Reducing the Cost of Group Communication with Semantic View Synchrony *

José Pereira	Luís Rodrigues	Rui Oliveira
Univ. do Minho	Univ. de Lisboa	Univ. do Minho
jop@di.uminho.pt	ler@di.fc.ul.pt	rco@di.uminho.pt

Abstract

View Synchrony (VS) is a powerful abstraction in the design and implementation of dependable distributed systems. By ensuring that processes deliver the same set of messages in each view, it allows them to maintain consistency across membership changes. However, experience indicates that it is hard to combine strong reliability guarantees as offered by VS with stable high performance.

In this paper we propose a novel abstraction, Semantic View Synchrony (SVS), that exploits the application's semantics to cope with high throughput applications. This is achieved by allowing some messages to be dropped while still preserving consistency when new views are installed. Thus, SVS inherits the elegance of view synchronous communication. The paper describes how SVS can be implemented and illustrates its usefulness in the context of distributed multi-player games.

1 Introduction

There is an increasing range of distributed applications that must balance reliability requirements with a good performance under a high load of requests. An example of such applications are distributed multi-player games. These are typically implemented today around a centralized, non-replicated, server. Since the failure of the server may affect a large number of clients, it is interesting to replicate it without loss of efficiency. Note that it is now common that commercial companies exploit long lived games whose state needs to be preserved in face of failures, ideally without service interruption. Another example are distributed control and monitoring applications which exhibit also a highly interactive behavior [4].

In such applications, the state of the server can be modeled as a relatively small collection of data items. The values of these items are frequently updated while handling requests from clients. A form of primary-backup replication can be used to replicate the server: In each system configuration, one of the servers is chosen to execute the requests from clients and to disseminate updated state to other replicas using multicast operations. Group communication [3] offers a convenient paradigm to program such applications [13]. Under this model, dynamic group membership keeps track of new members that join the group and of current members that leave, facilitating the choice of the primary. With View Synchrony (VS), all members receive exactly the same set of messages between view changes thus ensuring the consistency of replicas upon the crash of the primary.

Unfortunately, experience indicates that it is hard to sustain high message loads with strong reliability guarantees [23]. This is inherent to reliability itself and not an artifact of a specific implementation: Reliable protocols are required to store messages until delivered and then until they have been acknowledged by all group members. Under high loads, a single slow member may prevent messages from being delivered or acknowledged at the same pace they are being produced. This quickly leads to buffer space shortages and to global performance degradation due to flow control. View synchrony allows expelling perturbed members from the group, but unfortunately, *transient* performance perturbations may result in excessive reconfigurations which themselves threaten high throughput.

To tackle this problem, we propose in this paper a novel abstraction called Semantic View Synchrony (SVS). SVS uses semantic knowledge about the data exchanged by the application such that messages with obsolete content may be discarded in the presence of overload conditions. By allowing some messages to be discarded, the system tolerates better the occurrence of performance perturbations without demanding the allocation of additional resources. Nevertheless, these processes are guaranteed to receive sufficient messages to remain consistent, thus preserving the elegance of view synchrony. SVS makes it possible to avoid group reconfigurations at the cost of delivering less detailed information to processes suffering performance perturbations. If purging of obsolete messages is not enough to overcome the perturbation, reconfiguration can still happen as the dynamic nature of membership is preserved.

This paper builds on our previous work on semantic reliability. In [21] we have introduced the concept of semantic reliability based on message obsolescence and shown which factors affect its effectiveness without considering inter-replica consistency constraints. In [22] we have introduced the specification and implementation of a semantically reliable broadcast protocol but we consider only the restricted setting of a static membership. This paper makes the following contributions: It introduces the Semantic View Synchrony abstraction, provides a specification of this service and an algorithm to implement it; it shows how obsolescence can be characterized in distributed multi-user applications and represented efficiently; finally it shows the impact of SVS in a concrete application scenario.

The rest of the paper is organized as follows. Section 2 motivates our work by examining the obstacles to stable high throughput and the intuition underlying semantic reliability. Section 3 defines Semantic View Synchrony and presents an algorithm to implement it. Section 4 shows how to apply SVS by describing usage scenarios and how the obsolescence relation is represented. The performance of SVS in the context of distributed multi-user games is analyzed in Section 5. Finally, Section 6 compares our approach with related work and Section 7 concludes the paper.

2 Motivation

2.1 Performance Perturbations and Throughput

The low cost and high performance of off-the-shelf computer systems makes them attractive for high throughput services. However, it has been documented that the performance of group communication in such systems is frequently disappointing [23]. This is not an artifact of any specific implementation: To ensure reliability, messages have to be buffered until reception is acknowledged by all participants. If a performance perturbation delays transmission or acknowledgment, garbage collection is delayed. Eventually buffer space is exhausted and the sender blocks, thus affecting the whole group.

Performance perturbations occur in disk subsystems, scheduling, virtual memory, interference by background applications and system tasks, and network operation. These happen even in local area networks and in closely controlled environments and seem to be unavoidable given the complexity of current computer systems [1]. Their impact can be observed in several stages of reliable protocols:

- A message is stored in buffers until it is ready for delivery. This might require the message to be delayed for ordering or for acknowledgments by a majority of receivers to be collected in order to ensure uniformity [11]. In either case, spurious delays or dropped messages in the network result in messages being stored for longer periods of time.

- Messages can be delivered only as fast as the application can consume them. Depending on the resources used by the application, message delivery may be affected by the performance of CPU, scheduling, memory and disk subsystems [1]. Performance perturbations that are likely to happen in any of them result in messages being stored for longer in protocol buffers.

- With a View Synchronous protocol, the message might have to remain buffered even after local delivery, as agreement on delivered messages upon view change might require retransmissions to be performed by processes other than the original sender. Therefore the space occupied by a message can only be freed after it is known to be stable, i.e. received by all processes. Again, stability tracking is sensitive to perturbations of the network and thus might also lead to increased buffer occupancy.

Although flow control makes it possible to reduce the offered load until the perturbed member recovers, different group members may suffer transient performance faults in different points in time resulting in a performance loss that is unacceptable for high throughput applications. Although the correctness of algorithms developed for the asynchronous system model is not affected by these perturbations, the performance is seriously affected. This paper is concerned with the performance of group communication in systems where these sort of perturbations may occur without impact in correctness. Our approach is complementary to mechanisms that detect timing failures affecting correctness in real-time systems [9, 27].

2.2 Design Alternatives

We can identify different alternative approaches to address the problems caused by the occurrence of transient performance perturbations and to avoid degradation of group throughput by flow control:

- Exclude a process from the group as soon as it suffers a performance perturbation [23]. This allows the throughput to be preserved at the cost of reducing the resilience of the system. Eventually 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. The excessive number of group reconfigurations represent also an impediment to sustain high throughput.

- To configure large enough buffer space such that it is able to mask sufficiently large performance perturbations. However, this alternative is not resource efficient. Additionally, the use of large buffers has also the negative effect of increasing the latency of view changes (e.g. when crashes occur) due to the large number of messages that have to be processed during view installation, being itself an obstacle to throughput stability.

- To sacrifice the reliability of group communication in perturbed group members [5]. However, doing so forces the application to deal directly with most of the complexity of distributed programming; the same complexity that is supposed to be dealt by view synchrony.

The abstraction proposed in this paper offers a new, precise and meaningful reliability criterion that allows to conciliate the following goals: *i)* preserve the ability to exclude from the group processes that have crashed; *ii)* enforce a consistency criterion that is strong enough to simplify the development of dependable applications; *iii)* accommodate transient performance perturbations without degrading the system throughput and without forcing excessive group reconfigurations; *iv)* be resource efficient, in the sense that processing and memory capacity can be configured for the stable case.

2.3 Message Obsolescence

Our proposal to address these goals is motivated by the observation that when the system is congested, buffers in the path to the bottleneck are full and thus are likely to contain messages that have been produced in different points in time. In many applications, recent messages implicitly convey the content or overwrite the effect of previous messages, which thereby become obsolete prior to their delivery to slow processes.

If obsolete messages can be recognized within protocol buffers and then purged, the application is relieved from processing some of the outdated messages and resources are freed to process further messages. Therefore, a recipient that suffers a performance perturbation does not prevent messages from stabilizing and can then be accommodated within the group without disturbing the remaining members. Purging of obsolete messages is not observed by fast members, which quickly deliver messages before becoming obsolete. This means that only slow processes omit deliveries: They do not receive all the messages but they still receive enough messages to be allowed to remain in the group.

On the other hand, to benefit from message obsolescence the traffic pattern must exhibit some obsolescence, i.e., our solution is not a panacea. However, it can be observed that a large class of high throughput applications, such as distributed multi-player games, allow high purging rates (this issue is addressed in Section 5). The application must then provide the obsolescence relation to the protocol in the multicast request as discussed in Section 4. Before addressing these two issues, we show how Semantic View Synchrony can be implemented.

3 Semantic View Synchrony

3.1 System Model

We consider an asynchronous message passing system model augmented with a failure detector [7]. In detail, the distributed system is modeled as a set of sequential processes which can: send a message; receive a message; perform a local computation; and crash. We do not make any assumption on process relative speeds but assume crash-stop failures of at most a minority of processes.

Processes are fully connected by a network of point-to-point message passing channels. Channels are used through primitives $\mathit{send}_i(m, j)$ and $\mathit{receive}_i(m, j)$ and we assume that are reliable and FIFO ordered. These assumptions are actually not strictly required, but used to simplify the presentation of the protocol. We do not assume any bound on the time that a message takes to be transmitted.

A consensus protocol is assumed to be available[1] and modeled as a procedure which takes as an input parameter a proposed value and returns a decided value. Consensus ensures that all correct processes eventually decide the same value and that the decided value is one of the proposed values.

3.2 Definition

The multicast service is used through a pair of primitives: *multicast*(m) and *deliver*(m). *Multicast*(m) initiates the transmission of a message. *Deliver*(m) is used by the application to obtain a message from the delivery queue, when available. We use a down-call style of interface to ensure that messages not being processed are kept in the protocol buffers (this simplifies the purging of obsolete messages). View changes are signaled to the application by delivering a special control message. Each view notification includes the identification of the view and of the set of processes which constitute the current membership of the group.

The set of events that may lead to a view change are not relevant to the definition of Semantic View Synchrony, as we are concerned only with safety. Examples of possible causes for triggering a view change to remove a process from the group are the occurrence of failure suspicions [18], the lack of available buffer space at one or more processes [8] and simply the existence of processes that voluntarily want to leave.

The definition of semantic reliability is based on obsolescence information encapsulated as a relation on messages. This relation is encoded by the application using techniques that will be described in Section 4. This makes the SVS protocol independent of concrete applications. We assume that messages are uniquely identified. Let m and m' be any two messages. The fact that m is *obsoleted* by m' is expressed as $m \sqsubset m'$. Also, $m \sqsubseteq m'$ is used as a shorthand for

[1]Notice that consensus can also be solved without the reliable channels assumption [12].

$m = m' \vee m \sqsubset m'$. The obsolescence relation is an irreflexive partial order (*i.e.*, anti-symmetric and transitive). 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.

The safety properties required to enforce strong consistency are:[2]

Semantic View Synchrony (SVS): If a process p installs two consecutive views v_i and v_{i+1} and delivers a message m in view v_i, then all other processes installing both v_i and v_{i+1} deliver some m', such that $m \sqsubseteq m'$, before installing view v_{i+1}.

FIFO Semantically Reliable: For all pairs of messages m, m' such that some process multicasts m before m': (i) no process delivers m after m'; (ii) if a process p installs two consecutive views v_i and v_{i+1}, and delivers message m' in view v_i, then p delivers some m'', such that $m \sqsubseteq m''$, before installing view v_{i+1}.

Integrity : If a message m is delivered to a process in v_i, then m has been previously sent by some process (no-creation). No message m is delivered to a process p more than once (no-duplication).

Notice that SVS property relaxes View Synchrony [28], as every pair of processes installing two consecutive views v_i and v_{i+1} will not necessarily deliver the same set of messages but they are ensured to deliver (at least) the same set of messages that have not been made obsolete by subsequent messages up to view v_{i+1}. For instance, process p_1 may deliver in view v_i messages m_1 and m_2, such that $m_1 \sqsubset m_2$, and process p_2 may deliver only m_2 in the same view. If no messages m, m' exist such that $m \sqsubset m'$, SVS reduces to conventional VS. This makes SVS more general as different concrete semantics, including VS, can be obtained by defining an appropriate obsolescence relation.

The FIFO Semantically Reliable property relaxes the traditional FIFO Reliable properties [28]. Given a sequence of messages multicast by a process, this ensures that upon view installation only obsolete predecessors of the last message delivered can be omitted.

3.3 View Change Protocol

In this section, we present a protocol that allows purging to be applied in the delivery queues as well as during view changes. The protocol offers performance improvements when accommodating a slower receiver. Additionally, the protocol also reduces the latency of the view change operation, as shown in Section 5. Techniques to address the impact of slower network links are described in [22].

[2] A previous report describes the implementation of a primitive satisfying the FIFO Semantically Reliable broadcast primitive [22] for systems with fixed membership. The interested reader can find there a comprehensive discussion on the topic of ensuring liveness of semantic reliability (e.g. a suitable definition of Validity).

declare
  View cv = (Integer id, SetOfProcessIds memb); // *current view*
  Boolean blocked;
  OrderdedSetOfMessages delivered;
  OrderdedSetOfMessages to-deliver;
  SetOfMessages global-pred[]; // one instance for each view
  SetOfProcessIds pred-received[]; // one instance for each view
  SetOfProcessIds leave[]; // one instance for each view

function purge (OrderdedSetOfMessages S) **do**
  while $\exists m = [\text{DATA},\text{v, d}], m' = [\text{DATA},\text{v', d'}] \in S$:
    $(v = v') \wedge (m \sqsubset m')$ **do** remove (S, m);

$t1$: **upon** *deliver* $\wedge$ (to-deliver $\neq \emptyset$) **do**
  m := removeFirst (to-deliver);
  addToTail (delivered, m);

$t2$: **upon** *multicast* $data$ $\wedge$ $\neg$ blocked $\wedge$ self $\in$ memb(cv) **do**
  addToTail (to-deliver, [DATA, *cv*, *data*]);
  forall $p \in$ memb(cv): $p \neq$ self **do send** [DATA, *cv, data*] **to** p;
  purge (to-deliver);

$t3$: **upon receive** $m = [\text{DATA}, v, d]$ **from** p: (v = cv) $\wedge\neg$ blocked **do**
  if $\nexists m' \in$ (to-deliver $\cup$ delivered): $m \sqsubseteq m'$ **do**
    addToTail (to-deliver, [DATA, v, d]);
    purge (to-deliver);

$t4$: **upon** *trigger-view-change* (l) **do**
  forall $p \in$ memb(cv) **do send** [INIT, *cv, l*] **to** p;

$t5$: **upon receive** [INIT, v, l] **from** p: (v = cv) $\wedge\neg$ blocked **do**
  if $p \neq$ self **do forall** $p \in$ memb(cv) **do send** [INIT, v, l] **to** p;
  blocked := true;
  leave(cv) := $l \cap$ cv;
  local-pred(cv) := $\{$[DATA,v, d] $\in$ (delivered $\cup$ to-deliver): v = cv$\}$;
  forall $p \in$ memb(cv) **do send** [PRED, cv, local-pred(cv)] **to** p;

$t6$: **upon receive** [PRED, v, P] **from** p: (v = cv) **do**
  global-pred(cv) := global-pred(cv) $\cup$ P;
  pred-received(cv) := pred-received(cv) $\cup$ p;

$t7$: **upon** $\forall_{p \in \mathrm{memb(cv)}:\neg \mathrm{suspects(p)}}$: $p \in$ pred-received(cv) $\wedge$
    $|$pred-received(cv)$| > \frac{|memb(cv)|}{2}$ **do**
  proposal := $(id(cv) + 1$, pred-received(cv)$\setminus$ leave(cv));
  (next-view, pred-view) := consensus(cv, (proposal, global-pred(cv)));
  if self $\in$memb(next-view) **do**
    forall $m \in$ pred-view: $m \notin$ (to-deliver $\cup$ delivered) **do**
      addToTail (to-deliver, m);
    addToTail (to-deliver, [VIEW, next-view]);
    purge (to-deliver);
    cv = next-view;
    blocked := false;

Figure 1: Semantic View Synchrony.

Interestingly, SVS can easily be obtained by adapting an existing view synchronous protocol to include purging of obsolete messages at the appropriate steps. It is hence possible to derive SVS implementations to different systems models, by adapting different view synchronous implementations. The purpose of this section is not to re-invent view synchronous protocols, since these have been extensively studied in the literature [15]. However, we do want to illustrate what changes are needed to accommodate SVS. In order to do so, we opted to adapt a protocol designed to run on asynchronous systems augmented with a failure detector, which allows only processes to leave the group and that uses consensus as a building block [14]. The algorithm is depicted in Figure 1. The parts that have been added to accommodate SVS are highlighted in the figure (the changes are in gray). For self-containment, we provide a brief description of the complete algorithm.

Each process in the group keeps a variable *cv* with the most recent view, a boolean variable *blocked* that is used to prevent the reception and transmission of new messages during the view change protocol, and two FIFO ordered queues of messages: *to-deliver* and *delivered*. When messages are received they are inserted in the *to-deliver* queue where they wait for the application to consume them using the *deliver* operation. A message m in the *to-deliver* queue may be purged if a message $m' : m \sqsubset m'$ is received in the same view. This is modeled by the *purge* function. Delivery of a message m is simply modeled by removing m from the head of *to-deliver* and adding it to the tail of the *delivered* queue ($t1$). Delivery of views is modeled by the delivery of a control message. Two types of messages can be inserted in the *to-deliver* and *delivered* queues: data messages and view messages. A data message, denoted [DATA, v, d], is always tagged with the view v in which it is sent. A view message is denoted [VIEW, v]. The protocol uses two additional control messages whose purpose is explained in the following paragraphs.

Data messages can only be multicast if the group is not blocked ($t2$). A multicast message is tagged with the current view and sent to all the other processes in the view. The message is also inserted in the *to-deliver* queue of the sender. This will ensure that if the sender participates in the next view, all the messages it has sent will be delivered in the current view. Data messages are only accepted if the recipient is still in the view they were sent and if the group is not blocked ($t3$). As before, received messages are added to the *to-deliver* queue of the recipient.

The installation of a new view is triggered by an external event. In response to this event, the initiator of the view change simply disseminates a INIT control message to all group members ($t4$). Upon the reception of the first INIT message, a process forwards the INIT to all other members, ensuring that all correct processes initiate the view change ($t5$). Additionally, each process computes the sequence of messages it has accepted to deliver in the current view and sends this sequence to all other processes in a PREC control message. These sets are accumulated by all correct processes in the *global-prec* set ($t6$). The set of processes from which the PREC message has been received for the current view is maintained in the variable *pred-received*. When *pred-received* includes all processes from the current view that are not suspected, and this set contains a majority of processes, a new view as well as the sequence of messages to be delivered in the current view are proposed for consensus ($t6$). The proposed view corresponds to the *pred-received* set (minus the *l* processes that is given as input parameter to the view change procedure).

The view installation procedure is concluded after consensus returns. The agreed sequence of messages to be delivered in the current view is added to the *to-delivered* queue, followed by the agreed next view. Finally, the current view is updated and the group is unblocked.

3.4 Correctness Argument

When addressing the correctness of the algorithm we focus on Semantic View Synchrony and the second clause of FIFO Semantically Reliable. The reason for this is that these are the properties that differ from those found on VS algorithms and thus reflect the impact of purging obsolete messages.

The original VS algorithm, obtained from Figure 1 without the shaded lines or with an empty obsolescence relation, implements conventional VS [14]. From this we can derive the correctness of the implementation of SVS considering the following fact: the purge operation never discards maximal elements by the obsolescence relation $\sqsubset$ of the set of messages delivered by some process prior to installing a given view. If a process participates in view v_{i+1} and purges some message m, then there is some m' in *to-deliver* $\cup$ *delivered* such that $m \sqsubset m'$ that would be included in the *pred-view* set decided for v_{i+1} and thus m would not be maximal.

The correctness SVS follows from that. For any message m delivered by some process installing both v_i and v_{i+1}, either (i) m is maximal in the set of messages and thus is never purged and as in the original algorithm delivered by all processes before installing v_{i+1} or (ii) m is not maximal and there is some m' such that $m \sqsubset m'$ which is maximal.

The argument for the second clause of FIFO Semantically Reliable is similar. As channels are reliable and FIFO, it can easily be shown that without ever purging messages, *to-deliver* $\cup$ *delivered* contain always complete prefixes of sequences of messages multicast by each sender. The subset of maximal elements (as by the obsolescence relation $\sqsubset$), which is guaranteed to be maintained by purging is sufficient to ensure the desired property.

4 Capturing Message Obsolescence

For SVS to be applied in practice, one needs to develop efficient techniques to represent the obsolescence relation such that the protocol can recognize and purge obsolete messages. In this paper, we concentrate on applications that use reliable multicast to disseminate values of data items to a group of replicas. Other application scenarios are discussed in [20].

In detail, we assume that all group members maintain a collection of data items. The values of these items are continuously updated by one process upon handling requests from external client processes and then disseminated to other members of the group. Each multicast contains the updated value of one or more items in the collection. The role of SVS is to ensure that all members in the view receive the most up-to-date value of each item. Additionally, if the group needs to be reconfigured, SVS guarantees that all group members have the same state when a new view is installed. This behavior captures a fundamental issue in primary-backup replication, where a primary server executes requests from clients and forwards state updates to backup replicas. The equivalence of state ensures that on fail-over, any surviving replica can be selected for the role of the primary.

4.1 Types of Multicast Operations

We distinguish two relevant types of multicast messages: *single-item* and *multi-item* messages.

Single-item Message Each message contains the value of a single modified item. The definition of the obsolescence relation is here fairly simple: Messages containing values for the same item are related and all but the last can be considered obsolete.

Multi-item Messages A single message updates more than one item. The reason for updating several items with a single message is that this is a simple way of ensuring the atomicity of a composite update. When the message is delivered all items are updated, if the message is lost, none of the items is changed.

It is very hard to establish useful obsolescence relations among messages containing composite updates: if m and m' are two composite updates, we only have $m \sqsubseteq m'$ if the set of items updated by m' is a super-set of the items updated by m. Therefore, one needs a technique that allows the protocol to apply the obsolescence to individual updates within a composite multicast, while at the same time preserving the atomicity of the composite update.

The solution for this is to split an update into a batch of independent messages, where each message updates an individual item. The batch is terminated by a *commit* control message. Messages from a given batch are only applied when the correspondent commit message is received. Since FIFO order is used, the commit message is guaranteed to be only delivered after all the messages from batch have been delivered. Note that the role of the commit message can be performed by the last message in each update, thus eliminating the need for an extra message. In fact, all the updates from the same batch are easily piggybacked by the protocol in a single transport-level message. Therefore, this technique does not increase the number of transport-level messages exchanged by the protocol.

Since individual updates from a given batch can only applied when the commit message

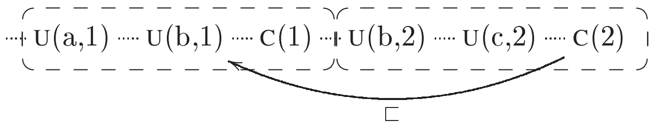


Figure 2: Preserving atomicity of updates in a multi-item operation.

arrives, obsolescence should also be only applied at that point. This restriction can be captured by ensuring that only the commit messages, and not the individual updates, can make messages from previous batches obsolete. For instance, in the example of Figure 2, it is $\mathrm{C}(2)$ the commit from the second batch, and not the second update to item b, $\mathrm{U}(\mathrm{b},2)$, that makes $\mathrm{U}(\mathrm{b},1)$ the first update to item b obsolete.

4.2 Representing Obsolescence

The obsolescence relation has be to encoded by the application before being conveyed to the protocol. We are interested in general purpose techniques that can be applied to a wide range of systems in an efficient manner. Therefore, we exclude solutions such as making the protocol aware of the contents of messages [6] or enriching the messages with code [26]. Instead, we prefer to let the application supply this information to the protocol as an extra parameter of the multicast operation. Upon multicast of a message m, the protocol is informed of all messages m' such that $m' \sqsubset m$. In the following paragraphs we propose and discuss three different representation techniques: item tagging, message enumeration and $k-$enumeration.

Item Tagging The simplest representation technique consists in associating a unique integer tag to each data item managed by the application. This is particularly effective for systems that use single-item updates. In this case, each message is tagged with the identifier of the data item it is updating. Tags are added to the message headers and used in combination with the sender identification and sequence numbers generated by the protocol: if two messages from the same sender carry the same tag, the one with the highest sequence number makes the other obsolete. Although simple, this technique cannot be easily extended to applications that use multi-item composite updates. In fact, using this technique it is difficult to express that a message makes obsolete several other unrelated messages.

Message Enumeration A more general alternative consists in having each message explicitly enumerate which preceding messages it makes obsolete. This approach is clearly more expressive than the item tagging approach. On the other hand, it is not compact and burdens the protocol with the task of determining the transitive closure of the relation. Consider three messages such that $m_1 \sqsubset m_2 \sqsubset m_3$. The representation of obsolescence should allow to verify that $m_1 \sqsubset m_3$ without requiring m_2 to be available.

To ensure that the transitivity of the obsolescence is preserved in the message enumeration technique, a message must enumerate not only its direct predecessors, but all the (transitive) predecessors. In practice, only the recent messages from the enumeration need to be carried by each message without any significant impact on the purging efficiency. This optimization is possible because it is very unlikely that two messages far apart in the message stream can be found simultaneously in the same buffer.

k-Enumeration The k-enumeration technique combines the efficiency and simplicity of the tagging approach with the expressiveness of the message enumeration approach. The technique exploits the fact that purging is mainly applied to pairs of messages that are close to each other in the message stream.

The technique works as follows. Each message explicitly enumerates which of the k preceding messages it makes obsolete. This information can be stored in a bitmap of k size. If the nth position of the bitmap is set to true, the message makes obsolete the nth preceding message. Each messages carries the k-enumeration bitmap as a representation of the obsolescence relation. More precisely, let $m.sn$ and $m.bm$ represent respectively the sequence number and the bitmap associated with message m. Given two messages m and m' the protocol considers that $m \sqsubseteq m'$ if $m'.sn - k \leq m.sn < m'.sn$ and $m'.bm[m'.sn - m.sn]$.

The k-enumeration is not only extremely compact to be stored and transmitted over the network but also makes it very easy to compute the representation of transitive obsolescence relations using only shift and binary "or" operators. This favors time and space efficient algorithms and data structures to manipulate protocol buffers and determine obsolete messages. It also makes it very easy to compute, using the same efficient operators, the representation of composite updates, as required for the commit messages used to support multi-item multicast.

5 Performance Evaluation

5.1 Application

Although we try to keep the discussion of semantic reliability as generic as possible, performance evaluation depends on concrete data about the actual obsolescence relation. As such, to present meaningful performance numbers we have to choose an application: a distributed multi-player game. This is an interesting example it typically is not supported by group communication services:

- High availability of servers has not been high in the list of priorities of game developers, as in the past games were normally short-lived, and servers managed on a best-effort basis frequently by players themselves.

- Off-the-shelf group communication services have traditionally been geared toward applications without the stringent throughput requirements of highly interactive applications.

However, this scenario is bound to change as the number of multi-player games hosted by commercial services is growing. As a result of this trend, long lived games have been appearing in an attempt to keep players loyal to a server. In such systems, the need to preserve the server state and offer continuous service becomes an important concern. Therefore, it is extremely relevant to ease the task of replicating this type of servers in an efficient manner.

5.2 Update Patterns

We have inspected the code of Quake™ [16], an open-source multi-player game, to extract concrete obsolescence relations. The state of the game is modeled as a set of items. An item is any object in the game with which players can interact. The background is described separately as it is immutable. Each item is represented by a data structure that stores its current position and velocity in the 3D space. The same data structure may also hold additional type specific attributes, such as the players remaining strength.

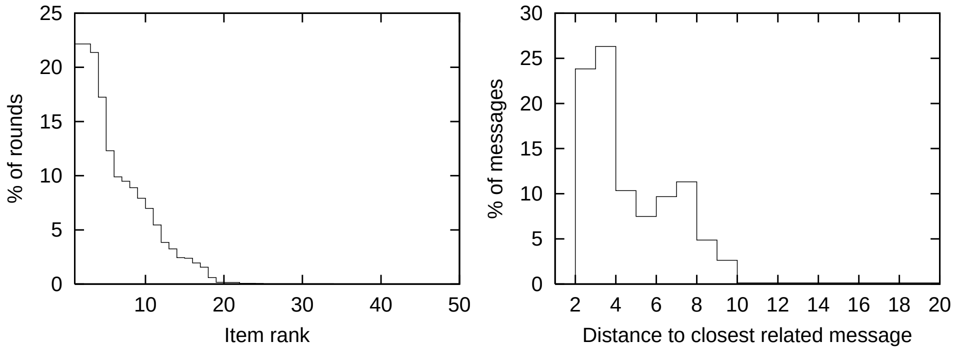
The game advances in rounds which correspond to frames that are displayed in players screens. Although the server tries to calculate 30 frames each second, this number can be reduced without loss of correctness. However, this degrades the perceived performance of the game hence the need to sustain a stable throughput. During each round the server gathers input from clients and re-calculates the state of the game. In each round, besides being updated, items can be created and destroyed. For instance, when a bullet is fired an item has to be created to represent it, and if a player is later hit, both the items of the bullet and the player have to be removed. The transmission of the updated state includes:

- Updated values of items, for instance, as their position is altered. These make previous values of updates obsolete as they convey newer values.

- Destruction and creation of items. These must be reliably delivered in order to ensure that items are kept consistent.

This application closely matches the multi-item message scenario described in Section 4. Therefore, we use the k-enumeration representation technique that we have described before with k equal to twice the buffer size.

We have instrumented the server of Quake to obtain experimentally the obsolescence patterns from real gaming sessions. We detect which items are changed at each round by monitoring internal functions used to update the system state and to disseminate changes to clients.



(a) Frequency of item modifications.

(b) Obsolescence distance.

Figure 3: Characterization of access to application state.

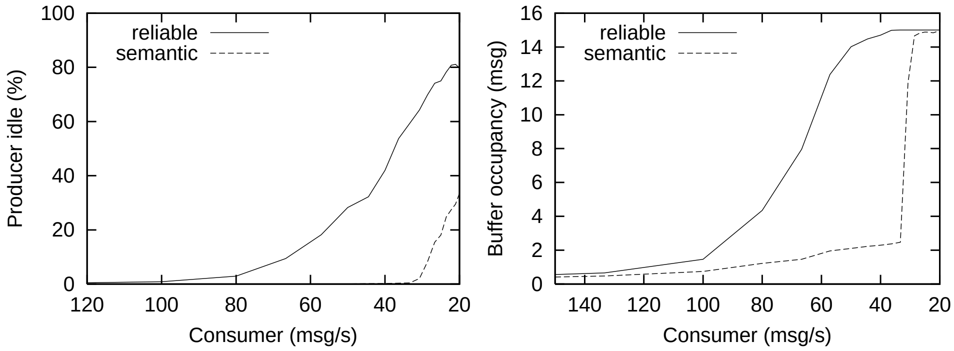
The results presented on this paper have been observed during a session with 5 players lasting for approximately 6 minutes and allowing us to record a total of 11696 rounds. This particular run was selected due to its length with a constant number of players.

From the traffic generated it was observed that a share of 41.88% of the messages never became obsolete. The obsolescence pattern of the remaining messages is related to the item update pattern. Although an average of 42.33 items were recorded active in each round, only an average of 1.39 items are modified. In addition, the results of Figure 3(a) show that a small number of items is modified frequently, while some items have not been modified at all during the measurement period. Therefore, consecutive updates of the same item are likely to be found close in the message stream. This is confirmed by Figure 3(b), which shows the distribution of distance between related messages. Notice that related pairs are usually close together (often within 10 messages of each other).

We have also collected data with other numbers of players. It can be observed that when more players join the game that the message rate increases, the share of messages that never become obsolete decreases, but the distance between related messages increases. This suggests that higher purging rates would be possible that those presented here, although at the expense of larger buffer sizes.

5.3 Simulation Model

In evaluating of the impact of purging we have used a high-level discrete event simulation. The use of simulation instead of a real protocol allows us to isolate performance degradation due to a slower receiver from other aspects of group performance. The network is modeled as $n \times n$ queues fully connecting all processes and it was configured with unlimited bandwidth in order not to be a limiting factor of system performance.



(a) Impact in the producer.

(b) Impact in buffer occupancy.

Figure 4: Sample runs of the simulation with a buffer holding 10 messages and increasingly slow consumer.

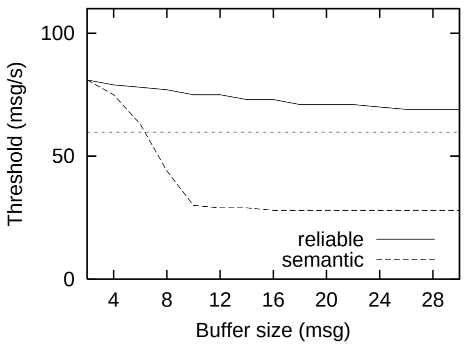
A producer injects traffic in one of the nodes according to the item update pattern recorded experimentally. Consumers are attached to all nodes. In the simulations, we show the impact of a single slow receiver in the group. Therefore, all processes except the slow one consume messages instantly; the time it takes for the slower process to consume each messages can be varied.

Each node implements the SVS protocol by managing local bounded buffers. When its delivery queue fills up, a node ceases to accept further messages from the network. Eventually, this will cause the outgoing buffers of the sender to be exhausted which, in turn, prevents further messages from the application from being accepted. At this point, throughput can only be sustained by expelling the slow member from the group. Note that the protocol must always reserve separate buffer space for control information and to allow group management function to operate, in particular to execute the view change procedure.

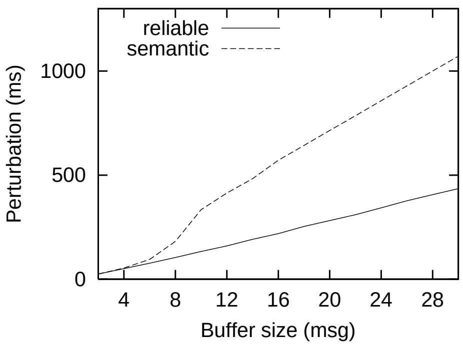
5.4 Simulation Results

Given a traffic profile and a buffer size, we can determine the minimum rate at which messages have to be consumed in order not to disturb the source. For instance, by selecting a buffer size of 15 messages, running the simulation with an increasingly slower consumer and measuring the amount of time the producer is blocked due to flow-control we obtain Figure 4(a). Notice that when using a reliable protocol the receiver has to be able to consume 73 msg/s in order not to disturb the sender more than 5%. When a semantically reliable protocol is used, the traffic profile and system configuration allow enough purging to leave the producer undisturbed until a receiver is limited to consume only 28 msg/s.

We also study the impact of SVS in the number of messages that need to be flushed in order



(a) Threshold value.



(b) Tolerated perturbation length.

Figure 5: Impact of purging in the performance of SVS.

to install a new view. This is related to buffer occupancy when a view change is triggered. Figure 4(b) presents the results of observing the amount of buffer used. Notice that between 73 to 28 msg/s, when purging is enough to prevent throughput degradation, this is achieved without buffers filling up. This is important as the amount of used buffer space impacts on the latency of the view change protocol, which must wait for all pending messages to be stable.

Of particular interest in such results are the points of inflexion of the curves of Figure 4(a). Figure 5(a) shows what is the lowest threshold value, for the degradation of a receiver, that can be tolerated (with less that 5% impact on the sender) as a function of the buffer size. The horizontal line shows which is the average rate of input traffic. In the presence of periodic traffic, a receiver could process messages at the average rate without affecting the group throughput. However, as it can be seen from the figure, due to the bursty nature of the game traffic pattern, when a reliable protocol is used, the receiver has to process messages at a faster pace (to accommodate the excess of messages during the bursts). As expected, it can be observed that larger buffers allow the reliable protocol to better accommodate message bursts. In any case, with a reliable protocol, the receiver's rate can never be lower than the average input rate, otherwise it eventually slows down the system no matter how large the buffers are. On the other hand, with SVS, slower receivers can be accommodated by increasing the buffer size which enables purging to be done. Notice that SVS is not effective for very small buffer sizes due to the distance among related messages.

The difference between the two lines of Figure 5(a) indicates the purging rate achieved by the protocol for each buffer size. The difference between the messages being produced and the messages being purged indicates the rate at which buffers fill-up for a given configuration. From this rate, we can also estimate the maximum length of the perturbation period that can be tolerated before the buffers are exhausted. As a function of the buffer size, Figure 5(b) shows for how long can be tolerated a receiver that completely stops to process messages. For instance, with a buffer size of 24 messages, a reliable protocol can only tolerate a perturbation of 342 ms while the SVS protocol can tolerate a perturbation of 857 ms. We conclude that SVS allows longer perturbations to be tolerated with the same amount of allocated buffer space. Since this is achieved at the cost of purging obsolete information, and not at the cost of storing additional messages, SVS as no negative impact on the latency of the view change protocol.

6 Related Work

The difficulty of ensuring stable high throughput with group communication systems has been pointed out in the context of stock exchange applications [23] and then further generalized to a larger class of applications [4]. This led to the introduction of a probabilistic reliable broadcast protocol which addresses throughput stability by dropping messages on processes that fail to meet performance assumptions [5]. The use of a probabilistic protocol results in an application programming model which, unlike ours, differs significantly from conventional view synchrony.

Although message semantics is here used to relax reliability, it has often been used for relaxing the ordering of messages. For instance lazy replication [17] relies on message semantics to relax causal order. Generic broadcast [19] is a relaxation of total order based on message semantics captured as a binary relation. The work on Optimistic Virtual Synchrony [25] also uses semantic information to alleviate the cost of view changes but, unlike our approach, does not address the issue of limiting the number of these changes. It would be interesting to combine these approaches with our proposal.

The Bayou [26] replication system is sensitive to semantics of update messages. However, it relies on programs embedded in the updates which makes the implementation much more complex. In contrast, our proposal uses a simple mechanism that fits general purpose protocols.

In the context of synchronous systems, the notion of time has been used to define obsolescence relations in the Δ-causal [2] and deadline constrained [24] causal protocols. These protocols allow timing constraints to be met at the cost of discarding delayed messages. Our protocol allows to express obsolescence relations that are not merely based on the passage of time. As described in the previous section, this makes it useful for applications other than strictly periodic traffic.

A primary-backup protocol which discards messages and provides real-time guarantees has also been proposed [29], although offering only a weak consistency model. In contrast, our proposal provides strong consistency and a generic multicast primitive which can be used for purposes other than primary-backup replication.

7 Conclusions

In this paper we have addressed the problem of sustaining high throughput in group communication systems in the presence of processes that may suffer performance perturbations. We have introduced a novel abstraction, called Semantic View Synchrony (SVS) and shown how it can be easily implemented by modifying existing implementations of View Synchrony (VS).

SVS exploits the notion of message obsolescence to accommodate performance perturbations without incurring in the disadvantages of previous approaches which relax reliability. Namely, our solution does not require the offered load to be limited due to a single slow process, it does not require the slow process to be immediately excluded when it exhibits transient delays, and avoids resources to be over-allocated to accommodate overload periods. On the other hand, SVS still retains the machinery that allows processes that have crashed to be expelled from the group while ensuring that group members have a consistent state when a new view is installed. Additionally, by not requiring buffer sizes to be over-dimensioned, SVS does not have a negative impact on the latency of view changes.

SVS is a key element in the design of a full group communication toolkit offering semantic reliable multicast services. Besides SVS, this encompasses also causally and totally ordered multicast [20]. Semantic reliability based on message obsolescence is also being considered as a generally desirable feature of multicast transport protocols [10].

To illustrate the advantages of SVS we have applied it to replicate the server of a distributed multi-user game. We have proposed efficient techniques to encode the obsolescence relation in this type of applications. Finally, we have collected experimentally data from a running application to obtain a realistic characterization of obsolescence for a concrete game. This information was used to feed simulations and allowed us to discuss the impact of relevant configuration parameters, such has the buffer size, on the performance of SVS.

References

[1] R. Arpaci-Dusseau and A. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Hot Topics in Operating Systems (HotOS 8)*, May 2001.

[2] R. Baldoni, R. Prakash, M. Raynal, and M. Singhal. Efficient Δ-causal broadcasting. *Intl. Journal of Computer Systems Science and Engineering*, 13(5):263–269, September 1998.

[3] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[4] K. Birman. A review of experiences with reliable multicast. *Software Practice and Experience*, 29(9):741–774, July 1999.

[5] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, 1999.

[6] A. Carzaniga, D. Rosenblum, and A. Wolf. Content-based addressing and routing: A general model and its application. Technical Report CU-CS-902-00, Dept. of Computer Science, Univ. of Colorado, January 2000.

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

[8] B. Charron-Bost, X. Défago, and A. Schiper. Time vs. space in fault-tolerant distributed systems. In *Proc. of the 6th IEEE Intl. Workshop on Object-oriented Real-time Dependable Systems (WORDS'01)*, Rome, Italy, January 2001. IEEE Computer Society.

[9] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, June 1999.

[10] S. Elf and P. Parnes. A literature review of recent developments in reliable multicast error handling. Technical report, CDT, Lulea Univ. of Tech., 2001.

[11] A. Gopal and S. Toueg. Inconsistency and contamination. In Luigi Logrippo, editor, *Proc. 10th ACM Symp. on Principles of Distributed Computing (PODC'91)*, pages 257–272. ACM Press, August 1991.

[12] R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn communication channels. Technical Report 98-278, Département d'Informatique, École Polytechnique Fédérale de Lausanne, 1998.

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

[14] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, January 2001.

[15] M. Hiltunen and R. Schlichting. Properties of membership services. In *Proc. 2nd. Intl. Symp. on Autonomous Decentralized Systems*, pages 200–207, April 1995.

[16] Id Software Inc. Quake Homepage. http://www.quake.com.

[17] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. *ACM SIGOPS Operating Systems Review*, 25(1):49–54, January 1991.

[18] K. Lin and V. Hadzilacos. Asynchronous group membership with oracles. In *DISC'1999*, pages 79–93, 1999.

[19] F. Pedone and A. Schiper. Generic broadcast. In *Proc. of the 13th Intl. Symp. on Distributed Computing (DISC'99, formerly WDAG)*, September 1999.

[20] J. Pereira. *Semantically Reliable Group Communication*. PhD thesis, Univ. of Minho, 2002. (to appear).

[21] J. Pereira, L. Rodrigues, and R. Oliveira. Semantically reliable multicast protocols. In *Proc. of the Nineteenth IEEE Symp. on Reliable Distributed Systems*, pages 60–69, October 2000.

[22] J. Pereira, L. Rodrigues, and R. Oliveira. Semantically reliable broadcast: Sustaining high throughput in reliable distributed systems. In P. Ezhilchelvan and A. Romanovsky, editors, *Concurrency in Dependable Computing*, chapter 10. Kluwer, 2002. (to appear).

[23] R. Piantoni and C. Stancescu. Implementing the Swiss Exchange Trading System. In *Proc. of The Twenty-Seventh Annual Intl. Symp. on Fault-Tolerant Computing (FTCS'97)*, pages 309–313. IEEE, June 1997.

[24] L. Rodrigues, R. Baldoni, E. Anceaume, and M. Raynal. Deadline-constrained causal order. In *The Proc. of the 3rd IEEE Intl. Symp. on Object-oriented Real-time distributed Computing*, March 2000.

[25] J. Sussman, I. Keidar, and K. Marzullo. Optimistic virtual synchrony. In *Proc. of the Nineteenth IEEE Symp. on Reliable Distributed Systems*, pages 42–51, October 2000.

[26] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of the 15th Symp. on Operating Systems Principles (SOSP-15)*, December 1995.

[27] P. Veríssimo, A. Casimiro, and C. Fetzer. The timely computing base: Timely actions in the presence of uncertain timeliness. In *Proc. Intl. Conf. on Dependable Systems and Networks (DSN'00)*, pages 533–542, New York City, USA, June 2000. IEEE Computer Society Press.

[28] R. Vitenberg, I. Keidar, G. Chockler, and D. Dolev. Group communication specifications: A comprehensive study. Technical Report MIT-LCS-TR-790, The Hebrew Univ. of Jerusalem and MIT, September 1999.

[29] H. Zou and F. Jahanian. Real-time primary-backup replication with temporal consistency guarantees. In *Proc. IEEE Intl. Conf. on Distributed Computing Systems (ICDCS'98)*, June 1998.