FastCast: A Throughput- and Latency-Efficient Total Order Broadcast Protocol

Gautier Berthou[1] and Vivien Quéma[2]

[1] Grenoble University
[2] Grenoble INP

Abstract. Many uniform total order broadcast protocols have been designed in the last 30 years. Unfortunately, none of them achieves both optimal throughput and low latency. Indeed, protocols achieving optimal throughput rely on a ring dissemination pattern, which induces high latencies. Protocols achieving low latency rely on IP multicast and fail to achieve good throughput because of message losses. In this paper, we describe *FastCast*, the first protocol that achieves both optimal throughput and low latency. To achieve low latency, *FastCast* relies on IP multicast. To achieve optimal throughput, *FastCast* defines a protocol responsible for dynamically computing the throughput at which processes can send IP multicast messages. Thanks to this dynamic bandwidth allocation protocol, *FastCast* allows multiple processes to simultaneously send messages, while avoiding message losses. An evaluation of *FastCast* on a cluster of 8 machines shows that it indeed achieves optimal throughput and a very low latency.

1 Introduction

State-machine replication [1] is a popular technique to ensure fault-tolerance in computer systems. The operating principle of state-machine replication is simple: several replicas of the same software object are maintained on different machines (also called processes). Each replica executes the same requests in the same order and is thus consistent with other replicas. Consequently, if one or more replicas fail, remaining replicas are consistent and guarantee accessibility to the object. To ensure that replicas execute requests in the same order, each replica broadcasts the requests it receives to other replicas using a *uniform total order broadcast* [2], and executes requests in the order in which they are delivered by the protocol. A uniform total order broadcast protocol ensures the following properties 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.

Many uniform total order broadcast protocols have been designed in the last 30 years [3]. They can be classified into two categories: those targeting *low latency*, and those targeting *high throughput*. Latency measures the time required to complete a single message broadcast without contention, whereas throughput

D. Eyers and K. Schwan (Eds.): Middleware 2013, LNCS 8275, pp. 1–20, 2013.
© IFIP International Federation for Information Processing 2013

measures the number of broadcasts that the processes can complete per time unit when there is contention.

Protocols targeting low latency usually rely on IP multicast, a low-level networking protocol allowing senders to reach multiples destinations using a single message. These protocols do not achieve high throughput for the following reason: IP multicast messages are dropped when the network is congested. To limit congestion, protocols are designed in such a way that only one process at a time can send IP multicast messages. As we explain in Section 3.2, this does not allow achieving optimal throughput.

Protocols targeting high (actually *optimal*) throughput [4, 5] organize processes in a virtual ring topology: each process only communicates with its successor on the ring, using a reliable point-to-point communication protocol: TCP. These protocols achieve significantly higher throughput than protocols targeting low latency, e.g. +25% in a system comprising 4 processes. Nevertheless, these protocols have a significant drawback: because of the ring topology they rely on, latency linearly increases with the number of processes in the system.

In this paper, we present, *FastCast*, the first protocol that achieves both *optimal* throughput[1] and low latency. To achieve low latency, *FastCast* relies on IP multicast. To achieve optimal throughput, *FastCast* allows multiple processes to simultaneously send IP multicast messages. Message ordering is achieved by a fairly classical fixed-sequencer scheme [3]. The novelty in *FastCast* lies in a subprotocol executed by all processes that dynamically computes at which throughput each process can send IP multicast messages.

We have implemented *FastCast* in C++ and have compared its performance to that achieved by two recent state-of-the-art protocols: LCR [5] and RingPaxos [6]. The former achieves optimal throughput, whereas the latter aims at achieving both high throughput and low latency. Our evaluation on a cluster of 8 machines shows that *FastCast* achieves optimal throughput and very low latency. More precisely, *FastCast* achieves up to 86% faster throughput than RingPaxos, and up to 247% lower latency than LCR.

This paper is organized as follows. Section 2 gives a brief overview of the related work. Section 3 presents the *FastCast* protocol. A detailed performance evaluation is provided in Section 4, before concluding the paper in Section 5.

2 Related Work

Various total order broadcast protocols have been devised during the past 30 years [3]. We can distinguish two classes of protocols: those providing *uniform* agreement and those providing *non-uniform* agreement. In uniform agreement protocols, if a process delivers a message, then all correct processes will eventually deliver it. This is not necessarily the case in non-uniform protocols: if a node delivers a message and subsequently fail, the message might not be delivered by remaining (correct) processes. Total order broadcast protocols ensuring

[1] As proved in [5], a total order broacast protocol can only achieve optimal throughput if all processes simultaneously broadcast messages.

uniform agreement are more complex to implement and are often less efficient than non-uniform protocols. Nevertheless, they can be used for a much broader sets of applications. Consequently, the protocol we propose in this paper implements uniform agreement. In the remainder of this section, we do thus put more emphasis on uniform total order broadcast protocols.

Défago and Schiper have written an extensive survey on total order broadcast protocols [3]. They distinguish five types of total order broadcast protocols: fixed-sequencer, moving sequencer, privilege-based, communication history, and destination agreement. As is explained in the survey [3], "communication history" and "destination agreement" protocols [7–16] are less efficient than other types protocols. The three other types of protocols work as follows. In a *fixed sequencer* protocol [6, 17–23], a single process is elected as the sequencer and is responsible for the ordering of messages. The sequencer is unique, and another process is elected as a new sequencer only in the case of sequencer failure. *Moving sequencer* protocols [24–27] 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. Finally, *privilege-based* protocols [28–33] 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. As with moving sequencer protocols, the throughput when all processes broadcast cannot be higher than when only one process broadcasts.

All the protocols mentioned above have been designed with the goal to ensure low broadcast latency. Latency measures the time required to complete a single message broadcast without contention. As shown in [5], above-mentioned protocols are far from sustaining optimal 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 more important 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. The authors of [5] prove that in a system comprising N nodes interconnected by a fully-switched network where each link has a bandwidth of B, the optimal throughput that can be achieved by a total order broadcast protocol is equal to $B * N/(N-1)$. For instance, in a system with 4 nodes interconnected by a gigabit ethernet switch (B=1Gb/s), each node can deliver messages at a throughput of 1,33Gb/s. The

only protocol currently able to sustain that throughput is the LCR protocol [5]. In other protocols, the maximum throughput at which a node can deliver messages is B (1Gb/s in the example taken before). This is for instance the case of protocols known to be efficient such as Spread [33], RingPaxos [6], or the protocol designed by Chang and Maxemchuck [24]. The reason why these protocols do not achieve optimal throughput is that only one node at a time is allowed to broadcast a message. As explained in [5], optimal throughput can only be achieved when all nodes are allowed to simultaneously broadcast messages. Throughput-wise, LCR is thus much more efficient than other protocols. Nevertheless, the throughput-efficiency of LCR comes at a price: latency linearly increases with the number of nodes in the system. This comes from the fact that, in order to sustain high throughput, LCR uses a ring-based pipelining patterns: nodes are organized in a virtual ring. Each node only communicates with its successor in the ring. This pipelining pattern is efficient as it avoids message collisions, but it is not latency-efficient. In this paper, we propose a protocol that reaches optimal throughput, but that achieves a much lower latency than the LCR protocol.

3 The *FastCast* Protocol

In this section, we describe the *FastCast* protocol. We start by a description of the system model we consider. We then give an overview of *FastCast*, followed by a description of the three subprotocols that compose it.

3.1 System Model

We have designed the *FastCast* protocol for small clusters of homogeneous machines interconnected by a local area network. We assume that machines can only fail by crashing (i.e. Byzantine failures are out of the scope of this paper), that crashes are rare, and that each node is equipped with a *perfect* failure detector (P) [34]. A perfect failure detector outputs the list of alive processes and guarantees strong accuracy (correct machines are never suspected to have crashed) and strong completeness (every crash is eventually detected). In order to implement a perfect failure detector, each machine creates a TCP connection to all other machines and maintains this connection during the entire execution of the protocol (unless the machine fails). When a connection fails, the machine tries to re-establish it five times. If the machine does not succeed, it considers that the other machine crashed. This is a reasonable assumption provided that, on a cluster, the latency of the network interconnecting the machines is very low [35].

3.2 Overview

Our goal is to design a uniform total order broadcast protocol achieving optimal throughput, while guaranteeing a low latency. In order to ensure low latency, the best option is to use IP multicast. Indeed, using IP multicast, a process can reach all other processes in the system sending a single message. This choice is natural and

most total order broadcast protocols rely on IP multicast. Unfortunately, IP multicast is not reliable: messages are dropped as soon as the network gets congested.

In order to reduce the ratio of message losses, most state-of-the-art total order broadcast protocols rely on a simple technique: only one process is allowed to send IP multicast messages. That way, it is easy to avoid network congestion by controlling the rate at which the sending process broadcasts IP multicast messages. Unfortunately, using one single sender is not enough to reach optimal throughput. To clarify that point, we depict in Figure 1 a system comprising 3 nodes interconnected by a 1Gb/s ethernet switch. On the left part of the Figure, only one node sends IP multicast messages. The maximum throughput at which nodes of the system can deliver messages in that configuration is 1Gb/s. On the right part of the Figure, we display a configuration where the 3 nodes simultaneously send IP multicast messages. Each node sends at a throughput of 500Mb/s. In that configuration, the maximum throughput at which nodes of the system can deliver messages is equal to 1,5Gb/s: each node delivers 500Mb/s that it produces itself, and 1Gb/s that are sent by other nodes. This is explained by two facts: (i) network cables and Network Interface Cards (NIC) are full-duplex (i.e. a node can simultaneously send and receive messages on the same network cable), and (ii) switches only forward IP multicast messages to nodes other than the source (i.e. a node does not receive its own messages via the network).

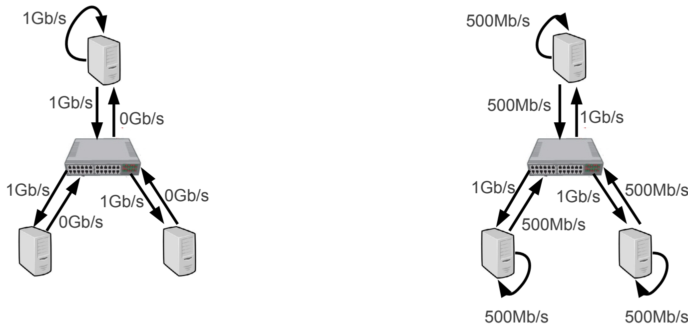

Fig. 1. Multicasting messages (one sender on the left, multiple senders on the right) in a system comprising 3 nodes

As the goal of *FastCast* is to reach optimal throughput while ensuring low latency, the protocol allows multiple processes to simultaneously send IP multicast messages. There are well-known algorithms for ensuring uniform total order of messages multicast by different senders [3]. In this paper, our goal is not to design a new one. Therefore, we take the simplest one, called *fixed-sequencer* protocol (see Sections 3.3 and 3.4 for a short description). Rather, we focus on designing a subprotocol in charge of synchronizing the various senders (see Section 3.5). More precisely, our protocol allows every sender to gather the bandwidth requirements of other senders and to adapt its bandwidth accordingly (using a max-min fair bandwidth allocation algorithm [36]). The idea implemented by the protocol is simple and, as we show in Section 4, yields excellent performance.

3.3 Ordering Subprotocol

FastCast is a uniform total order 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.

The ordering subprotocol implementing these four properties is given in Figure 2. This is a fairly classical fixed-sequencer pattern [3]. One process is designated *leader*, and is in charge of assigning and broadcasting sequence numbers. It is important to notice that the leader is not in charge of forwarding content messages (named DATA message in Figure 2). Rather, these are processes that are in charge of sending their DATA messages to all other processes (line 10). In order to ensure uniform agreement on message delivery, every node acknowledges the reception of the messages and the sequence numbers associated with them (line 19 for the leader, and line 24 for other processes). Every node waits for an acknowledgment from all nodes before delivering a message (lines 30 and 31). That way, a node is sure that the message it delivers is known (together with its sequence number) by all other nodes and will thus be delivered by all correct nodes even if it subsequently fails. Note that to handle message losses, a node that broadcasts a message uses a timer (line 12). If after some amount of time, a node has not delivered its own message (i.e. the message is still in the pendings array as checked in line 35), it resends the message (line 36).

3.4 Membership Management Subprotocol

In order to handle nodes joining and leaving the system, the *FastCast* protocol is built on top of a group communication system [37] relying on a perfect failure detector [34]. Processes are organized into groups, which they can leave or join. When a process joins or leaves a group, this triggers a view change protocol. Thanks to the perfect failure detector, faulty processes are excluded from the group after crashing. Upon a membership change, processes agree on a new view: the current view v_r is replaced by a new view v_{r+1}.

The *view_change* procedure is detailed in Figure 3. Note that when a view change occurs, every process first completes the execution (if any) of all other procedures described in Figure 2. It then freezes those procedures and executes the view change procedure. The latter works as follows (Note that the view change functions make use of two primitives Rsend and Rreceive that implement reliable communication channels. In our implementation, these primitives are implemented using TCP): every process sends its pendings and seqnos arrays to

```
Procedures executed by any process p_i
 1: procedure initialize(initial_view)
 2:     pendings[] ← ∅
 3:     seqnos[] ← ∅
 4:     acks[][] ← ∅
 5:     snToDeliver ← 0
 6:     leader = p_0
 7:     sn ← 0

 8: procedure utoBroadcast(m)
 9:     id_m ← hash(p_i, m)
10:     Send ⟨DATA, id_m, m⟩ to all processes
11:     pendings[id_m] ← m
12:     SetTimeout ⟨id_m⟩

13: upon Receive ⟨DATA, id_m, m⟩ from p_j do
14:     if p_i = leader then
15:         if ∄ seqnos[id_m] then
16:             seqnos[id_m] ← sn
17:             sn ← sn + 1
18:             acks[id_m][p_i] = 1
19:         Send ⟨ACK, id_m, seqnos[id_m]⟩ to all processes
20:     pendings[id_m] ← m
21:     tryDeliver()

22: upon Receive ⟨ACK, id_m, sn_m⟩ from p_j do
23:     if p_j = leader and ∃ pendings[id_m] then
24:         Send ⟨ACK, id_m, sn_m⟩ to all processes
25:         acks[id_m][p_i] = 1
26:     seqnos[id_m] ← sn_m
27:     acks[id_m][p_j] = 1
28:     tryDeliver()

29: procedure tryDeliver()
30:     while ∃ id_m s.t. (seqnos[id_m] = snToDeliver and sum(acks[id_m]) = n) do
31:         utoDeliver(m)
32:         snToDeliver ← snToDeliver + 1
33:         pendings ← pendings − pendings[id_m]

34: upon Timeout⟨id_m⟩ do
35:     if ∃ pendings[id_m] then
36:         Send ⟨DATA, id_m, pendings[id_m]⟩ to all processes
37:         SetTimeout ⟨id_m⟩
```

Fig. 2. Pseudo-code of the ordering mechanism

all other processes (line 2). Upon receiving these arrays, every process updates its own pendings and seqnos arrays using those received from all other processes (lines 15 and 17). Then, the processes send back an ACK_RECOVER message (line 18). Processes wait until they receive ACK_RECOVER messages from all processes (line 3) before sending an END_RECOVERY message to all (line 4). When a process receives END_RECOVERY messages from all processes (line 5), it can deliver all the messages for which it has a sequence number (lines 19 to 24). Thus, at the end of the view change procedure, all processes belonging to the new view will have delivered the same messages in the same order. Each process then empties its pendings, seqnos and acks arrays (lines 8 to 10). Moreover, each process uses as new leader the first process in the new view (line 11).

```
Procedures executed by any process p_i
 1: upon view_change(new_view) do
 2:    Rsend ⟨RECOVER, p_i, pendings, seqnos⟩ to all p_j ∈ new_view
 3:    Wait until received ⟨ACK_RECOVER⟩ from all p_j ∈ new_view
 4:    Rsend ⟨END_RECOVERY⟩ to all p_j ∈ new_view
 5:    Wait until received ⟨END_RECOVERY⟩ from all p_j ∈ new_view
 6:    forceDeliver()
 7:    view ← new_view
 8:    pendings[] ← ∅
 9:    seqnos[] ← ∅
10:    acks[][] ← ∅
11:    leader = first process in view
12:    sn ← nextToDeliver

13: upon Rreceive ⟨RECOVER, pendings_{p_j}, seqnos_{p_j}⟩ from p_j do
14:    for each [id_m] ∈ pendings_{p_j} do
15:       pendings[id_m] ← pendings_{p_j}[id_m]
16:       if ∃ seqnos_{p_j}[id_m] then
17:          seqnos[id_m] ← seqnos_{p_j}[id_m]
18:    Rsend ⟨ACK_RECOVER⟩ to p_j

19: procedure forceDeliver()
20:    for each id_m ∈ seqnos[id_m], ordered by increasing sequence number do
21:       if ∃ pendings[id_m] and seqnos[id_m] ≥ snToDeliver then
22:          toDeliver(pendings[id_m])
23:          pendings ← pendings − pendings[id_m]
24:       snToDeliver ← seqnos[id_m] + 1
25:    for each id_m ∈ keys(pending[id_m]), ordered by increasing id_m do
26:       toDeliver(pendings[id_m])
27:       pendings ← pendings − pendings[id_m]
```

Fig. 3. Pseudo-code of the membership management subprotocol

3.5 Bandwidth Allocation Subprotocol

In this section, we describe the bandwidth allocation protocol implemented in *FastCast*. We start by describing the principles underlying its design. We then comment a detailed pseudo-code. Finally, we give an illustration of its behavior.

Principles. The goal of the bandwidth allocation protocol is to allocate bandwidth for each sending node in order to allow multiple nodes to simultaneously send IP multicast packets, while avoiding message losses. As explained before, having multiple senders is a requirement to ensure that the full network capability is used. If we assume that at a given time, all nodes know the bandwidth requirements of all other nodes, it is easy to allocate bandwidth using a max-min fair bandwidth allocation algorithm [36]. For instance, let us consider a system comprising 3 nodes interconnected by a 1Gb/s ethernet switch. Let us assume that each node knows that, e.g. node 1 requires 700Mb/s, node 2 requires 600Mb/s, and node 3 requires 300Mb/s. Each node can deterministically compute the following fair bandwidth allocation: 500Mb/s for nodes 1 and 2, and 300Mb/s for node 3. It is indeed not possible to allocate more than 500Mb/s to nodes 1 and 2. Otherwise, node 3 would have to receive messages at a higher throughput than 1Gb/s, which it cannot do. Indeed, the network link connecting node 3 to the switch has a capability of 1Gb/s.

It is possible to design a protocol allowing nodes to exchange their bandwidth requirements and ensuring that every node knows, at any time, the bandwidth requirements of other nodes. Such a protocol would nevertheless be costly and would require to force all nodes to synchronize whenever one node wants to change its bandwidth. Interestingly, it is possible to fairly allocate bandwidth with a weaker requirement: it is enough that every node receive the various bandwidth requirements from other nodes in the same order. This property can be easily achieved by leveraging the *FastCast* protocol itself. Each time a node wants to modify its allocated bandwidth (e.g. to increase it, or to decrease it), it sends a message to all other nodes using the *FastCast* protocol. That way, all nodes receive the bandwidth requirement messages in the same order.

The question that remains to answer is: when can nodes actually modify their bandwidth? A node behaves differently depending on whether it requires a decrease of its bandwidth or an increase of its bandwidth. In the case of a bandwidth decrease, the node actually decreases its bandwidth before sending the message notifying other nodes. That way, when other nodes receive its notification message, they know that the node already decreased its bandwidth and they can recompute the bandwidth allocation and possibly decide to increase their own bandwidth. In the case of a bandwidth increase, a node n cannot directly increase its bandwidth (otherwise, that could congest the network). The node does thus first send the message notifying others that it wants to increase its bandwidth. Upon receiving the notification that node n wants to increase its bandwidth, other nodes locally recompute the bandwidth allocation (based on the new bandwidth requirement sent by node n) and possibly reduce their own bandwidth. Then, each node sends an acknowledgement to node n. It is only after it has received acknowledgments from all other nodes that node n can actually increase its bandwidth (by locally computing the bandwidth allocation).

Detailed Pseudo-Code. Figure 4 gives the pseudo-code of the bandwidth allocation protocol. Every node stores the bandwidth requirements of other nodes in the bwRequirements array and its current bandwidth in the currentBW variable. The ongoing_increase, delivered_req, and acks fields are used when a node wants to increase its bandwidth: ongoing_increase stores the required increase (before being stored in bwRequirements when all other processes will have acknowledged it); the delivered_req field indicates whether the increase notification message has been delivered by the requiring node itself (if that is not the case, the requiring node cannot take its own request into account even if it received an acknowledgement from all other processes); finally, the acks field is used to count the number of acknowledgements that have been received for the ongoing bandwidth increase request.

Before going into the details of the protocol, let us remark that the BW_allocation function (lines 34 to 47) implements a classical max-min fair bandwidth allocation algorithm [36]. The only important point to mention is that it uses a variable, called availableBW, that represents the maximum capability of a network link. This capability is dependent from the average message size (it is well-known that the larger the messages, the higher the throughput that can be

Procedures executed by any process p_i

```
 1: procedure initialize(initial_view)
 2:     bwRequirements[] ← [0, ··· , 0]
 3:     currentBW ← 0
 4:     ongoing_increase ← 0
 5:     delivered_req ← false
 6:     acks ← 0

 7: procedure increase_BW(amount)
 8:     wait until ongoing_increase = 0
 9:     ongoing_increase ← amount
10:     utoBroadcast ⟨INCR, amount⟩ to all processes

11: upon utoDeliver ⟨INCR, amount⟩ from p_j ≠ p_i do
12:     bwRequirements[p_j] ← bwRequirements[p_j] + amount
13:     currentBW ← BW_allocation()
14:     Rsend ⟨ACK⟩ to p_j

15: upon utoDeliver ⟨INCR, amount⟩ from p_i do
16:     delivered_req ← true

17: upon Rreceive ⟨ACK⟩ from p_j do
18:     acks ← acks + 1
19:     if acks = N − 1 then
20:         wait until delivered_req = true
21:         bwRequirements[p_i] ← bwRequirements[p_i] + ongoing_increase
22:         currentBW ← BW_allocation()
23:         acks ← 0
24:         ongoing_increase ← 0
25:         delivered_req ← false

26: procedure decrease_BW(amount)
27:     wait until ongoing_increase = 0
28:     bwRequirements[p_i] ← bwRequirements[p_i] − amount
29:     currentBW ← BW_allocation()
30:     utoBroadcast ⟨DECR, amount⟩ to all processes

31: upon utoDeliver ⟨DECR, amount⟩ from p_j ≠ p_i do
32:     bwRequirements[p_j] ← bwRequirements[p_j] − amount
33:     currentBW ← BW_allocation()

34: function BW_allocation()
35:     nodes ← p_i and the (N-2) other iggest values in bwRequirements
36:     availableBW ← B
37:     do
38:     allocated = false
39:     for p_j in nodes do
40:         if bwRequirements[p_j] ≤ availableBW/size(nodes) then
41:             nodes ← nodes − p_j
42:             availableBW ← availableBW − bwRequirements[p_j]
43:             allocated = true
44:     while(nodes ≠ ∅ and allocated = true)
45:     if p_i ∈ nodes then
46:         return availableBW/size(nodes)
47:     return bwRequirements[p_i]
```

Fig. 4. Pseudo-code of the bandwidth allocation protocol

achieved by a communication protocol [5,6]). In our implementation, we use 4kB as the average message size and set the value of availableBW to the capability that the network links exhibit when used with 4kB messages (this capability is close to the optimal one). To be sure that this is the actual capability that network links will have at runtime, the *FastCast* protocols batches messages to ensure that sent messages are at least 4kB large (unless there is no contention, in which case small messages can be sent as the protocol does not need to sustain high throughput in such cases).

Let us now describe the bandwidth allocation subprotocol. A node can either ask to increase its bandwidth (using the increase_BW procedure at line 7) or to decrease it (using the decrease_BW procedure at line 26). Let us first describe what happens when a node wants to increase its bandwidth. The node calls the increase_BW procedure. Inside this procedure, the node utoBroadcasts

Table 1. A first example execution of the bandwidth allocation protocol

step	*process*	*bwRequirements*	*currentBW*	*ongoing_increase*	*acks*	*delivered_req*	
S1	p_0	$[0,0,0]$	0	0	0	-	Initial state
	p_1	$[0,0,0]$	0	0	0	-	
	p_2	$[0,0,0]$	0	0	0	-	
S2	p_0	$[0,0,0]$	0	**800**	0	-	p_0 calls increase_BW(800)
	p_1	$[0,0,0]$	0	**300**	0	-	p_1 calls increase_BW(300)
	p_2	$[0,0,0]$	0	0	0	-	
S3	p_0	$[0,0,0]$	0	800	0	-	p_2 utoDelivers $\langle \text{INCR}, 800\rangle_{p_0}$
	p_1	$[0,0,0]$	0	300	0	-	p_2 utoDelivers $\langle \text{INCR}, 300\rangle_{p_1}$
	p_2	$[\mathbf{800},\mathbf{300},0]$	0	0	0	-	
S4	p_0	$[0,0,0]$	0	800	**1**	-	p_0 Rreceives $\langle \text{ACK}\rangle_{p_2}$
	p_1	$[0,0,0]$	0	300	**1**	-	p_1 Rreceives $\langle \text{ACK}\rangle_{p_2}$
	p_2	$[800,300,0]$	0	0	0	-	
S5	p_0	$[0,\mathbf{300},0]$	0	800	1	√	p_0 utoDelivers $\langle \text{INCR}, 800\rangle_{p_0}$
	p_1	$[0,0,0]$	0	300	1	-	p_0 utoDelivers $\langle \text{INCR}, 300\rangle_{p_1}$
	p_2	$[800,300,0]$	0	0	0	-	
S6	p_0	$[0,300,0]$	0	800	1	√	p_1 Rreceives $\langle \text{ACK}\rangle_{p_0}$
	p_1	$[0,0,0]$	0	300	**2**	-	
	p_2	$[800,300,0]$	0	0	0	-	
S7	p_0	$[0,300,0]$	0	800	1	√	p_1 utoDelivers $\langle \text{INCR}, 800\rangle_{p_0}$
	p_1	$[\mathbf{800},\mathbf{300},0]$	**300**	**0**	**0**	-	p_1 utoDelivers $\langle \text{INCR}, 300\rangle_{p_1}$
	p_2	$[800,300,0]$	0	0	0	-	
S8	p_0	$[\mathbf{800},300,0]$	**700**	**0**	**0**	**-**	p_0 Rreceives $\langle \text{ACK}\rangle_{p_1}$
	p_1	$[800,300,0]$	300	0	0	-	
	p_2	$[800,300,0]$	0	0	0	-	

Table 2. A second example execution of the bandwidth allocation protocol

step	*process*	*bwRequirements*	*currentBW*	*ongoing_increase*	*acks*	*delivered_req*	
S9	p_0	$[800, 300, 0]$	700	0	0	-	Initial state (equal to S8 in Table 1)
	p_1	$[800, 300, 0]$	300	0	0	-	
	p_2	$[800, 300, 0]$	0	0	0	-	
S10	p_0	$[800, 300, 0]$	700	0	0	-	p_2 calls increase_BW(600)
	p_1	$[800, 300, 0]$	300	0	0	-	
	p_2	$[800, 300, 0]$	0	**600**	0	-	
S11	p_0	$[800, 300, \mathbf{600}]$	**500**	0	0	-	p_0 utoDelivers $\langle \text{INCR}, 600\rangle_{p_2}$
	p_1	$[800, 300, \mathbf{600}]$	**300**	0	0	-	p_1 utoDelivers $\langle \text{INCR}, 600\rangle_{p_2}$
	p_2	$[800, 300, 0]$	0	600	0	$\surd$	p_2 utoDelivers $\langle \text{INCR}, 600\rangle_{p_2}$
S12	p_0	$[800, 300, 600]$	500	0	0	-	p_2 Rreceives $\langle \text{ACK}\rangle_{p_0}$
	p_1	$[800, 300, 600]$	300	0	0	-	
	p_2	$[800, 300, 0]$	0	600	1	$\surd$	
S13	p_0	$[800, 300, 600]$	500	0	0	-	p_2 Rreceives $\langle \text{ACK}\rangle_{p_1}$
	p_1	$[800, 300, 600]$	300	0	0	-	
	p_2	$[800, 300, \mathbf{600}]$	**500**	**0**	**0**	-	

an INCR message to all other processes (line 10). When delivering this message, other processes update their bwRequirements array (line 12), recompute the bandwidth allocation (line 13) using the BW_allocation function, and sends an ACK message back to the requiring process (line 14). When the requiring node has both received an acknowledgement from all other nodes and delivered its own increase request (line 16), it updates its bwRequirements array (line 21) and recompute the bandwidth allocation (line 22).

Let us now describe what happens when a node wants to decrease its bandwidth. The node calls the decrease_BW procedure. Inside this procedure, the node updates its bwRequirements array (line 28) and recompute the bandwidth allocation (line 29). The requiring node then utoBroadcasts a DECR message to all other processes (line 30). When delivering this message, other processes update their bwRequirements array (line 32) and recompute the bandwidth allocation (line 33), using the BW_allocation function.

Illustration. We provide three illustrations of the bandwidth allocation protocol in Table 1, Table 2, and Table 3. We consider a system with 3 processes interconnected by a 1Gb/s switch. In each table, we describe a set of steps that happen in the system and we illustrate how the different fields of the three processes are updated. Initially, the three processes have a null bandwidth (currentBW is equal to 0 in Table 1, step S1). In Table 1 we depicts what happens when from this initial

Table 3. A third example execution of the bandwidth allocation protocol

step	*process*	*bwRequirements*	*currentBW*	*ongoing_increase*	*acks*	*delivered_req*	
S14	p_0	[800, 300, 600]	500	0	0	-	Initial state (equal to S13 in Table 2)
	p_1	[800, 300, 600]	300	0	0	-	
	p_2	[800, 300, 600]	500	0	0	-	
S15	p_0	[800, 300, 600]	500	0	0	-	p_2 calls decrease_BW(500)
	p_1	[800, 300, 600]	300	0	0	-	
	p_2	[800, 300, **100**]	**100**	0	0	-	
S16	p_0	[800, 300, **100**]	**700**	0	0	-	p_0 utoDelivers $\langle \text{DECR}, 500 \rangle_{p_2}$
	p_1	[800, 300, **100**]	**300**	0	0	-	p_1 utoDelivers $\langle \text{DECR}, 500 \rangle_{p_2}$
	p_2	[800, 300, 100]	100	0	0	-	p_2 utoDelivers $\langle \text{DECR}, 500 \rangle_{p_2}$

state, p_0 calls increase_BW(800) and p_1 calls increase_BW(300). Processes reach a state (step S8) in which p_0 has its currentBW variable equal to 700Mb/s and p_1 has its currentBW variable equal to 300Mb/s. From that state (also depicted in Table 2, step S9), Table 2 depicts what happens when p_2 calls increase_BW(600). Processes reach a state (step S13) in which p_0 and p_2 both have their currentBW variable equal to 500Mb/s, and p_1 has its currentBW variable equal to 300Mb/s. From that state (also depicted in Table 3, step S14), Table 3 depicts what happens when p_2 calls decrease_BW(500). Processes reach a state (step S16) in which p_0 has its currentBW variable equal to 700Mb/s, p_1 has its currentBW variable equal to 300Mb/s and p_2 has its currentBW variable equal to 100Mb/s.

4 Performance Evaluation

In this section, we assess the performance of the *FastCast* protocol and compare them to that achieved by two state-of-the-art protocols: LCR [5] and RingPaxos [6]. All three protocols ensure uniform total order delivery of messages. We chose LCR because it is the only existing protocol ensuring optimal throughput [5]. Moreover, the choice of RingPaxos is motivated by the fact, as shown in [6], it is the only protocol to "*achieve very high throughput while providing low latency*". The experiments only evaluate the failure free case because failures are expected to be very rare in the targeted environment. Note that in the faulty case, the performance of *FastCast* would be very similar to that of LCR provided that both protocols implement almost similar recovery algorithms. LCR and *FastCast* relies on the use of a perfect failure detector, whereas RingPaxos assumes a bound on the number of faulty processes.

We start by a description of the experimental setup. We then assess the bandwidth allocation protocol of *FastCast*, and the throughput, the response time, and the latency of *FastCast*, LCR and RingPaxos. Our evaluation shows that *FastCast* is both throughput- and latency-efficient. More precisely, throughput-wise, *FastCast* is as efficient as LCR. Latency-wise, *FastCast* is more efficient than RingPaxos.

4.1 Experimental Setup

The experiments were run on a cluster comprising eight 8-core machines interconnected by a gigabit ethernet switch. Each core runs at 2.5GHz and is equipped with 16GB of RAM. Moreover, each machine runs a Linux 2.6.32 kernel. The raw bandwidth over IP between two machines (measured with Netperf [38]) is equal to 942Mb/s. In order to ensure that the evaluation is fair, we have implemented the *FastCast* and LCR protocols in C++, using the same code base as the RingPaxos protocol. Finally, all the presented experiments start with a warm-up phase, followed by a phase during which performance are measured. The measurement phase lasts 5 minutes.

4.2 Bandwidth Allocation Assessment

We first assess the bandwidth allocation protocol implemented in *FastCast*. For that purpose, we perform the following experiment. We deploy 4 nodes that send messages of variable sizes: from 1kB to 6kB. The bandwidth requirements of nodes vary during the experiment: initially all nodes require one fourth of the total available bandwdith. After $10s$, node 0 decreases its requirements, followed by node 1 at time $20s$. At time $30s$, node 2 increases its bandwidth requirement. Finally, at time $40s$, node 0 increases its bandwidth requirement, whereas node 2 decreases them. The results are depicted in Figure 5. The X axis represents the time, whereas the Y axis is used to represent the bandwidth requirements of the 4 nodes, as well as the achieved and optimal throughput. We observe that the achieved throughput is very close to the optimal one, thus confirming that the bandwidth allocation protocol works efficiently. Moreover, we have used that experiment to assess the time it takes for a node to increase its bandwidth, i.e. the time that elapses between the moment when the node notifies other nodes that it has new bandwidth requirements and the moment when the node is allowed to increase its bandwidth. We have run that experiment multiple times and the average time required by the different nodes to increase their bandwidth was $3.8ms$.

4.3 Throughput Assessment

To assess the throughput of the three protocols, we run the following benchmark: we deploy N nodes that broadcast messages at the maximum throughput they can sustain. The message size is fixed and set to 10kB, which allows reaching the best possible throughput for each studied protocol. Each process periodically computes the throughput at which it delivers messages. In this experiment, the

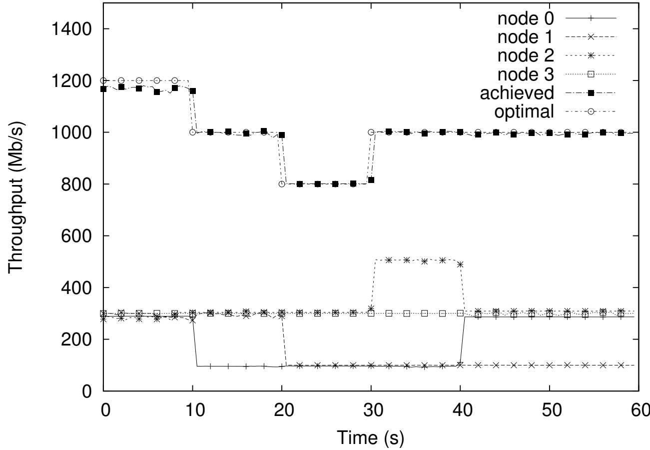

Fig. 5. Assessment of *FastCast*'s bandwidth allocation protocol

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.

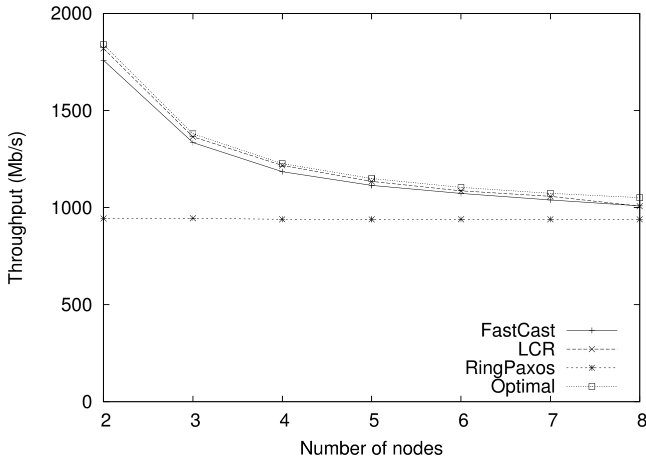

Fig. 6. Throughput as a function of the number of nodes in the system for the *FastCast*, LCR, and RingPaxos protocols

Figure 6 plots the throughput achieved by *FastCast*, LCR and RingPaxos when varying the number of nodes from 2 to 8. As reference, we plot the optimal throughput that can be achieved by ($N/(N-1)$ times the maximum link speed of 942Mb/s). We can make several observations. First, the throughput of *FastCast* and LCR is very close to optimal. As mentioned in the previous section, this confirms the fact that the bandwidth allocation algorithm works efficiently. Second, the throughput of RingPaxos is almost constant (at 939Mb/s). Again,

this behavior is expected: in RingPaxos, only one process at a time is allowed to send IP multicast messages. This limits the throughput that can be sustained by the protocol. For instance, with 4 nodes, *FastCast* and LCR are about 25% faster than RingPaxos. In a system with 2 nodes, *FastCast* and LCR are about 86% faster than RingPaxos.

4.4 Response Time Assessment

In this section, we evaluate the response time of *FastCast*, LCR, and RingPaxos in a system comprising 8 nodes. In this experiment, we vary the throughput at which the nodes inject new messages in the system. The size of messages that are broadcast is 10kB. During the measurement phase, for every message m it broadcasts, a 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.

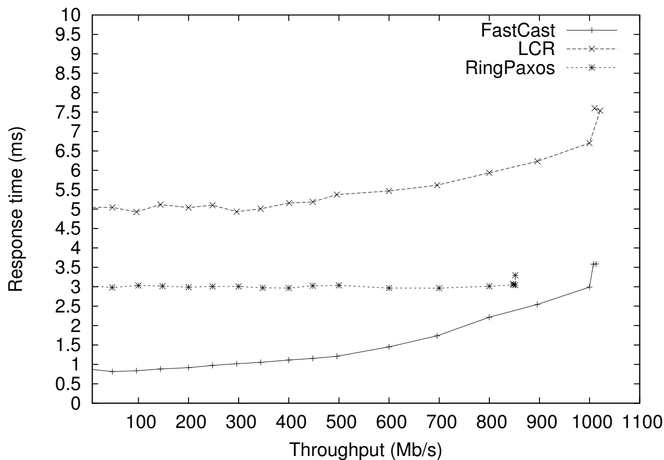

Fig. 7. Response time as a function of the aggregated sending throughput for the *FastCast*, LCR, and RingPaxos protocols

Results are depicted in Figure 7. The X axis represents the aggregated sending throughput, whereas the Y axis represents the response time. We observe that *FastCast* exhibits a consistently lower response time than both LCR and RingPaxos. More precisely, *FastCast* achieves an up to 400% lower response time than LCR and an up to 246% lower response time than RingPaxos. This comes from the fact that both LCR and RingPaxos rely on a ring topology for sending some of the messages that are exchanged among nodes: data messages in the case of LCR, and ordering messages in the case of RingPaxos (notice that, unlike in LCR, in RingPaxos, not all processes are organized in a ring [6]). The pipelining pattern introduced by a ring topology increases the time it takes to process each message with respect to a pure IP multicast protocol such as *FastCast* in which no pipelining pattern is used.

4.5 Latency Assessment

In this section, we evaluate the latency achieved by the *FastCast*, LCR, and RingPaxos protocols. We vary the size of the system from 2 to 8 nodes. Recall that latency is defined as the time required to complete a message broadcast when there is no contention. In order to measure the latency of the various protocols, we perform the following experiment: one node in the system broadcasts 10kB messages at a very low throughput (1Mb/s). The sending node evaluates the average time that elapses between the broadcast of each message and its delivery.

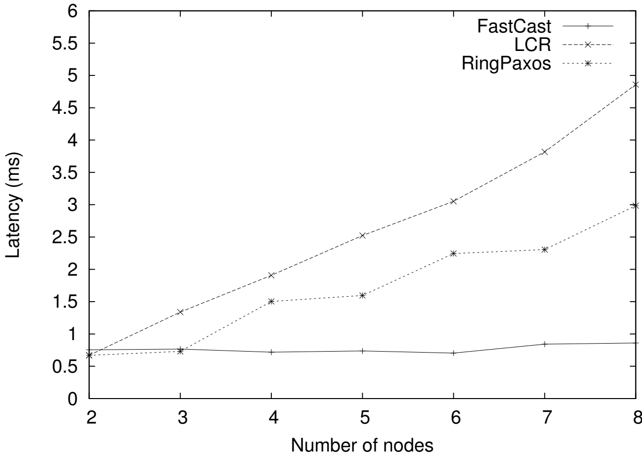

Fig. 8. Latency as a function of the number of nodes in the system for the *FastCast*, LCR, and RingPaxos protocols

Results are depicted in Figure 8. The X axis represents the number of nodes, whereas the Y axis represents the latency. We observe that *FastCast* exhibits a consistently lower latency than both LCR and RingPaxos. More precisely, *FastCast* achieves an up to 465% lower latency than LCR and an up to 247% lower latency than RingPaxos. Moreover, we observe that the latency of *FastCast* is constant, whereas that of RingPaxos and LCR increases with the number of nodes. This again comes from the fact that both LCR and RingPaxos rely on a ring topology for sending some of the messages. The reason why the curve for RingPaxos is not linear is that in RingPaxos only a majority of nodes need to be present in the ring. For instance, RingPaxos uses the same ring size (3) for systems comprising 4 and 5 nodes, whereas in LCR, the ring size linearly increases with the number of nodes in the system.

5 Conclusion

We have presented *FastCast*, a uniform total order broadcast protocol that achieves both optimal throughput and very low latency. Unlike previous

throughput-optimal protocols, *FastCast* does not rely on a ring topology for message dissemination. Rather, *FastCast* uses IP multicast, a low-level communication protocol that allows reaching multiple processes using a single message. To avoid network congestion (and thus IP multicast packet drops), *FastCast* implements a subprotocol in charge of dynamically computing the throughput at which processes are allowed to send IP multicast messages. We have evaluated *FastCast* on a cluster of 8 machines and have compared its performance to that achieved by two recent state-of-the-art protocols: LCR and RingPaxos. The evaluation shows that *FastCast* achieves optimal throughput and very low latency.

Currently, *FastCast* assumes that it is the only source of network traffic. In our future work, we plan to study extensions of *FastCast* to take into account background traffic. Our intuition is that a possible approach is to have all applications running on a set of nodes share the same bandwidth allocation mechanism.

Acknowledgements. We would like to thank Baptiste Lepers and the anonymous reviewers for their useful feedback on this work. Moreover, the presented work has been funded by the French ANR project called SocEDA (`http://www.soceda.org`) and by the EU FP7 Specific Targeted Research Project "PLAY" (`http://www.play-project.eu`).

References

1. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22(4), 299–319 (1990)
2. Hadzilacos, V., Toueg, S.: Fault-tolerant broadcasts and related problems, pp. 97–145 (1993)
3. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv. 36(4), 372–421 (2004)
4. Guerraoui, R., Levy, R.R., Pochon, B., Quéma, V.: High Throughput Total Order Broadcast for Cluster Environments. In: IEEE International Conference on Dependable Systems and Networks (DSN 2006), Philadelphia, PA, USA (2006)
5. Guerraoui, R., Levy, R.R., Pochon, B., Quéma, V.: Throughput optimal total order broadcast for cluster environments. ACM Trans. Comput. Syst. 28(2), 5:1–5:32 (2010), `http://doi.acm.org/10.1145/1813654.1813656`
6. Marandi, P., Primi, M., Schiper, N., Pedone, F.: Ring paxos: A high-throughput atomic broadcast protocol. In: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 527–536 (2010)
7. Peterson, L., Buchholz, N., Schlichting, R.: Preserving and using context information in interprocess communication. ACM Trans. Comput. Syst. 7(3), 217–246 (1989)
8. Malhis, L., Sanders, W., Schlichting, R.: Numerical performability evaluation of a group multicast protocol. Distrib. Syst. Enj. J. 3(1), 39–52 (1996)
9. Ezhilchelvan, P., Macedo, R., Shrivastava, S.: Newtop: a fault-tolerant group communication protocol. In: Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS 1995). IEEE Computer Society, Washington, DC (1995)

10. Ng, T.: Ordered broadcasts for large applications. In: Proceedings of the 10th IEEE International Symposium on Reliable Distributed Systems (SRDS 1991), pp. 188–197. IEEE Computer Society, Pisa (1991)
11. Moser, L., Melliar-Smith, P., Agrawala, V.: Asynchronous fault-tolerant total ordering algorithms. SIAM J. Comput. 22(4), 727–750 (1993)
12. Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)
13. Birman, K., Joseph, T.: Reliable communication in the presence of failures. ACM Trans. Comput. Syst. 5(1), 47–76 (1987)
14. Luan, S., Gligor, V.: A fault-tolerant protocol for atomic broadcast. IEEE Trans. Parallel Distrib. Syst. 1(3), 271–285 (1990)
15. Fritzke, U., Ingels, P., Mostefaoui, A., Raynal, M.: Consensus-based fault-tolerant total order multicast. IEEE Trans. Parallel Distrib. Syst. 12(2), 147–156 (2001)
16. Anceaume, E.: A lightweight solution to uniform atomic broadcast for asynchronous systems. In: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS 1997). IEEE Computer Society, Washington, DC (1997)
17. Kaashoek, F., Tanenbaum, A.: An evaluation of the amoeba group communication system. In: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS 1996). IEEE Computer Society, Washington, DC (1996)
18. Armstrong, S., Freier, A., Marzullo, K.: Multicast transport protocol. RFC 1301, IETF (1992)
19. Carr, R.: The tandem global update protocol. Tandem Syst. Rev. 1, 74–85 (1985)
20. Garcia-Molina, H., Spauster, A.: Ordered and reliable multicast communication. ACM Trans. Comput. Syst. 9(3), 242–271 (1991)
21. Birman, K., van Renesse, R.: Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press (1993)
22. Wilhelm, U., Schiper, A.: A hierarchy of totally ordered multicasts. In: Proceedings of the 14th Symposium on Reliable Distributed Systems. IEEE Computer Society, Washington, DC (1995)
23. Ban, B.: JGroups – A Toolkit for Reliable Multicast Communication (2007), `http://www.jgroups.org`
24. Chang, J.-M., Maxemchuk, N.: Reliable broadcast protocols. ACM Trans. Comput. Syst. 2(3), 251–273 (1984)
25. Whetten, B., Montgomery, T., Kaplan, S.: A high performance totally ordered multicast protocol. In: Birman, K.P., Mattern, F., Schiper, A. (eds.) Theory and Practice in Distributed Systems. LNCS, vol. 938, pp. 33–57. Springer, Heidelberg (1995)
26. Kim, J., Kim, C.: A total ordering protocol using a dynamic token-passing scheme. Distrib. Syst. Eng. J. 4(2), 87–95 (1997)
27. Cristian, F., Mishra, S., Alvarez, G.: High-performance asynchronous atomic broadcast. Distrib. Syst. Eng. J. 4(2), 109–128 (1997)
28. Friedman, T., Renesse, R.V.: 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 1997). IEEE Computer Society, Washington, DC (1997)
29. Cristian, F.: Asynchronous atomic broadcast. IBM Technical Disclosure Bulletin 33(9), 115–116 (1991)

30. Ekwall, R., Schiper, A., Urban, P.: Token-based atomic broadcast using unreliable failure detectors. In: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS 2004), pp. 52–65. IEEE Computer Society, Washington, DC (2004)
31. Amir, Y., Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Ciarfella, P.: The Totem single-ring ordering and membership protocol. ACM Transactions on Computer Systems 13(4), 311–342 (1995)
32. Gopal, A., Toueg, S.: Reliable broadcast in synchronous and asynchronous environments (preliminary version). In: Bermond, J.-C., Raynal, M. (eds.) WDAG 1989. LNCS, vol. 392, pp. 110–123. Springer, Heidelberg (1989)
33. Amir, Y., Danilov, C., Miskin-Amir, M., Schultz, J., Stanton, J.: The spread toolkit: Architecture and performance. CNDS-2004-1, Johns Hopkins University, Tech. Rep. (2004)
34. Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
35. Dunagan, J., Harvey, N.J.A., Jones, M.B., Kostic, D., Theimer, M., Wolman, A.: Fuse: Lightweight guaranteed distributed failure notification. In: Proceedings of 6th Symposium on Operating Systems Design and Implementation, OSDI 2004 (2004)
36. Le Boudec, J.-Y.: Rate adaptation, congestion control and fairness: A tutorial. Ecole Polytechnique Fédérale de Lausanne (2012)
37. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. In: Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP 1987), pp. 123–138. ACM Press, New York (1987)
38. Jones, R.: Netperf (2007), http://www.netperf.org/