

Semantically Reliable Broadcast: Sustaining High Throughput in Reliable Distributed Systems

José Pereirai
U. Minho

L. Rodrigues
U. Lisboa

R. Oliveira
U. Minho

June 26, 2009

Abstract

Replicated services are often required to sustain high loads of multiple concurrent requests. This requirement is hard to balance with strong consistency. Typically, to ensure inter-replica consistency, all replicas should receive all updates. Unfortunately, in this case, a single slow replica may degrade the performance of the whole system. This paper proposes a novel reliable broadcast primitive that uses semantic knowledge to weaken reliable delivery guarantees while, at the same time, ensuring strong consistency at the semantic level. By allowing some obsolete messages to be dropped, the protocol that implements this primitive is able to sustain a higher throughput than a fully reliable broadcast protocol. The usefulness of the primitive and the performance of the protocol are illustrated through a concrete example.

Introduction

Replication is a widely used technique to implement reliable services. One of the most important primitives to support replication is reliable broadcast [6]. Intuitively, reliable broadcast ensures that all replicas receive the same messages and, therefore, have the information required to reach a consistent state. Unfortunately, reliable broadcast is also an expensive primitive, since messages need to be stored to ensure recovery in the case of network omissions. In systems with many concurrent clients, the replicated servers have to cope with a high throughput of requests, and this may quickly exhaust the memory available for storing messages. Therefore, it is a hard task to balance the need to sustain a high input of concurrent requests and the need to keep the replicas consistent.

This paper proposes a novel reliable broadcast primitive to address this problem. This primitive offers a weak form of reliability, as it allows some messages to be dropped by the protocol. This eases the task of sustaining high throughput by alleviating the memory requirements imposed by stored messages. However, instead of simply dropping messages at random, a strategy that would compromise

the consistency of the replicated service, our primitive uses semantic knowledge on the contents of the message, to selectively drop only messages that have become obsolete while in transit. The primitive ensures that, at the semantic level, all replicas receive the same up-to-date information and are therefore guaranteed to have the information needed to preserve inter-replica consistency.

The new primitive is called FIFO Semantic Reliable Broadcast (FIFO-SRB), and it is an extension to our previous work with semantic reliability [9]. While our previous work motivated the need for semantic reliability in point-to-multipoint streams by showing that in some applications it cannot be avoided that many messages become obsolete while still in transit, without concern for inter replica consistency, in this paper we make the following contributions: We present a specification of the FIFO-SRB primitive, present a protocol to implement FIFO-SRB and, finally, show using a concrete example how FIFO-SRB can be used to support the replication of services with many concurrent clients.

The rest of the paper is structured as follows: Section 1 motivates the need to weaken reliability to sustain high throughput. Section 2 introduces the formal definition of the broadcast primitive and an algorithm implementing it. Section 3 shows how to apply the primitive in strongly consistent replication and illustrates the benefits with performance measurements. Section 4 concludes the paper.

1 Weakening Reliability to Sustain High Throughput

This section motivates our work by presenting the problem of throughput stability in applications based on reliable broadcast communication protocols. Then we briefly examine existing proposals and outline intuition underlying semantic reliability.

1.1 Throughput Stability

The difficulty with sustaining high throughput has been identified as one of the limiting factors for the scalability of reliable broadcast protocols [11]. This is unfortunate as throughput stability is a requirement for many demanding applications that would benefit from reliable broadcast primitives [2]. In the following paragraphs we explain why this feature is inherent to any fully reliable broadcast protocol.

A reliable broadcast protocol must ensure the delivery of a message to all correct recipients, despite the occurrence of faults. Typical faults to be considered are omissions in the links and crashes of processes. For instance, a sender may crash while transmitting a message. This may cause a message to be delivered to some of the recipients but not to the others. To ensure recovery, processes need to store messages for the case retransmissions are needed. These messages must not be discarded until one is sure the message has been received by all processes (at that point, the message is said to be *stable*). Furthermore, messages need also to be stored until consumed by the application.

To sustain a high input rate from senders, one must ensure that: *i*) enough memory is available to store incoming messages until previous messages have been delivered; *ii*) messages are consumed at the same pace new messages arrive. If these conditions are not verified, memory is quickly exhausted and clients are prevented from obtaining further service. Unfortunately, in order for a message to be discarded from buffers, it has to be acknowledged by all replicas. This means that if a single replica is slower (for instance, due to a transient overload of the hosting node) all the group is affected.

The typical solution to cope with a replica that is continuously slower than the remaining replicas is to consider that the replica is faulty, and exclude it from the group of replicas. However, if one wants to preserve a given replication degree, a new replica needs to be added to the group in order to replace the excluded replica, and this typically requires the execution of an expensive integration procedure. Due to the costs of excluding a replica, it is advisable to not exclude replicas that are subject to transient overloads. On the other hand, even if no single replica is continuously slow, different replicas may exhibit transient overloads at different moments always impacting all the group, preventing the replicated service from sustaining a high throughput of requests.

1.2 Relaxing Reliability

One approach to address the throughput problem described above is to weaken the reliability requirements of the broadcast primitive, such that slower recipients are not required to deliver all messages. This would allow messages to be purged earlier from the retransmission buffers and would prevent a single process from slowing down the entire group of replicas.

Two examples of this approach can be found in literature. Bimodal multicast [3] offers probabilistic reliability guarantees which do not hold for slower processes that fail to meet performance assumptions. Application Level Framing (ALF) [5] does not perform automatic retransmissions, and requires the receivers to explicitly request retransmissions of lost messages that are considered relevant.

Both these approaches introduce a significant complexity to be managed by the application. If a message loss compromises the correctness of a server, and the message is no longer available for retransmission, the server may be forced to exclude itself from the group and rejoin later in order to get a correct copy of the state. Notice that even if some mechanism is implemented to notify the receiver that some messages have been dropped, the application might be unable to take any corrective measure since it has no knowledge of that message's content and thus cannot evaluate whether the unknown message is relevant. This last problem can be circumvented by the use of two multicast protocols in parallel [12]: An unreliable protocol used for payload and a reliable protocol used to convey metadata describing the content of data messages sent on the payload channel. Using information from the control channel, the receiver may evaluate the relevance of lost messages in the payload channel and explicitly request retransmission when

needed.

Our work is also inspired in the Δ -causal [1] and deadline constrained [13] causal protocols. These protocols allow real-time constraints to be met at the cost of discarding delayed messages.

1.3 Message Obsolescence

Our goal is to avoid the increased complexity of the previous approaches while retaining the assumption that some messages do not need to be re-transmitted if lost. The approach derives from the observation that some messages implicitly convey or overwrite the content of other messages sent in the past, therefore making these old messages irrelevant. Obsolete information can be then safely purged from re-transmission buffers as soon as newer messages, that overwrite the contents of obsolete messages, are safely available.

By immediately purging obsolete messages, the freed resources can be allocated to the remaining messages. Thus it is possible to accommodate receivers with different capacities within the same group. The resulting reliability criterion is *semantic reliability*, as all current information is delivered to all receivers, either implicitly or explicitly, without necessarily delivering all messages.

In order to provide information to the communication protocol about which messages are related, the application has simply to tag each message with a label that conveys information about the obsolescence relation. For instance, a label can be associated with each data item managed by the application: Two messages that overwrite the value of the same data item would carry the same label. Using simple labeling schemes it is possible for the protocol to manage purging tasks in an efficient manner.

The notion of semantic reliability has been previously proposed in [9]. In that paper we have studied the performance of message obsolescence relations in point-to-multipoint channels. As it is expected, the performance of the approach is mostly dependent of the degree of obsolescence of the traffic generated by the application. Fortunately, applications exhibiting high throughput due to rapidly changing data tend to exhibit meaningful obsolescence rates as recent values make older ones obsolete. Such applications range from multiplayer games to distributed control, and on-line transaction processing. We have also considered the semantic reliability in the context of probabilistic multicast protocols [10]. However, we have not addressed previously applications with inter-replica consistency requirements. Such requirements are taken into account only in FIFO-SRB and associated protocol that are presented in the following sections.

2 Semantically Reliable Broadcast

In this section we present and discuss the specification of FIFO-SRB. Then we introduce an algorithm and its correctness proof. Finally we discuss how this algo-

rithm is mapped to protocol implementation techniques.

2.1 Specification

We consider an asynchronous message passing system [4]. Briefly, we consider a set of sequential processes communicating through a fully connected network of point-to-point reliable channels. The system is asynchronous, which means that there are no bounds on processing or network delays. Processes can only fail by crashing and do not recover (a correct process does not crash). We assume that at most f processes may crash.

The definition of FIFO-SRB is based on obsolescence information formalized as a relation on messages. This relation is defined by the application program and encapsulates all the semantics ever required by the protocol. This way the FIFO-SRB protocol can be developed independently of concrete applications. The fact that m is *obsoleted* by m' is expressed as $m \sqsubset m'$. The obsolescence relation is a strict partial order (*i.e.* anti-symmetric and transitive) coherent with the causal ordering of events. The intuitive meaning of this relation is that if $m \sqsubset m'$ and m' is delivered, the correctness of the application is not affected by omitting the delivery of m . $m \sqsubseteq m'$ is used as a shorthand for $m \sqsubset m' \vee m = m'$.

FIFO Semantically Reliable Broadcast (FIFO-SRB) is defined by the following properties:

Validity: If a correct process broadcasts a message m and there is a time after which no process broadcasts m'' such that $m \sqsubset m''$, then eventually it delivers some m' such that $m \sqsubseteq m'$.

Agreement: If a correct process delivers a message m and there is a time after which no process broadcasts m'' such that $m \sqsubset m''$, then all correct processes eventually deliver some m' such that $m \sqsubseteq m'$.

Integrity: For every message m , every process delivers m at most once and only if m was previously broadcast by some process.

FIFO Order: If a process broadcasts a message m before it broadcasts a message m' , no process delivers m after delivering m' .

FIFO Completeness: If a process broadcasts a message m before it broadcasts a message m' and there is a time after which no process broadcasts m''' such that $m \sqsubset m'''$, no correct process delivers m' without eventually delivering some m'' such that $m \sqsubseteq m''$.

The intuitive notion that a message can be substituted by another that makes it obsolete is captured in the previous definitions by the statement “deliver some m' such that $m \sqsubseteq m'$ ”. When compared with the specification of reliable broadcast [6], our definition has two interesting differences: *i)* if there is an infinite sequence of messages that obsolete each other, the implementation may omit all of

these messages; and *ii*) it requires the implementation to ensure FIFO order and completeness. We will address each of these differences in turn.

The possibility of omitting all messages that belong to an infinite sequence is captured by the statement “there is a time after which no process broadcasts m'' such that $m \sqsubset m''$ ”. It may seem awkward at first that such occurrence is allowed. However, it should be noted that the application, by an judicious use of the labels that capture the obsolescence relation, can easily prevent infinite sequences from occurring. Actually, the application can decide exactly which is the most appropriate length of any sequence of messages from the same obsolescence relation. On the other hand, if the protocol was forced to deliver messages from an infinite sequence (by omitting the statement above from the specifications), the protocol designer would be forced to make an arbitrary decision of which messages to choose from that infinite sequence (e.g. one out of every k messages). It is clearly preferable to leave this decision under control of the application.

The FIFO completeness property ensures that full consistency at the semantic level is always guaranteed to be eventually reached, even if FIFO-SRB allows for temporary inconsistency of replicas. This happens because the specification enforces that, if a message is delivered, then all previous messages have already been delivered or have been made obsolete by subsequent messages whose delivery has been guaranteed.

2.2 Algorithm Overview

At first glance, it may seem that an implementation of FIFO-SRB can be obtained directly from an implementation of a Reliable Broadcast protocol. A naive implementation would just delete from the buffers messages made obsolete by the reception of a subsequent message. However, purging alone does not implement FIFO-SRB.

Consider for instance the following scenario: *i*) a process p broadcasts two unrelated message m_1 and m_2 ($m_1 \not\sqsubset m_2$); *ii*) the same process p broadcasts an infinite sequence of messages m_3, m_4, \dots such that $m_1 \sqsubset m_3$ and $\forall_{i \geq 3} : m_i \sqsubset m_{i+1}$. Consider that p purges m_1 from its buffer before sending both m_1 and m_2 to another process q . Since m_2 was sent after m_1 , q will not deliver m_2 before m_1 arrives (which would never occur) or until it realizes that m_1 was purged because it was made obsolete by some other message m_i . However, since m_i belongs to an infinite sequence, it may never arrive at q . Therefore, the protocol must incorporate some mechanism to ensure that q is informed about the purging of m_1 .

There is another more subtle issue regarding the implementation of FIFO-SRB. Even if information about purged messages is propagated, it is possible to show that the naive implementation would not ensure both Validity and FIFO Completeness in the case of failures. Consider the same scenario as above and the following sequence of events: *i*) process p broadcasts m_1, m_2 and m_3 ; *ii*) p purges m_1 due to m_3 and informs the remaining processes that m_1 was purged; *iii*) p sends m_2 which is delivered by some process q ; *iv*) p crashes. Clearly, this sequence violates FIFO

Completeness. The problem is that m_1 was purged before ensuring the delivery of m_3 . A message is guaranteed to be eventually delivered as soon as it has been received by $f+1$ processes, where f is the maximum number of processes that may fail. When this condition holds, we say that the message is *safe*. In the particular sequence above, violation of FIFO Completeness could be avoided if purging of m_1 was delayed until m_3 was known to be safe.

Given these observations, our protocol is based on the following principles:

- As in any reliable protocol, processes forward all the messages they received to mask the failure of the sender.
- In a retransmission buffer, a message m may be purged only if there is another message m' such that: $m \sqsubset m'$ and m' is safe.
- When a message is purged, enough information is stored to inform the remaining processes that the message has been purged.

The protocol is presented in detail in the next section.

2.3 Algorithm Specification

We describe the algorithm using an abstract specification. The use of this level of abstraction simplifies the proof of correctness and highlights the fundamental aspects of the solution. Later in Section 2.5 we show how a practical implementation can be obtained from this specification.

The system execution is modeled as a sequence of states. Each state is a mapping from state variables to values. A next state relation is a predicate on pairs of states. A specification is a set of executions, which can be defined by a next state relation which is true for consecutive states in legal executions, plus fairness assumptions written in temporal logic.

The state describes both the algorithm and the environment. Processes and channels are not explicit: A process state is a portion of system state and channel operations are modeled as copying elements between the state of two processes [7]. Process crash is denoted explicitly by state variables. A process is considered correct if its crashed state is forever false.

We use the common notation for sets. For tuples, we use the usual notation π_n to denote projection of element n . This notation is extended for sets of tuples with Π_n to denote the set of projections. For sequences, we use $\langle m \rangle$ to denote a sequence with one element m and \circ to denote concatenation. $elems(S)$ denotes the set of elements of a sequence S . A message broadcast by process i is expressed as $m \in elems(B_i)$ and a message delivered by process i is expressed as $m \in elems(D_i)$. Likewise, order of broadcast and delivery are expressed as ordering in sequences B_i and D_i .

The variables used by each process i are listed in Figure 1. Variable B_i simply keeps the messages that have been broadcast by the process. The variable c_i records

State for each process i :

- B_i : messages broadcast, initially empty sequence
- D_i : messages delivered, initially empty sequence
- c_i : boolean, initially false
- $O_{i,j}$: outgoing toward j , initially empty
- $I_{i,j}$: incoming via j , initially empty
- Q_i : queued for delivery, initially empty

Figure 1: State variables.

<p>TE1: $transmit_{j,i}(m, o)$</p> <p>PRE-CONDITION:</p> $(m, o) \in O_{j,i} \wedge \neg c_j \wedge$ $m \notin \Pi_1(I_{i,j}) \wedge \neg c_i$ <p>EFFECT:</p> $I_{i,j} := I_{i,j} \cup \{(m, o)\}$	<p>TE2: $crash_i$</p> <p>PRE-CONDITION:</p> $ \{j : c_j\} \cup \{i\} < f$ <p>EFFECT:</p> $c_i := true$
--	---

Figure 2: Transitions associated with the environment.

the state of the process (false if the process is correct and true if the process is crashed). Each process keeps a pair of buffers $I_{i,j}$ and $O_{i,j}$ for each process j . $I_{i,j}$ are incoming buffers, where messages waiting to be ordered are stored. $O_{i,j}$ stores messages waiting to be transmitted. Messages ordered are copied to a local delivery queue Q_i and messages that have been delivered are recorded in D_i . Messages in $O_{i,j}$ and Q_i can be purged. Purging is modeled by changing an attribute that is associated with each message in a given queue. This attribute can have one of two values: D (the message contains data) or P (the message has been purged).

Figure 2 depicts the transitions of the environment. Transition TE1 simply specifies that messages in output buffers are eventually inserted in the corresponding input buffers from the destination processes (this models the transmission of messages in the links). Transition TE2 specifies that a process may crash as long as the maximum number of faulty processes has not been reached.

Figure 3 depicts the transitions for each process i .

- Transition TP1 corresponds to the broadcast of a message m . In this transition the fact that m has been broadcast is stored in B_i and the message is sent to self by inserting it in $O_{i,i}$ (note that the environment will eventually move the message to $I_{i,i}$). Notice that the predicate $next(m, S)$, which ensures that all predecessors of message m are available in set S , is used to enforce that the message being broadcast has the right sequence number.
- Transition TP2 captures the forwarding procedure executed by every node. When a message is received for the first time and it is the next message in

<p>TP1: $broadcast_i(m)$ PRE-CONDITION: $next(m, O_{i,i}) \wedge \neg c_i$ EFFECT: $B_i := \langle m \rangle \circ B_i$ $O_{i,i} := O_{i,i} \cup \{(m, D)\}$</p>	<p>TP4: $purge_q_i(m)$ PRE-CONDITION: $\exists m' : (m, D), (m', D) \in Q_i \wedge$ $m \sqsubset m' \wedge \neg c_i$ EFFECT: $Q_i := (Q_i \setminus \{(m, D)\}) \cup \{(m, P)\}$</p>
<p>TP2: $enqueue_i(m, o)$ PRE-CONDITION: $\exists j : (m, o) \in I_{i,j} \wedge m \notin \Pi_1(Q_i) \wedge$ $next(m, Q_i) \wedge \neg c_i$ EFFECT: for all $k \neq i: O_{i,k} := O_{i,k} \cup \{(m, o)\}$ $Q_i := Q_i \cup \{(m, o)\}$</p>	<p>TP5: $purge_r_i(m)$ PRE-CONDITION: $\exists m' : (m, D), (m', D) \in O_{i,j} \wedge$ $m \sqsubset m' \wedge safe_i(m') \wedge \neg c_i$ EFFECT: $O_{i,j} := (O_{i,j} \setminus \{(m, D)\}) \cup \{(m, P)\}$ $safe(m, i) =$ $\{j : m \in I_{i,j}\} > f$</p>
<p>TP3: $deliver_i(m)$ PRE-CONDITION: $(m, D) \in Q_i \wedge m \notin elems(D_i) \wedge \neg c_i$ EFFECT: $D_i := \langle m \rangle \circ D_i$</p>	<p>$next(m, S) =$ $\forall s < seq(m), \exists m \in \Pi_1(S) :$ $snd(m) = snd(m') \wedge seq(m') = s$</p>

Figure 3: Transitions associated with process i .

the sequence, as enforced by $next(m, S)$, it is copied to all output buffers and inserted in Q_i for delivery.

- Transition TP3 captures the delivery of messages (note that, in practice, when a purged message is delivered the application is not disturbed).
- Transition TP4 specifies that a message m , waiting to be delivered, can be purged as long as in the same queue there is a subsequent message m' that makes m obsolete.
- Finally, transition TP5 specifies that a message m in an output buffer can only be purged if in the same queue there is a subsequent message m' that makes m obsolete and m' is safe. This is ensured by predicate $safe(m, i)$, which checks that process i has received m from more than f processes.

The fairness assumptions for the algorithm are the following. No fairness assumptions for $broadcast_i(m)$, $purge_q(m)$, $purge_r(m)$ and $crash$, thus allowing them to be forever enabled but never executed. Weak fairness is assumed for $transmit_{j,i}(m, o)$, for all i, j, m, o , and for $enqueue_i(m, o)$ and $deliver_i(m)$, for all i, m, o . This requires them to be eventually executed if forever enabled. Notice that there is no fairness imposed on purging operations, thus allowing reliable executions where no message is discarded.

2.4 Proof Sketch

We focus on proving liveness properties of the specification because these are the ones that make the difference to strict reliability and are those that can be compromised by losing messages.

Of the remaining properties, Integrity is trivially satisfied. The correctness of FIFO Order derives from *i*) Q_i containing a complete prefix and *ii*) a message which is purged in Q_i is never available as data after that in Q_i .

The proof of each of the liveness properties of the specification requires that if some condition on a message m is true, then some message m' such that $m \sqsubseteq m'$ is eventually delivered. This is split in two steps:

1. We prove that if for a pair of correct processes $j, i, m \in \Pi_1(O_{j,i})$ and there is a time after which no process broadcasts m'' such that $m \sqsubset m''$, then eventually exists some $m' \in \text{elems}(D_i)$ such that $m \sqsubseteq m'$.
2. For each specification property, we prove that the condition it imposes on m implies that for some pair of correct processes $j, i, m \in \Pi_1(O_{j,i})$.

This makes the proof associated with the first step in Lemma 3 the only eventuality proof required. This proof uses the results of two auxiliary lemmata which summarize interesting aspects of the protocol. The proofs use some additional notation: The *path* to a process i , denoted H_i , is defined as $H_i = \bigcup_{-c_j} (O_{j,i} \cup I_{i,j})$. The *world* W is $\bigcup_{i=0, j=0}^{n,n} (I_{i,j} \cup O_{i,j})$. The predecessors of a message m are $\text{Pred}(m) = \{m' \in M : \text{snd}(m') = \text{snd}(m) \wedge \text{seq}(m') < \text{seq}(m)\}$.

Lemma 1 *If $(m, P) \in H_i$ then there is some m' such that $m \sqsubset m'$ and for every process j (correct or not) $m' \in \Pi_1(H_j)$.*

PROOF: If $(m, P) \in H_i$ then for some process $k, (m, P) \in O_{k,i}$ or $(m, P) \in I_{i,k}$. Moreover, if $(m, P) \in I_{i,k}$ then $(m, P) \in O_{k,i}$. This is true as *i*) (m, P) is never removed from $O_{k,i}$ and *ii*) the only action that inserts elements in $I_{i,k}$ is only enabled if the same element is in $O_{k,i}$.

Trivially if $(m, P) \in O_{k,i}$ then $(m, P) \in W$. If $(m, P) \in W$ then there is some $m', m \sqsubset m'$, and a set of processes L with $|L| > f$, such that for any $l \in L, m' \in I_{k,l}$. This is true as *i*) the only action that inserts (m, P) in W is only enabled when m' is in more than f incoming queues and *ii*) if $m' \in \Pi_1(I_{k,l})$ once, then it is forever true. Therefore, for any process $l \in L, m' \in O_{l,k}$ and there is at least one $l \in L$ that is correct, as crash is enabled only for f processes.

If $m' \in \Pi_1(O_{l,k})$ then for all $j, m \in O_{l,j}$. This is true as *i*) transition $\text{enqueue}_j(m, o)$ always inserts m in all $\Pi_1(O_{l,j})$. Therefore, for any $j, m' \in \Pi_1(H_j)$. \square

Lemma 2 *Any path H_i contains a complete sequential prefix of the message ordering: For all $m \in \Pi_1(H_i), \text{Pred}(m) \subset \Pi_1(H_i)$.*

PROOF: For all i , $\Pi_1(Q_i)$ is a prefix of the ordering. This is true as the only action that changes it is only enabled when the new message is the next in the sequence. Moreover, $\Pi_1(O_{j,i} \cup I_{i,j})$ is always equal to $\Pi_1(Q_j)$. The only actions that change $\Pi_1(O_{j,i} \cup I_{i,j})$ also change $\Pi_1(Q_j)$ accordingly. Therefore, as the union of prefixes is still a valid sequential prefix, any path H_i contains a prefix. \square

Lemma 3 *If forever $m \in \Pi_1(H_i)$ then eventually $m' \in \text{elems}(D_i)$ such that $m \sqsubseteq m'$.*

PROOF: We define a set of tuples $\text{Stat}(m) \subseteq P \times M \times 2^M \times 2^M \times 2^M \times 2^M$ such that $(r, x, s_0, s_1, s_2) \in \text{Stat}(m)$ iff $m \sqsubseteq x$; $\bigcup_{i=0}^3 s_i = \text{Pred}(x) \cup \{x\}$ and $\forall i \neq j : s_i \cap s_j = \emptyset$.

We define a relation \prec in $\text{Stat}(m)$ such that $t \prec t'$ iff either $\pi_1(t) \subset \pi_1(t')$; or $\pi_1(t) = \pi_1(t')$ and $\pi_1(t) \sqsubset \pi_1(t')$; or $\pi_1(t) = \pi_1(t')$ and $\pi_1(t) \not\sqsubset \pi_1(t')$ and for some a for all $3 \leq a < b \leq 5$, $\pi_a(t) = \pi_b(t')$ and $\pi_a(t) \subset \pi_b(t')$. If the set of messages that make m obsolete is finite, then $\text{Stat}(m)$ is also finite. $(\text{Stat}(m), \prec)$ is a strict partial order because both strict set inclusion and obsolescence are strict partial orders. Thus $(\text{Stat}(m), \prec)$ is well-founded.

We now define a function $f_{i,m}$ from system state to $\text{Stat}(m)$ defined for states in which process i has not crashed and $m \in \Pi_1(H_i)$. Let $f_{i,m} = (r, x, s_0, s_1, s_2)$ such that:

- $r = \{i : \neg c_i\}$
- choose $x \in \Pi_1(H_i)$ such that $m \sqsubseteq x$ and $\forall m' \in \Pi_1(H_i) : x \not\sqsubseteq m'$;
- $s_0 = \Pi_1(\bigcup_{k \in c} O_{k,i}) \setminus \Pi_1(\bigcup_{k \in c} I_{i,k}) \cap \text{Pred}(x)$
- $s_1 = \Pi_1(\bigcup_{k \in c} I_{i,k}) \setminus \Pi_1(Q_i) \cap \text{Pred}(x)$
- $s_2 = \Pi_1(Q_i) \setminus D_i \cap \text{Pred}(x)$

Assuming that $m \in \Pi_1(O_{j,i})$ such that j is correct (forever $\neg c_j$), we prove that eventually $m' \in \text{elems}(D_i)$ by ensuring that *i*) if $m \in O_{j,i}$ then $f_{i,m} \in \text{Stat}(m)$, which is true by definition; *ii*) for some helpful transitions either $f_{i,m} \prec f'_{i,m}$ or $m' \in \text{elems}(D_i)$ and at least one is enabled or $m' \in D_i$; and *iii*) for the remaining transitions, never $f'_{i,m} \prec f_{i,m}$.

The transitions considered helpful and respective resulting values for $f_{i,m} = (r, x, s_0, s_1, s_2)$ are:

- $\text{transmit}_{j,i}(m', o)$ if $m' \in s_0$, leads to $(r, x, s_0 \setminus \{m'\}, s_1 \cup \{m'\}, s_2)$.
- $\text{enqueue}_i(m', o)$ if $m' \in s_1$, leads to $(r, x, s_0, s_1 \setminus \{m'\}, s_2 \cup \{m'\})$.
- $\text{deliver}_i(m')$ if $m \not\sqsubseteq m' \wedge m' \in s_2$, leads to $(r, x, s_0, s_1, s_2 \setminus \{m'\})$. Notice that $m' \not\sqsubseteq m$ implies $m' \neq x$.
- $\text{deliver}_i(m')$ if $m \sqsubseteq m'$. Goal reached with $m' \in \text{elems}(D_i)$.

At least one of these is enabled: If $transmit_{j,i}(m, o)$ is not enabled for all j , then at least $m \in \Pi_1(I_{i,j})$ for some j as $m \in \Pi_1(H_i)$. If $enqueue_i(m, o)$ is also not enabled, then $m \in \Pi_1(Q_i)$. Otherwise, with $m \in \Pi_1(H_i)$ and H_i containing complete prefixes (by Lemma 2), transmission would have to be enabled. If $delivery_i(m), m \not\sqsubseteq m'$ is also not enabled then $s_2 = \{x\}$, as Q_i contains x . Otherwise $delivery_i(x)$ is enabled as x must be tagged with D. Otherwise (by Lemma 2) it would not be a maximal element.

There are transitions which help but are not guaranteed to occur. Either because there is no fairness, namely in $broadcast_i(m')$ if $x \sqsubset m'$ and $crash_j$ if $\neg c_j$, or are fair but may never be enabled, namely $enqueue_k(m')$ if $x \sqsubset m'$. Other actions leave $f_{i,m}$ unchanged. Notice that $crash_i$ does not happen by assumption that i is correct. \square

Theorem 1 (Validity) *If a correct process broadcasts a message m and there is a time after which no process broadcasts m'' such that $m \sqsubset m''$, then eventually it delivers some m' such that $m \sqsubseteq m'$.*

PROOF: It is trivially true that if $m \in B_i$ then $m \in O_{i,i}$ and process i is correct by assumption. Proof follows immediately by Lemma 3. \square

Theorem 2 (Agreement) *If a correct process delivers a message m and there is a time after which no process broadcasts m'' such that $m \sqsubset m''$, then all correct processes eventually deliver some m' such that $m \sqsubseteq m'$.*

PROOF: By a simple invariance proof, if i delivers m then $m \in O_{i,j}$ for all j and process i is correct by assumption. Proof follows immediately by Lemma 3. \square

Theorem 3 (FIFO Completeness) *If a process broadcasts a message m before it broadcasts a message m' and there is a time after which no process broadcasts m''' such that $m \sqsubset m'''$, no correct process delivers m' without eventually delivering some m'' such that $m \sqsubseteq m''$.*

PROOF: By a simple invariance proof (same as Agreement), if i delivers m' then $m' \in O_{i,j}$ for all j . By Lemma 2, the same $O_{i,j}$ contains m . Thus either m is delivered or by Lemma 1 some m'' such that $m \sqsubseteq m''$ exists. \square

2.5 Deriving an Implementation

The abstract specification of the algorithm makes several simplifications, such as assuming that information about past messages indefinitely accumulates in the variables at each process. The specification also requires that information about purged messages is always explicitly sent on the network. We now argue how a practical implementation can be derived from the specification. For clarity, we address first the case where no purging occurs before discussing how purging can be implemented.

In the algorithm, sets $O_{j,i}$, $I_{i,j}$ and Q_i represent a point-to-point FIFO reliable channel as follows: Messages currently in transit (sent but not yet received), are $O_{j,i} \setminus Q_i$. Messages available only at the sender side are $O_{j,i} \setminus I_{i,j}$. Messages available at the receiver side waiting to be ordered are $I_{i,j} \setminus Q_i$. Notice that operations *transmit* and *enqueue* never refer to the content of messages in Q_i but need only the knowledge of which is the sequence number of the last message delivered.

In practice, this can be implemented using a pair of buffers (one on each side of the channel) and a sequence counter on the receiver: *i*) messages sent are placed in the outgoing buffer, being eventually sent and if necessary repeatedly resent to the network (this is first line of *enqueue*); *ii*) upon reception, an acknowledgment is sent back and if necessary, repeatedly resent; *iii*) upon reception of acknowledgment the message is removed from the sender buffer (this implements *transmit*); *iv*) when a message bearing the next sequence number is available at the receiver, it is removed from the buffer and the sequence number is incremented (this implements the second line of *enqueue*).

Therefore, it is possible to implement the abstract specification of a channel using a window-based protocol. Since in the proposed algorithm there is symmetric connection (*i.e.* $O_{i,j}$, $I_{j,i}$ and Q_j), acknowledgments can be piggybacked on messages traveling in the opposite direction as happens in TCP/IP in which acknowledgments are implicit in the lower bound of the window.

When purging happens in $O_{j,i}$, (m, D) is replaced by (m, P) . In practice, for purging to be useful this must be implemented as freeing all resources (memory and bandwidth) consumed by m . That this can be done in the sender's buffer, thus preventing network resources from being wasted. However, the receiver has to be notified that m has been purged in order to advance the sequence counter without receiving m .

This can be done using the following strategy: *i*) assume a fixed window of size w : the sender never puts sequence $s + w$ in the network without previously receiving an acknowledgment to s ; *ii*) the sender knows that it has not received an acknowledgment s if some m such that $seq(m) = s$ is in the buffer; *iii*) if m is purged, it is removed from the buffer thus allowing $s + w$ to be put in the network and eventually received. When the receiver gets $s + w$ without ever getting s it must conclude that the message with sequence s has been purged. This implements (m, P) being inserted in $I_{i,j}$. Notice that no message labeled with P is in the algorithm ever used for anything besides inspecting its sequence number, which in practice translates to it not occupying space. Note also that if there are no further messages to send, a message indicating that the window is empty needs to be explicitly sent.

Likewise, Q_i and D_i abstract a FIFO queue holding messages $Q_i \setminus D_i$ ordered by sequence number. Messages are inserted by *enqueue* and removed by *deliver*. Purging in this buffer is implemented by removing the purged message.

If broadcast links are available, it is also possible to optimize the message forwarding procedure that, in the abstract specification, requires each message to be transmitted on the network n^2 times. Two optimizations are possible:

- As a message is always simultaneously inserted in all outgoing channels $O_{i,j}$, a network level broadcast mechanism can be used to transmit it, thus reducing the complexity to n .
- As soon as a message is received in some $I_{i,j}$ it can be acknowledged in all incoming channels: the receiver advances the lower bound of the window thus allowing the sender to immediately remove the message from its buffers. This also reduces the complexity to n .

Using both optimizations simultaneously, a message can be transmitted only once in the network. In addition, as in conventional reliable broadcast protocols, the explicit point-to-point acknowledgment mechanism can be replaced by a global stability tracking mechanism thus further improving performance and scalability.

3 Applying FIFO-SRB

This section illustrates the use of FIFO-SRB. For that we consider a set of replicated servers that store the current value of a set of data items. These values are updated by a stream of requests from one or more concurrent clients.

3.1 Defining the Obsolescence Relation

To make the presentation clearer, we concentrate on a single stream of updates from a client to the set of servers. Note that this simplification does not make our simulation less relevant: The studied scenario is characteristic of one of the main techniques to implement dependable servers; the so called primary-backup replication. In a primary-backup system, requests from clients are executed at the primary that subsequently broadcasts an update to the backup replicas. Should the primary fail, it is required that replicas are consistent among them and with the primary so that any of them can be promoted to primary. When using FIFO-SRB, this translates to ensuring that all backups have the most recent version of all items.

It is assumed that each request modifies at most one data item of the state. The obsolescence relation is determined by the identification of the item carried by each message, thus $m \sqsubset m'$ if both m and m' refer to the same item and m is broadcast prior to m' . As we want the state of clients strongly consistent with the server's, we expect that:

- If the server stops to modify its state then eventually the same state is reached by all correct clients.
- If the server crashes then eventually the state of all correct clients is the same and equal to the state of the server at some point in time.

In the first scenario, we know that the last update to each item never becomes obsolete. Therefore, by Semantic Validity the last update is always delivered back

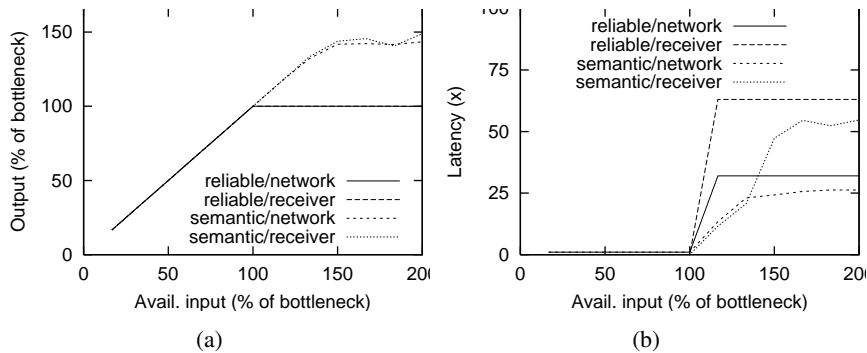


Figure 4: Performance of FIFO-SRB and a slow network link or a slow receiver.

to sender. By Semantic Agreement and Integrity, each other correct process delivers the same set of last updates. By FIFO Order, each of the last updates is delivered after other updates to the same item thus resulting in the same state as in the sender.

In the second scenario, consider the last update message to each item delivered by some process. By Integrity, Semantic Agreement and FIFO Order, all other correct processes deliver the same messages as last updates as they never become obsolete. Therefore the state among receivers is consistent. Consider the last messages delivered by receivers. By FIFO Completeness, we know that for every message broadcast by the sender prior to the last message delivered, at least one message that makes it obsolete has been delivered. Thus the state is the same as the state of the sender at the moment in time when it broadcast the last message delivered.

3.2 Performance

The impact of purging in the performance of a protocol without inter-replica consistency concerns has been extensively explored in [9]. Therefore we are here concerned only with the impact of the protocol mechanisms used to ensure inter-replica consistency, such as the necessity to ensure safety prior to purging a message from retransmission buffers. This is done using an high-level discrete event simulation allowing us to isolate performance degradation due to slower receivers and network links from other aspects of group performance and to directly compare the results with our previous work.

The network is modeled as $n \times n$ queues fully connecting all processes. We model network latency by adjusting the delay of messages in the queue and network bandwidth by using a simple leaky-bucket scheme. The capacity of receivers is determined by the time required to consume each message. Each process implements the FIFO-SRB protocol by managing local bounded buffers. When its delivery queue fills up, it ceases to accept further messages from the network. Eventu-

ally, this will cause the outgoing buffers of the sender to be exhausted which, in turn, prevents further messages from the application from being accepted. This degrades throughput to all the remaining group members. Consumers are attached to all nodes. A single producer injects traffic in one of the nodes according to item access pattern described in the previous section. Item access frequency is generated using the distribution observed in a stock-trading application [8]:

Number of Stocks:	25	100	750	Total:	875
Frequency:	50%	40%	10%		100%

The fact that some items are accessed much more frequently than others (*e.g.* 25 of 875 items are accountable for 50% of total accesses) increases the probability of messages containing updates to the same item being near in the message stream. Such access patterns are common in high throughput applications.

The results from the simulations obtained for a configuration with 5 processes, and for all combinations of reliable and semantically reliable protocols with bottlenecks in the network and in the receiver are depicted in Figure 4. The relation between the sustained output and the desired input in the presence of a bottleneck (such as when the aggregate bandwidth of network links to a node is reduced or when the processing capacity of a receiver is limited) is illustrated in Figure 4(a). When the available input is less than the capacity of the bottleneck, all messages can be transmitted and consumed and thus the output equals the input as shown by the 45 degree slope in the graphic. It is also possible to observe that a reliable protocol is unable to sustain a output larger than the limit of the bottleneck, as shown by the horizontal line in the graphic for input greater than 100%. On the other hand, with FIFO-SRB, the remaining receivers are unaffected by the bottleneck as long as purging remains effective. As noted before, the purging rate depends on the buffer size and on traffic profile: in the simulated scenario purging remains effective with loads up to 150% of the bottleneck throughput. In addition, purging also improves latency by reducing average buffer usage as presented in Figure 4(b). Notice that when the bottleneck is the network, messages queue only on the sender’s buffer thus cutting latency in half regardless of semantic purging. In both situations, the observed purging rates are similar to those observed under the same circumstances without consistency concerns [9].

4 Conclusions and Future Work

Achieving stable high throughput in large and heterogeneous networks supporting reliable distributed systems is a challenging task. In this paper we address this challenge using the notion of message obsolescence, a technique that uses semantic knowledge about the contents of messages to discard old information from buffers.

The paper has proposed a new primitive, FIFO Semantic Reliable Broadcast (FIFO-SRB) that makes use of the obsolescence relation. FIFO-SRB is particularly useful for applications that have to disseminate updates to a collection of data

items: It ensures that all correct processes are guaranteed to receive the last update to each item, even if they do not receive exactly the same set of updates. An algorithm to implement FIFO-SRB has been specified and proved.

The performance of FIFO-SRB was evaluated through simulation. For our experiences we have used the reported obsolescence pattern of stock-trading applications. The results show that a FIFO-SRB protocol can prevent a slow process or link from becoming a bottleneck for the complete group.

In this paper we have considered systems using a fixed set of processes. The work on FIFO-SRB can be combined with a membership service to offer a generalization of View Synchronous communication that takes message obsolescence into account. It is also possible to define different coding techniques to capture more complex obsolescence relations.

References

- [1] R. Baldoni, A. Mostefaoui, and M. Raynal. Causal delivery of messages with real-time data in unreliable networks. *J. Real-time Systems*, 10(3), 1996.
- [2] K. Birman. A review of experiences with reliable multicast. *Software Practice and Experience*, 29(9):741–774, July 1999.
- [3] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Computer Systems*, 17(2):41–88, 1999.
- [4] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.
- [5] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM Symp. on Communications Architectures and Protocols*, pages 200–208, Philadelphia, PA, September 1990. ACM.
- [6] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell Univ., Computer Science Dept., May 1994.
- [7] L. Lamport. Processes are in the eye of the beholder. *Theoretical Computer Science*, 179(1–2):333–351, 1997.
- [8] P. Peinl, A. Reuter, and H. Sammer. High contention in a stock trading database: A case study. *ACM SIGMOD Record*, 17(3):260–268, September 1988.
- [9] J. Pereira, L. Rodrigues, and R. Oliveira. Semantically reliable multicast protocols. In *Proc. 19th IEEE Symp. Reliable Distributed Systems*, pages 60–69, October 2000.

- [10] J. Pereira, L. Rodrigues, R. Oliveira, and A.-M. Kermarrec. Probabilistic semantically reliable multicast. In *Proc. IEEE Int'l Symp. Network Computing and Applications (NCA)*, February 2002.
- [11] R. Piantoni and C. Stancescu. Implementing the Swiss Exchange Trading System. In *Proc. 27th Annual Int'l Symp. Fault-Tolerant Computing (FTCS'97)*, pages 309–313. IEEE, June 1997.
- [12] S. Raman and S. McCanne. Generalized data naming and scalable state announcements for reliable multicast. Technical Report CSD-97-951, Univ. of California, Berkeley, June 1997.
- [13] L. Rodrigues, R. Baldoni, E. Anceaume, and M. Raynal. Deadline-constrained causal order. In *3rd IEEE Int'l Symp. Object-oriented Real-time distributed Computing*, March 2000.