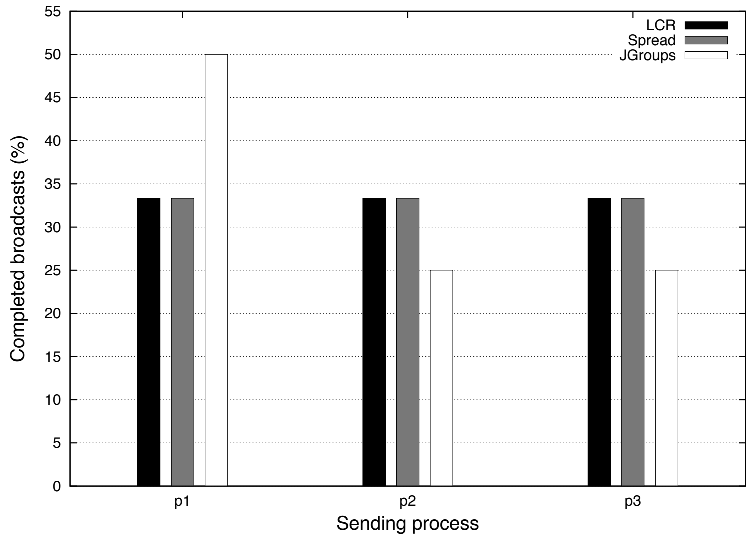
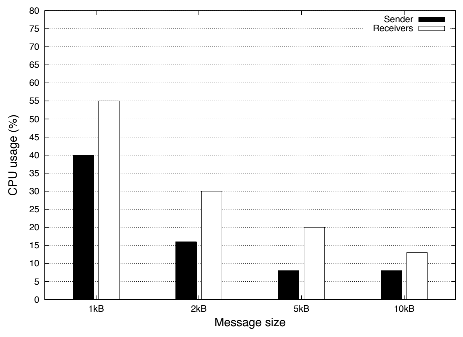


Fig. 17. Fairness assessment of the LCR, Spread and JGroups protocols. Experiments were performed with 5 processes and 3 senders.

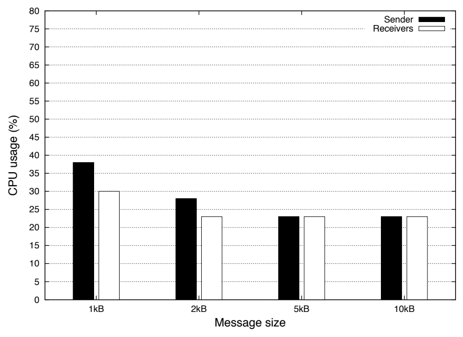
6.5 CPU Usage

The last performance metric we evaluate is the CPU usage of LCR, Spread and JGroups under high load. During the experiment, the CPU usage of all active protocol threads was periodically logged, added up and averaged. We study both the 1-to-n and n-to-n cases.

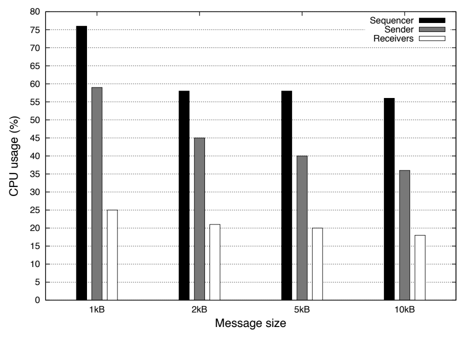
The experiment in Figure 18 plots the CPU usage measured in a system with 5 processes, of which one broadcasts $10kB$ messages. The X axis represents the message size (in kB). The Y axis represents the CPU consumption (in %). To ease the comparison between the various protocols, the three graphs use the same scale. In the case of JGroups, the sender was not sequencer in order to be able to isolate the CPU consumption of the sequencer, the sender and receiver processes, respectively. The first remark we can make is that among the three protocols, JGroups has the highest CPU consumption. In particular, the sequencer process consumed more than 55% of the CPU in all experiments we performed. The sender performs also significantly more work than receiver processes due to the fact that it needs both to send and receive the messages it broadcasts. Spread and LCR have a CPU usage that is very reasonable: it is systematically below 30% with messages bigger than $1kB$. In LCR, it is interesting to notice that the sender has less work to do than other processes. The reason is that the sender does not receive messages to forward; it only receives acks. In contrast, the sender and the receivers in Spread use the same percentage of CPU time. Finally, it is interesting to note that Spread uses more CPU than LCR for large messages.



(a) LCR



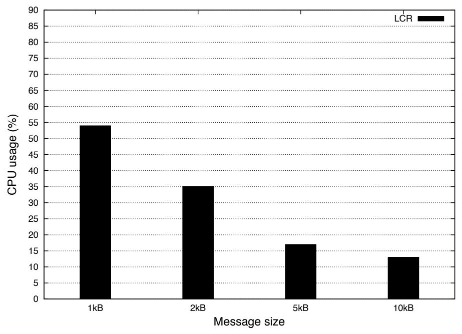
(b) Spread



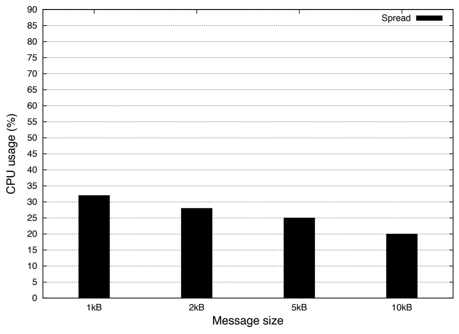
(c) JGroups

Fig. 18. CPU usage during high load 1-to-n broadcasts of the LCR, Spread, and JGroups protocols.

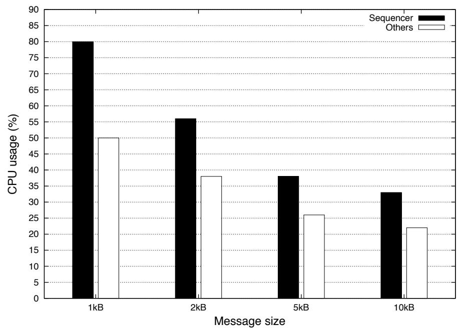
The experiment in Figure 18 plots the CPU usage measured in a system with 5 processes, each broadcasting $10kB$ messages. As was the case with only one sender, we observe that JGroups consumes more CPU than other protocols. Interestingly, the consumption of the sequencer is a bit lower than what it was in the previous experiment. We explain this by the fact that in this experiment, the sequencer itself broadcasts messages, thus reducing the number of messages it gets from other processes, and thus its CPU consumption. We can also remark that Spread and LCR have a slightly higher CPU usage than in the previous case. This is explained by the fact that both protocols handle more messages (they achieve higher throughput) in the n-to-n case than in the 1-to-n case, thus requiring higher CPU usage. Finally, we observe that, similarly to the previous case, LCR consumes less CPU than Spread when messages are larger.



(a) LCR



(b) Spread



(c) JGroups

Fig. 19. CPU usage during high load n-to-n broadcasts of the LCR, Spread, and JGroups protocols.

7. SUMMARY

We have presented LCR, a uniform total order broadcast protocol that can be used as the main communication block of a replication scheme to achieve software-based fault-tolerance.

LCR is the first uniform total order broadcast protocol that is throughput optimal in failure-free periods. In short, throughput optimality captures the ability to deliver the largest possible number of message broadcasts, regardless of message broadcast patterns. This notion is precisely defined in a round-based model of computation which accurately captures message passing interaction patterns over clusters of homogeneous machines interconnected by a fully switched LAN. LCR is based on a ring topology and only relies on point-to-point inter-process communication. LCR is also fair in the sense that each process has an equal opportunity of having its messages delivered by all processes. Performance benchmarks showed that LCR had a very high throughput in all cases, while exhibiting very reasonable response time. Moreover, benchmarks showed that LCR has a low CPU usage.

LCR has been designed for homogeneous clusters, in which machines are connected by a fully-switched, dedicated network. The ring topology it relies on would clearly not be adequate in an environment where nodes would not be homogeneous, or where the network would not be dedicated. We believe that an interesting research challenge is to understand how ring-based communication patterns can be combined with other communication patterns (e.g., tree, multicast) to ensure high performance in heterogeneous settings.

8. ACKNOWLEDGEMENTS

We would like to thank Idit Keidar for her insightful comments on this paper. Furthermore, we are very grateful towards Yair Amir, Bela Ban, Emmanuel Cecchet, and Dejan Kostić for the useful discussions on the topic.

REFERENCES

Amir, Y., Danilov, C., Miskin-Amir, M., Schultz, J., and Stanton, J. 2004. The spread toolkit: Architecture and performance. Tech. rep., CNDS-2004-1, Johns Hopkins University.

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

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

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

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

Ban, B. 2007. *JGroups – A Toolkit for Reliable Multicast Communication.* http://www.jgroups.org.

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

Bar-Noy, A. and Kipnis, S. 1997. Multiple message broadcasting in the postal model. *Networks 29,* 1, 1–10.

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

Birman, K. and Joseph, T. 1987b. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5,* 1, 47–76.

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

Carr, R. 1985. The tandem global update protocol. *Tandem Syst. Rev. 1*, 74–85.

Cecchet, E., Marguerite, J., and Zwaenepoel, W. 2004. Cjdbc: Flexible database clustering middleware.

Chandra, T. and Toueg, S. 1996a. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43,* 2, 225–267.

Chandra, T. and Toueg, S. 1996b. Unreliable failure detectors for reliable distributed systems. *J. ACM 43,* 2, 225–267.

Chang, J.-M. and Maxemchuk, N. 1984. Reliable broadcast protocols. *ACM Trans. Comput. Syst. 2,* 3, 251–273.

Cristian, F. 1991. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin 33,* 9, 115–116.

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

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

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

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

Dunagan, J., Harvey, N. J. A., Jones, M. B., Kostic, D., Theimer, M., and Wolman, A. 2004. Fuse: Lightweight guaranteed distributed failure notification. In *Proceedings of 6th Symposium on Operating Systems Design and Implementation (OSDI '04).*

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

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

Friedman, T. and Renesse, R. V. 1997a. 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)*. IEEE Computer Society, Washington, DC, USA.

Friedman, T. and Renesse, R. V. 1997b. Packing messages as a tool for boosting the performance of total ordering protocls. In *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing.* IEEE Computer Society, Washington, DC, USA, 233.

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

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

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

Guerraoui, R., Kostic, D., Levy, R. R., and Quéma, V. 2007. A High Throughput Atomic Storage Algorithm. In *The 27th IEEE International Conference on Distributed Computing Systems (ICDCS'07)*. Toronto, Canada.

Guerraoui, R., Levy, R. R., Pochon, B., and Quéma, V. 2006. High Throughput Total Order Broadcast for Cluster Environments. In *IEEE International Conference on Dependable Systems and Networks (DSN'06)*. Philadelphia, PA, USA.

Hadzilacos, V. and Toueg, S. 1993. Fault-tolerant broadcasts and related problems. 97–145.

Jones, R. 2007. *Netperf.* http://www.netperf.org/.

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

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

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

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

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

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

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

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

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

Schneider, F. B. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv. 22,* 4, 299–319.

Stanton, J. R. 2002. *A Users Guide to Spread*. http://www.spread.org/docs/guide/users_guide.pdf.

Urbán, P., Dfago, X., and Schiper, A. 2000. 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)*. 582–589.

van Renesse, R. and Schneider, F. B. 2004. Chain replication for supporting high throughput and availability. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*. USENIX Association, Berkeley, CA, USA, 7–7.

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

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

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