High-Performance Replicated Distributed Objects in Partitionable Environments *

Roy Friedman Alexey Vaysburd
Department of Computer Science
Cornell University

{roy,alexey}@cs.cornell.edu

July 16, 1997

Abstract

This paper presents an implementation of replicated distributed objects in asynchronous environments prone to node failures and network partitions. This implementation has several appealing properties: It guarantees that progress will be made whenever a majority of replicas can communicate with each other; it allows minority partitions to continue providing service for idempotent requests; it offers the application the choice between optimistic or safe message delivery. Performance measurements have shown that our implementation incurs low latency and achieves high throughput while providing globally consistent replicated state machine semantics. The paper discusses both the protocols and interfaces to support efficient object replication at the application level.

*A preliminary version of this work was accepted to SRDS 97. This research was supported by ARPA/ONR grant N00014-96-1-1014

1 Introduction

Object replication is one of the principal ways to provide fault-tolerance and high availability in a distributed system. A common requirement from a replicated system is that actions of all replicas be indistinguishable from those of a single fault-tolerant process.[1] The *replicated state machine* approach was suggested in [18, 24] as a way to provide such a consistent behavior: With this approach, all object replicas run identical state machines, and all communication in the system is performed via totally ordered multicasts, where delivery of multicast messages is implemented via invocations of appropriate methods of object replicas.

Since all replicas start in the same state, and proceed through exactly the same sequence of multicast-message-delivery method invocations, they advance through the same sequence of states and perform the same sequence of actions. This property ensures that actions taken by the system as a whole appear to the outside world as if they were performed by a single object.

Implementations of replicated state machines typically differ in the assumptions they make of the environment, and whether they are *optimistic* or *pessimistic*. Optimistic implementations, *e.g.*, [7, 19], tend to deliver messages fast, yielding good performance, although the local state of individual replicas may become invalid under certain failure scenarios as a result of delivering messages that never become "authoritative". On the other hand, pessimistic implementations, *e.g.*, [2, 6, 16], deliver messages only when it is safe to do so, thereby avoiding the risk of having to invalidate the state of a replica. This is achieved at the cost of additional communication rounds.

Many existing implementations of replicated state machines ignore the issue of network partitions [8, 23]. Other implementations either have a limited support for partitions, but might block forever even after all partitions have been reconciled [6, 7], or are pessimistic [2, 16, 22]. A proper handling of partitions is vital for applications targeted for large scale or wide-area environments where partitions are common. However, the high communication cost associated with pessimistic approaches may be prohibitive for many applications.

In this paper we propose an implementation of replicated distributed objects using the replicated state machine paradigm. Our implementation can tolerate network partitions, and only requires object replicas to log a single bit once, at initialization time, on stable storage. Our implementation is modular (layered) and gives an application the freedom to choose, at run time, and within the same framework, between stronger properties (safe message delivery) and improved performance (optimistic delivery). Even in the safe-delivery case, the performance of our solution is as good and in many cases better than performance of any other pessimistic implementation. Modularity of our implementation allows to efficiently address orthogonal issues such as reliability of message delivery, total ordering of messages, global consistency (primary views), state transfer, and flow control. In particular, we provide several implementations of certain protocol layers, with different performance characteristics and tradeoffs. Having multiple implementations of different layers and being able to interchange between them allows to

[1] This is similar to the *serializability* requirement in databases and the *sequential consistency* requirement in distributed shared memories.

choose at run time the protocol configuration which best matches the requirements of a given application, both in terms of logical properties and performance. Note that our approach allows us to accurately measure the added cost of individual layers and thereby let application developers get a realistic feeling for the cost of stronger logical properties.

As mentioned above, with our approach multiple concurrent partitions are allowed to exist in the system. However, we guarantee that at most one of those partitions is *primary*. All group members know if they are in a primary partition. Only members of the primary partition are allowed to take actions that can change their state. Whenever two partitions remerge, they perform a *state transfer* that reconciles their states, which is necessary to ensure linearizability of the progress of the system as a whole. During state transfer, the partition with an older state adopts the state of the more up-to-date partition. The primary partition is always guaranteed to be the one with the most up-to-date state.

Note that as indicated in [9], we cannot guarantee that a primary partition will exist at all times, because link failures or communication delays are possible. However, we do guarantee that if a majority of processes do not crash, a primary partition will be restored as soon as the network becomes stable again, and the initial state of this primary partition will be the final state of the previous one. We follow a standard quorum-based dynamic replication paradigm [10] requiring a message to be delivered by a majority of group members in order to become authoritative. However, the optimistic implementation we present uses one phase communication for delivering messages, which yields good performance.

In order to keep the discussion in this paper focused, our protocols have been implemented in Horus [25], and therefore our discussion sometimes deals with specific details of the Horus implementation. We also present a programming interface, available in Horus, that supports replicated distributed objects at the application level [26]. However, the proposed protocols and interfaces can easily be applied to any group communication system that supports partitioned operation, including Transis [11] and Relacs [4].

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 presents basic definitions and concepts. Section 4 describes how primary views are used to create a replicated state machine semantics in partitionable networks, and presents the protocol that we developed for this. The tradeoffs between optimistic and safe delivery and our implementation of global safety are discussed in Section 5. Performance analysis is presented in Section 6. An object-oriented interface supporting the replicated state machine functionality at the application level is described in Section 7. We conclude with a discussion in Section 8.

2 Related Work

Fault-tolerant protocols for implementing replicated state machines that assume a synchronous environment, or at least the existence of a perfect failure detector, but do not support network partitions, have been proposed in [23]. In a recent paper, Bressoud and Schneider presented a hardware-based implementation of a replicated state machine using off-the-shelf workstations, called Hypervisor [8]. This solution is workable only in tightly coupled systems, and is definitely not suitable for wide-area distributed environments. Also, Hypervisor provides replicated state machine semantics at the granularity of machine instructions, while the granularity provided by our solution is at the level of updates to the replicated state as defined by the application.

The ISIS toolkit [7] provides the `abcast` primitive and the notion of a *primary partition* as abstractions for implementing the replicated state machine semantics in asynchronous distributed environments. In ISIS, processes that are removed from the primary view are forced to quit the system. Hence, network partitions may cause ISIS to lose a majority in the primary view and thus block forever, even if these partitions later remerge. Moreover, by shutting down processes that are removed from the primary view, ISIS does not allow minority partitions to execute idempotent operations, even though many applications could benefit from such a feature.

Phoenix [19] is a recent group communication system that also supports the primary partition model. Phoenix tries to avoid the availability problems of ISIS in the following way: Whenever a subset of the primary partition suspects that a majority of processes have failed, it suspends the execution until it can remerge with enough members to create a primary partition. This approach is somewhat similar to our implementation, in avoiding the permanent loss of the primary partition. However, unlike our solution, Phoenix does not allow minority partitions to install non-primary group views and perform idempotent actions. Also, Phoenix does not fully address the issues involving state transfer. It is important to treat state transfer in the context of providing a higher-level semantics, such as that of a replicated state machine, in order to ensure it is performed in a meaningful way.

Pessimistic implementations of global total ordering that can tolerate network partitions have been proposed in [1, 2, 16, 22]. Being pessimistic, they require more communication rounds to perform an operation.[2] Also, these solutions require all messages to be logged on stable storage, which adds a substantial overhead. Our implementation only requires logging of a single bit, which is performed by every process only once at initialization. As a tradeoff, with our approach the state of a member is lost in a crash failure and can only be recovered by means of state transfer from surviving members. Consequently, our protocols will block if a majority of processes in the group crash simultaneously, whereas the implementation in [16] can sustain any number of crashes (assuming that failed processes are eventually restarted). Also, the protocols in [16] can make progress even if a majority view can never be formed due to link failures, which makes that approach suitable for WAN environments with very low quality of communication. However, many important distributed applications (such as replicated databases) have rather demanding performance requirements that render message logging impractical. Those applications are usually deployed in environments with higher quality of communication links, where a majority of processes will be connected most of the time. Our protocols are best suited for use in such high-performance systems, running over

[2] Note that [2] claims to deliver messages as soon as they arrive from the transport layer, without end-to-end acknowledgments, but it assumes that the transport layer provides total safe (uniform) delivery, which requires at least 2 communication rounds at the transport level.

partitionable yet reasonable-quality networks.

Consul [20] is another implementation of a replicated state machine that is also designed to be highly modular, in the framework of the x-kernel [14]. Consul, however, does not handle network partitions, and require processes to periodically log their state on stable storage.

Atomic transactions also implement a replicated state machine. However, most implementations of atomic transactions either cannot recover from network partitions, or are pessimistic and thus incur high overhead [6, 17].

In summary, we have mentioned several implementations that provide some of the following properties: being modular, allowing the application to choose at run time between optimistic and pessimistic delivery, being able to handle network partitions, not requiring logging of the state on stable storage, and providing good performance. Our solution is novel in being the first one to combine all of these properties.

3 Basic Definitions and Building Blocks

3.1 Replicated State Machine

We assume an asynchronous system with general omission failures. The system consists of a group of application processes, each running a deterministic *state machine* [18], and communicating by sending *messages* to each other. An application process' state machine is specified by a set of its internal configurations, or *states*, and a set of *transitions* between states. Each application process in the group runs the same state machine, starting in the same initial state. All state transitions are triggered by incoming (delivered) messages. An application process may produce some deterministic *output* associated with a transition, such as sending messages or doing some externally observable actions. Thus, any two application processes that have delivered the same sequence of messages will be in the same state.

3.2 System Architecture

Our implementation assumes a layered architecture, such as the one presented by Horus [25]. It is designed to reside on top of a system which provides *totally ordered broadcast* in a *partitionable virtually synchronous* environment [12, 21], and below the application level. (See illustration in Figure 1.) In the specific example of Horus, totally ordered broadcast is implemented as a separate layer on top of the partitionable virtual synchrony layer. However, this separation is not necessary for our implementation.

3.3 Partitionable Virtual Synchrony

There are several variants of partitionable virtually synchronous models. These include, e.g., strong virtual synchrony [12], extended virtual synchrony [21], and a few variations of partial

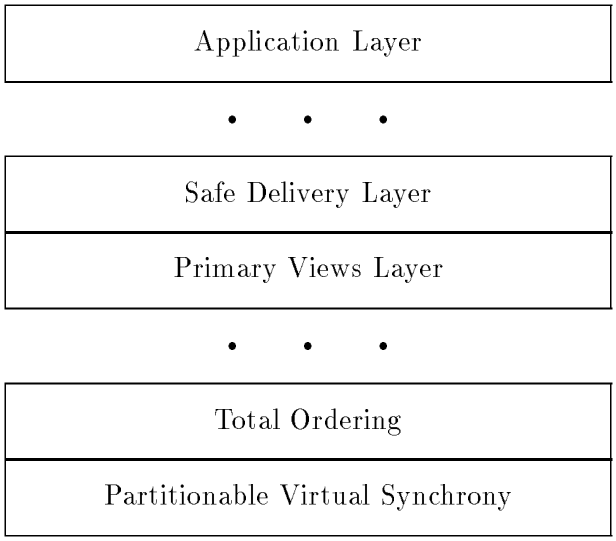

Figure 1: A Layered Protocol Architecture

view synchrony [5]. The exact differences between these models is beyond the scope of this paper. Instead, here we only outline those properties which are common to all these models, and happen to be useful for our implementation of replicated state machines.

In partitionable virtual synchrony models, each process has a view of the group (list of "accessible" members) at any time, while allowing multiple concurrent views to be installed at different processes. Most partitionable virtually synchronous models can be implemented in a non blocking way, i.e., the implementation guarantees that a new view will eventually be presented to the application. These implementations, however, may leave the group without a globally consistent state.[3] In this paper we discuss how to guarantee global consistency based on such models.

A view is typically installed in two phases. In the first phase, a process *proposes* the view by sending a view message to a set of processes. In the second phase, when a process receives a view message, it may decide to *accept* it (*i.e.* commit to the new view locally). The partitionable virtual synchrony layer guarantees the following properties regarding installation of new views:

Property 3.3.1 (Validity) *If a process accepts a view, then this view was proposed by some process.*

[3] Babaoglu et al showed that every implementation of strong partial view synchrony may block [5]. However, this result is based on the fact that strong partial view synchrony does not allow concurrent overlapping views. Other definitions of partitionable virtual synchrony (including ours) do not require this property, and therefore can be implemented in a non-blocking way.

Property 3.3.2 (Self-Inclusion) *If a process proposes or accepts a view, it is included in it.*

Property 3.3.3 (View Causality) *View messages are delivered in causal order.*

Property 3.3.4 (Agreement on View Ordering) *If two processes p and q both accept views V_1 and V_2, then p and q accept V_1 and V_2 in the same order.*

Property 3.3.5 (Agreement on Successors) *If V_1 and V_2 are two consecutive views accepted by a process, then V_1 and V_2 are consecutive at* any *process that accepts both V_1 and V_2.*

In addition, the partitionable virtual synchrony layer provides an *agreement* on sets of messages delivered by processes between view changes. It is assumed that all messages are multicast to the entire current view of the sender:

Property 3.3.6 (View Atomicity) *If V_1 and V_2 are two consecutive views accepted by p and q, then p and q deliver the same set of messages after accepting V_1 and before accepting V_2.*

3.4 Maintaining Coherence within a View

The total ordering layer (Figure 1) guarantees that all processes in a view deliver messages in the same order. More formally, it guarantees the following property:

Property 3.4.1 (Strong Prefix) *Suppose both p and q accept a view V. Let S_p and S_q be the sequences of messages delivered by p and q in V. Then either S_p is a prefix of S_q, or S_q is a prefix of S_p.*

In a system with a partitionable virtual synchrony model, there are scenarios in which a message may need to be dropped to prevent a violation of the strong prefix property (Property 3.4.1).

The following *Active Replication* property is a key property in providing the replicated state machine semantics:

Property 3.4.2 (Active Replication)

- *If V_1 and V_2 are two consecutive views accepted by p and q,* and *p and q are in the same state when they accept V_1, then (1) p and q are in the same state when they accept V_2,* and *(2) p and q produce the same output (including the sequence of externally observable actions) after accepting V_1 and before accepting V_2.*

- *Suppose p and q accept a view V in the same state. Let O_p and O_q be the output produced by p and q in V (including the sequences of externally observable actions). Then either O_p is a prefix of O_q, or vice versa.*

Property 3.4.2 provides the replicated state machine semantics for processes in a view between two consecutive view changes. However, since processes from different views do not communicate with each other or otherwise coordinate their actions, it is possible that concurrent non-communicating views of the same group will be in mutually inconsistent states. Thus, although each particular view may satisfy Property 3.4.2, and thereby be *internally* coherent, there will be no consistent *global* state among all views. As a result of inconsistencies between views, the entire system will not have the semantics of a replicated state machine.

The *primary views* approach allows to maintain global consistency of the group state and provide the replicated state machine semantics for the entire system. We describe our implementation of primary views in the following section.

4 Primary Views That Can Tolerate Partitions

In this section, we discuss the properties of primary views and their relation to the replicated state machine semantics, and describe our implementation of primary views in Horus. We assume a group has a fixed size known to members at the initialization time.

4.1 Replicated-State Properties of Primary Views

Primary views are a subset of views installed in the system during an execution; a primary view must include a *majority* of group members. This requirement naturally imposes a linear ordering: The order of two primary views V_1 and V_2 is the order in which they were accepted by a process in their (necessarily non-empty) intersection. By Property 3.3.4, provided by the membership layer, this definition does not depend on the choice of a process in $V_1 \bigcap V_2$.

We assume that whenever processes from two separate views merge to form a joint view, processes from one of these views adopt the state and message history of processes from the other view. In particular, if one of these views is the most recent primary view, then its state will prevail. This is called *state transfer*.

We say that a process *accepts* a message either if it actually delivers the message, in which case we say that it *explicitly accepts* the message, or if the message is included in the history adopted by this process as a result of a state transfer, in which case we say that the process *implicitly accepts* the message. Messages which are accepted (explicitly or implicitly) by a majority of processes are called *authoritative*. These messages will be included in histories of all subsequent primary views, and actions initiated by delivery of such messages will have a permanent effect.

Recall that when a process p adopts the state of a process q, it also adopts q's history. Thus, the order in which p accepts messages is the same as the order in which q accepts them. In particular, there is a naturally defined linear ordering of authoritative messages. The order of two authoritative messages, m_1 and m_2, is the order in which they were accepted by a process that accepted both of them (such a process necessarily exists). This order is the same for any process that accepts both m_1 and m_2.

Properties 3.4.1 and 3.3.6 guarantee that processes that have the same state at the beginning of a view will continue to have identical states when they both accept the same new view. Also, by the assumption that processes from one view adopt the state of processes from another (more up-to-date) view when views are merged, all processes that accept a primary view are in the same state when the view is installed. This state is called the *accepting state* of the primary view. Thus, all processes in a primary view start in the same state and deliver the same sequence of messages, and therefore execute the same sequence of actions and produce the same output, as provided by Property 3.4.2. In particular, all processes that deliver an authoritative message are in the same state when this message is delivered, known as the *accepting state* of the message. Also, all these processes produce the same output following the delivery of this message and advance to the same new state.

A state of a process is an *authoritative state* if it is an accepting state of a primary view or an accepting state of an authoritative message. Note that authoritative states of a group are linearly ordered. Thus, the restriction of an execution of the group to authoritative states and authoritative messages is *linearizable*: It can be represented by a sequence of events, just as an execution of a single process. We define the *group's state history* as the sequence of authoritative states accepted through an execution of the group. Correspondingly, the *group's message history* is the sequence of authoritative messages delivered through an execution.

We are ready to formulate the following property:

Property 4.1.1 (Replicated State Machine) *Suppose* $M = (m_0, \ldots, m_k)$ *is the* group message history *and* $S = (S_0, \ldots, S_k)$ *is the* group state history *of a primary-view execution of a group of processes, each running the same state machine* $\aleph$*. Then a single process running* $\aleph$ *with* M *as its message history will pass through the same sequence of states,* S*.*

4.2 Implementing Primary Views

The main goals of our protocols are to merge concurrent views (partitions), to form larger views, notify a primary view that it is primary, and do state transfer so that the initial state of every newly formed primary view is the last authoritative state of the previous primary view.

We assume that each member maintains a state version number, which obeys the following properties: (a) whenever two members hold the same state version number, then their state is the same, and (b) a state version number N_1 of a process p is larger than a state version number N_2 of another process q if the state of p is more advanced than the state of q. We discuss how state version numbers are maintained shortly.

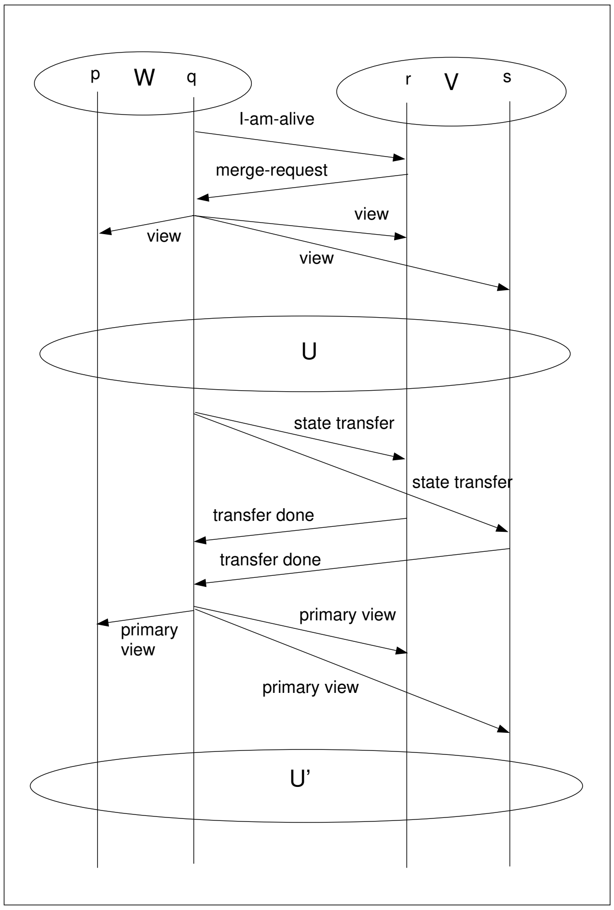

Figure 2: Merge Protocol and State Transfer

A time-space diagram of the view merge/state transfer protocol is shown in Figure 2. In order to merge views, an `I-am-alive` message is broadcast periodically by each view. In order to reduce the number of messages in the system, each view has a *contact* member; the contact is the only member that sends `I-am-alive` messages.

When the contact r of a certain view V receives an `I-am-alive` message, it replies by sending a `merge_request` to the contact q of the view W that sent the `I-am-alive` message. (Other members of V ignore `I-am-alive` messages.) The `merge_request` message includes all members of W. When q receives this `merge_request` message, then if q's view is not already merging with another view, and if the state version number of r is smaller than the state version number of q, then a new view U, which is a union of both V and W, can be created.

In order to install U, q proposes U to all members of U, including itself, by sending a `view` message to them, which specifies the last authoritative state of W. When the proposed members of U accept it, they install this view by delivering a **view** event to the application. At this

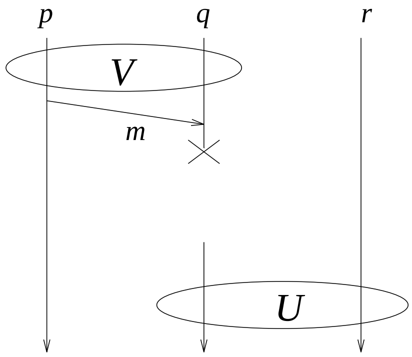

Figure 3: An example. In this scenario, both V and U may be primary, even though when U is installed, its initial state does not reflect the last authoritative state of V. To disallow this, we introduce the incarnation bit.

point, some members of U may need to do a state transfer in order to reconcile their state with the state of more advanced members. After completing the state transfer, a member sends an `xfer_done` message to q. When q receives `xfer_done` messages from all processes that need to do state transfer, it reinstalls U as a primary view $U\prime$.

A complication to this scheme comes from the fact that failed processes may eventually restart and will need to be brought back into the system in a safe way. It is assumed that processes do not log their state on a stable storage, so a reincarnation of a failed process starts up in an initial state. Now, consider the following scenario, as depicted in Figure 3, in which there are 3 processes in the system, p, q, and r, in which p and q initially form a primary view V. Following this, q crashes and then recovers, but does not retain its previous state. If q becomes connected with r, but not with p, then q and r may form a primary view U. However, the initial state of this view is "older" than the last authoritative state of the previous primary view. In particular, it may happen that authoritative messages delivered in the previous primary view are not reflected in the state of the newly formed primary view.

To overcome this problem, each group member uses a *recovery bit*, which is logged on non-volatile storage. The recovery bit is initially set to zero, and is set to one after the process delivers its first view. This allows the process to determine at the initialization time whether it was restarted after a crash, or if this is an initial incarnation. A reincarnation of a process is called a *zombie* until it becomes a member of a primary view. When computing whether a view has a majority of group members towards deciding if the view is primary or not, zombie processes do not count.

We now discuss how to maintain correct state version numbers. A version number has two fields, which are a primary view sequence number and a message sequence number. The primary view sequence number is the ID number of the last primary view that the process was a member of. The message sequence number is the number of messages delivered to this process by totally ordered broadcasts during that primary view. Recall that since primary views must include a majority of the processes, every newly formed primary view must include at least one member of the previous primary view. The contact that proposes the view sets the sequence number of the new primary view to a value higher than the maximum sequence number of all previously installed primary views. Hence, for any two version numbers $N_1 = (v_1, n_1)$ and $N_2 = (v_2, n_2)$, we define $N_1 < N_2$ if $v_1 < v_2$, or $v_1 = v_2$ and $n_1 < n_2$. Naturally, $N_1 = N_2$ if $v_1 = v_2$ and $n_1 = n_2$.

Note that this scheme guarantees that if state version numbers of two processes p and q are the same, then their states are identical, and if the state of p is more advanced than the state of q, then the state version number of p is larger than the state version number of q, as needed.

A pseudocode description of the protocol appears in Appendix A.

5 Globally Consistent Replication

5.1 Authoritative Agents

In this section we discuss global consistency from an application's point of view. The Replicated State Machine Property (introduced in section 4) guarantees that messages accepted by majorities of group members are delivered to all members in the same order – regardless of group partitions/merges and member crashes/recoveries. However, there is a practical issue of determining when and whether a message has become authoritative, i.e., accepted by a majority of group members and therefore safe to act upon. In particular, the entity responsible for issuing authoritative actions needs to collect message stability information and has to defer actions until corresponding messages become acknowledged by a majority of group members. We call such an entity an *authoritative agent.*

There are at least two ways to implement globally consistent replicated state machine semantics, depending on whether the authoritative agents are *members* or *clients* of the group. In the following sections we discuss the *globally safe* implementation, where group members are the authoritative agents, and the *optimistic* implementation, in which group clients are the authoritative agents. We will consider a distributed database system as an example application. In this setting, each group member holds a database replica maintaining a local copy of the global state.

5.2 The Optimistic Approach

Our implementation as described so far supports the optimistic approach. In systems where group members are not acting as authoritative agents, a member can deliver a message as soon as it has been received, without waiting for acknowledgements from a majority. With this "optimistic" approach, it is possible that a delivered message will never become authoritative (delivered by a majority), and therefore the local state might need to be rolled back. Thus, in order to issue an authoritative action based on the global state, a client has to collect replies from a majority of group members so that it knows the state has become authoritative and will never be rolled back.

The *optimistic* model is suitable for systems where writes dominate reads, for example on-line billing systems used by telephone companies. In a typical application of this type, the state of a group replica is kept in memory and local updates do not involve any disk writes, which allows to sustain thousands of updates per second. Also, the current global state is periodically dumped to disk. The disk images of all dumped local states are then compared to each other in order to reconstruct the authoritative state of the system. However, this work is done off-line and therefore does not affect the performance of the entire system. Note that in this example, on-line updates made by individual group members can be rolled back, and clients (off-line processing applications) act as authoritative agents.

5.3 The Global Safety Approach

In systems where each database replica acts as an authoritative agent doing updates to the local state, an update is made only when it is authoritative. This guarantee simplifies the implementation of database clients: It suffices to query just one group member (database replica) to learn about the authoritative state of the group.

The requirement that authoritative actions will never need to be rolled back implies that a member can deliver a message to the application, possibly triggering an authoritative action, only after the message has been acknowledged by a majority of group members and thus is known to be authoritative. This imposes an additional load on group members including buffering of messages and propagation of message stability information.

The *global safety* model is appropriate for systems where read operations dominate state updates, so that the additional price paid by group members for safe (majority-acknowledged) message delivery is justified. Typical transactional database systems are a good example of applications with the majority of operations being reads.

5.4 Implementing Global Safety

We have implemented *global safety* with an additional protocol layer that is placed on top of the primary-views layer in a protocol stack. During normal operation, the global-safety layer buffers incoming messages until they become majority-stable, and hence authoritative, at which point it delivers the messages to the application. Also, in order to ensure consistent state transfers, this layer maintains the version number of the current *authoritative prefix*, which consists of messages delivered to the application, and the *non-authoritative suffix*, consisting of buffered messages waiting for acknowledgements from a majority. During a state transfer, the authoritative prefix of a group member will never be rolled back; it will either grow or stay the same. However, the suffix of non-authoritative messages buffered at the global-safety layer may need to be rolled back or overwritten with a suffix of messages with a higher version number.

It is a responsibility of the global-safety layer to determine the direction in which a state transfer should proceed for both the authoritative prefix and the non-authoritative suffix, and to notify the application accordingly. The application layer is then responsible for transferring the authoritative part of the state, whereas the global-safety layer transfers the non-authoritative suffix.

Together with the primary-views, total ordering, and virtual synchrony layers, the global-safety layer provides the following property:

Property 5.4.1 (Global Ordering) *Let S_p and S_q be the sequences of messages delivered by group members p and q at arbitrary points in their execution. Then either S_p is a prefix of S_q, or S_q is a prefix of S_p.*

While *Global Ordering* is a safety property, progress is guaranteed as long as a majority of group members do not fail and whenever a majority can communicate with each other.

A pseudocode description of this protocol appears in Appendix B.

6 Performance Analysis

6.1 Preface

The protocol layers discussed in this paper have been implemented in the Horus system developed at Cornell [25]. Recall that Horus supports a layered architecture in which simple protocols are stacked on top of each other to obtain the desired functionality, as illustrated in Figure 1. In particular, an optimistic replicated state machine can be formed by running a total ordering layer with primary views virtual synchrony layers, while a globally safe implementation can be obtained by adding the global-safety layer to the above.

Both the optimistic and the globally safe approaches require a total ordering layer, and hence their performance heavily depends on the total ordering protocol that is used in their implementation. As it is impossible to experiment with all existing total ordering protocols, we have chosen for this study two protocols that, based on previous work [13], were expected to yield good performance. These protocols are the *dynamic sequencer* protocol and the *rotating token* protocol, both employing message packing techniques for improved performance [13].[4]

[4]In our framework, all that is required in order to utilize a new total ordering protocol is to replace the total ordering layer, while the rest of code remains the same. Thus, in the event that a superior total ordering protocol is found, it is fairly easy to incorporate it into the system, and benefit from its improved performance.

Stack name	Properties
PVSYNC	Primary views virtual synchrony
DYNSEQ:PVSYNC	Optimistic delivery using sequencer-based total ordering
TOKEN:PVSYNC	Optimistic delivery using token-based total ordering
SAFE:DYNSEQ:PVSYNC	Globally safe delivery using sequencer-based total ordering
SAFE:TOKEN:PVSYNC	Globally safe delivery using token-based total ordering

Figure 4: Properties provided by various stacks

In the dynamic sequencer protocol, one process is designated to be the *sequencer*; all messages are send point-to-point to the sequencer, who then forwards them to the entire group. In the rotating token protocol, a *token* is constantly rotating in the group, and a process must wait for the token in order to send messages. These protocols are described in great detail in [3, 13, 15].

We have measured latency and throughput for group sizes ranging from two to five members with the following protocol stacks providing primary views: (1) primary views virtual synchrony (abbreviated PVSYNC); (2) dynamic sequencer based total ordering protocol [15] over the primary views virtual synchrony stack (abbreviated DYNSEQ:PVSYNC); (3) rotating token based total ordering protocol [3] over the primary views virtual synchrony stack (abbreviated TOKEN:PVSYNC); (4) Global-safety layer over the dynamic sequencer based total ordering layer and the primary views virtual synchrony stack (abbreviated SAFE:DYNSEQ:PVSYNC); (5) Global-safety layer over the token based total ordering layer and the primary views virtual synchrony stack (abbreviated SAFE:TOKEN:PVSYNC). (See Figure 4). Options 2 and 3 corresponds to the optimistic approach, while Options 4 and 5 corresponds to the globally safe delivery approach. The difference between Option 2 (Option 4) and Option 3 (Option 5) is in the total ordering protocol being used, as discussed above.

6.2 Methodology

The performance tests were run using five Sparc 10s and Sparc 20s connected by a 10Mbps Ethernet. The following methodology was employed in those tests: For latency and throughput measurements, all group members were sending multicasts messages in rounds, each member sending one message in a round in latency tests and 100 messages in a round in throughput tests. After a group member received all messages sent by all members in a round, it immediately proceeded to the next round. Upon completion of the test (typically 100 rounds), the average throughput for the test and duration of a round (the mean) was computed. The tests were run multiple times, and medians of aggregate results were computed for each combination of test parameters. The test results turned out to be quite stable, so that best-case performance is close to median numbers.

6.3 Results

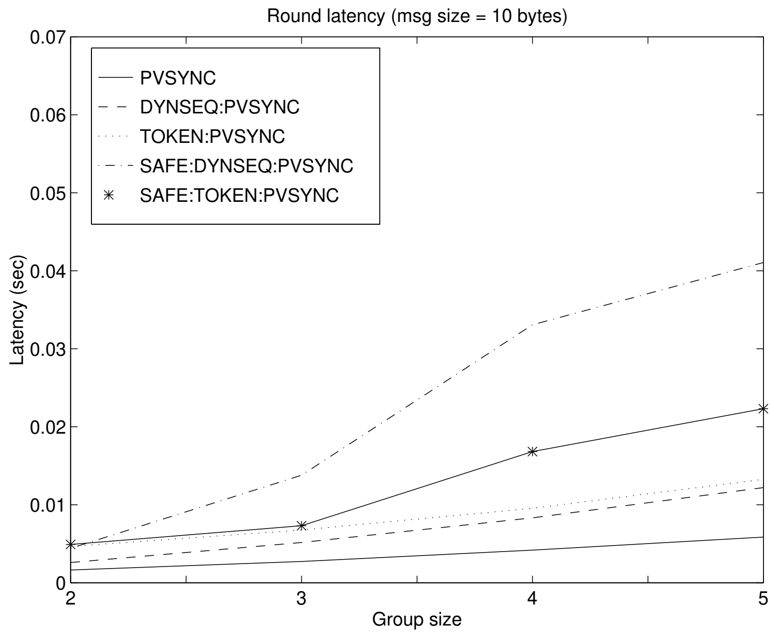

Figure 5: Latency per round vs. group size

Figure 5 shows the measured latency as a function of the group size, while Figure 6 shows the measured throughput per member (upper graph), and the aggregate throughput for the whole group (bottom graph). As can be seen from these graphs, the round latency of PVSYNC, DYNSEQ:PVSYNC, and TOKEN:PVSYNC protocol stacks increases linearly with the group size, ranging from 1.6 to 5.9 ms for the PVSYNC stack, 2.6 to 12.2 ms for the DYNSEQ:PVSYNC stack, and 4.7 to 13.3 ms for the TOKEN:PVSYNC stack. From this we can deduce that total ordering adds about a factor of two to the latency of unordered communication. Also, when safe delivery is not required, the sequencer based protocol is slightly faster than the token based protocol for small groups, but as the groups sizes grows, the token based protocol becomes better. This can be explained by the inherent flow control mechanism of token based protocols: Since processes wait for the token to send messages, there are fewer collisions, and hence we get better performance.

This phenomenon is becoming even more acute when globally-safe message delivery is required. Here too, the sequencer based protocol is slightly faster than the token based one for small groups (4.5 ms vs. 4.9 ms), but becomes much slower for larger groups (48.8 for sequencer vs. 22.3 ms for token). The graphs for both SAFE:TOKEN:PVSYNC and SAFE:DYNSEQ:PVSYNC show a peculiar "step" between group sizes three and four. This can be explained by the fact that in order to deliver a message safely, a process must wait for acknowledgments from a majority of group members. Thus, when going from three to four members, it becomes necessary to wait for two acknowledgements instead of only one. Such

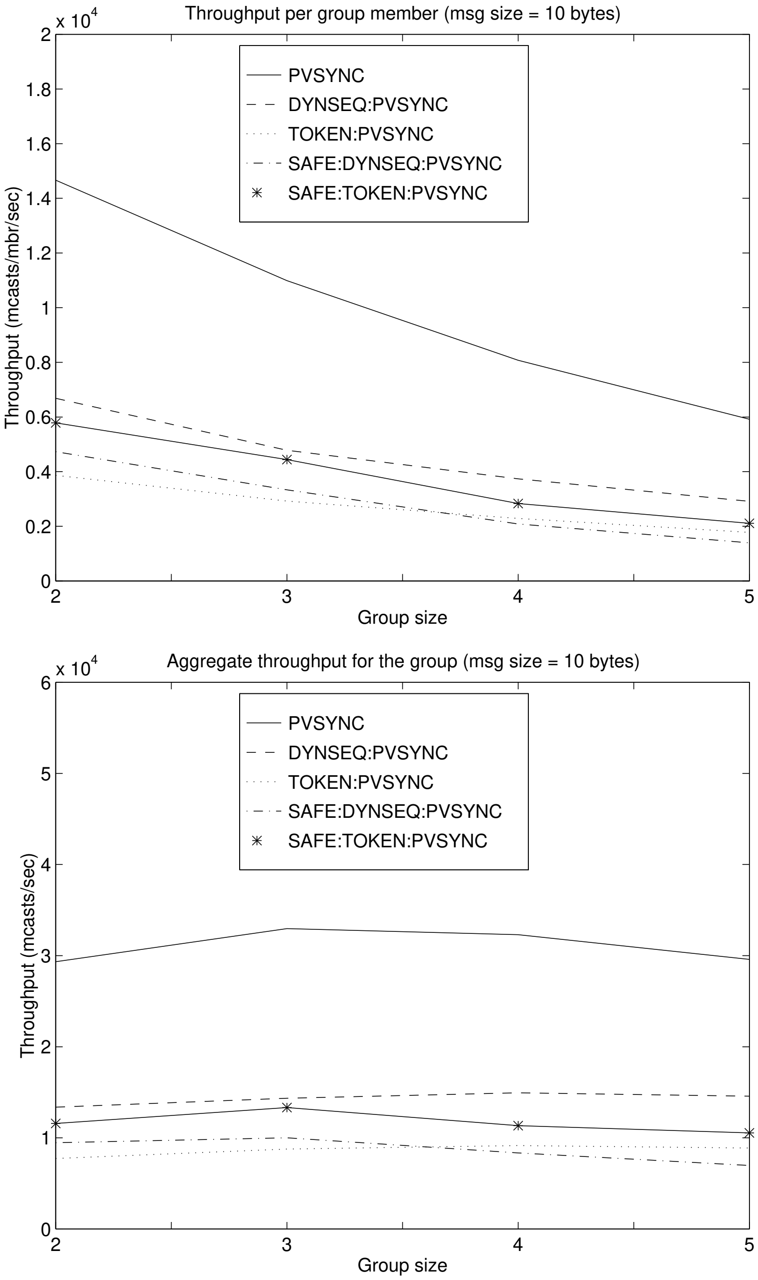

Figure 6: Throughput measurements. The top graph depicts throughput per member, while the bottom graph depicts aggregate throughput for the whole group.

an increase in the number of required acknowledgements does not happen when switching the group size from four to five members, since three members are enough to form a majority in five-member and four-member groups alike.

The best throughput numbers were obtained with the PVSYNC stack (ranging from 14,666 to 5918 messages per member per second, depending on size of the group). When safe delivery is not required, the dynamic sequencer-based total ordering has consistently outperformed the rotating token protocol (numbers ranging from 6683 to 2915 messages per member per second with the DYNSEQ:PVSYNC stack and 3865 to 1776 messages per member per second with the TOKEN:PVSYNC stack). However, adding the global-safety layer changes the picture: The throughput of the SAFE:TOKEN:PVSYNC stack is higher than the throughput of the SAFE:DYNSEQ:PVSYNC stack for all group sizes (5787 – 2110 vs. 4909 – 1271 messages per member per second respectively).

It appears that the added cost of safe delivery does not significantly affect throughput numbers. This is reasonable considering the fact that the added latency involved with safe delivery is caused mainly by the need to wait for acknowledgements, and not by doing significantly more work. Thus, safe delivery only shifts the delivery time, but does not hurt the capacity of the system.

Furthermore, the throughput of SAFE:TOKEN:PVSYNC is surprisingly *higher* than the throughput of the smaller TOKEN:PVSYNC protocol stack, which shows that the choice of a total-ordering protocol and flow control policies has a bigger impact on performance than the overhead added by an additional layer. Recall that in our implementation messages are packed. Thus, delaying messages by the global-safety layer allows us to pack more messages in each send operation, which effectively *increases* the throughput of the TOKEN:PVSYNC protocol stack (although the latency naturally decreases). In our implementation of the rotating token protocol, the token keeps cycling among the group members. Once a member receives the token, it immediately throws it to the next member, like a hot potato. This policy aims at decreasing latency when sending messages: Since with this protocol a member can only send a message when it has the token, the faster the token rotates around the group, the shorter will be the average delay before a pending message can be actually sent. However, the continuous rotation of the token adds an additional load on machines and the network and increases the time spent in `sendto` system calls, which in the end results in decreased throughput. Adding an additional layer (global-safety layer in our case) introduces an additional delay before forwarding the token, which positively affects throughput. Of course, the same effect could be achieved by the rotating token layer itself, by introducing an artificial delay (say, 1ms) before forwarding the token to the next group member. This would also increase throughput at the cost of increased latency.

The aggregate throughput numbers (the number of multicasts sent by all group members in a second) indicate the total capacity of the system, and in many settings, this is the important figure. For example, consider a bank in which ATM machines are connected to a distributed servers system, such that each ATM is connected to one server. In this case, the aggregate throughput indicates the total number of client's requests that the system can handle. The aggregate throughput achieved in our tests appears to be quite stable as the group size is increasing. In particular, with the SAFE:TOKEN:PVSYNC stack, a group of five members can sustain 10,550 authoritative (globally ordered and safe) multicasts per second. Recall that each authoritative message can be thought of as a single object transaction. (In the example of a banking system, this can be a deposit, a withdrawal, or an account status inquiry.) Also, with optimistic delivery, our system maintains a throughput of over 14,000 messages per second even with five members.

We did not use IP multicast for performance tests reported in this section (group multicasts are simulated with unicasts). We expect to see still better numbers when measuring performance with IPMC.

7 An Application Interface for Replicated Objects

In this section we describe an interface for distributed objects [26] that brings the replicated state machine functionality to the application layer. The interface is a part of Horus Object Tools which have been implemented over the Horus system at Cornell [25]. We discuss the functionality of our interface and give a usage example based on a simple replicated-value object application.

Replicated objects are implemented with the `ObjectReplica` class shown in Figure 7. Using standard terminology, we say that object replicas are members of the corresponding *object group*.[5] External clients can access a replicated-object group through the `Client` class defined in Figure 8.

7.1 Object Group Membership

An object replica can *join* its group with the `join` downcall method. Parameters to `join` specify the maximum group size and whether communication in the group is to be *optimistic* or *globally safe*. The latter option determines whether the SAFE layer is going to be included in the underlying protocol stack. An object can *leave* its group with the `leave` downcall method. An object is notified of membership changes in the group with `view_Callback` methods invoked by Horus. By default `view_Callback` is defined as a no-op function (to be overloaded in subclasses of `ObjectReplica`). The view callback specifies the current membership in the group (as a list of object ID's) and whether the view is *primary*. An object is allowed to initiate update operations that can potentially modify the replicated state of the group only if the current view is primary. Recall that the *current* view of an object is the last group view delivered to it with an invocation of the `view_Callback` method.

[5] Inessential details of C++ syntax and the class definition are left out.

```
class ObjectReplica {
    // Object group membership
    join(Bool optimistic, int maxGroupSize);
    leave();
    view_Callback(ObjectIDList members, Bool primary);

    // Communication within the object group
    multicast(Message msg);
    multicast_Callback(ObjectID origin, Message msg, Bool safe);
    safe_Callback(ObjectID origin, int nsafe);

    // Interaction with clients of the object group
    replyToClient(ClientID client, Message reply);
    clientRequest_Callback(ClientID client, Message request);

    // State transfer
    stateTransfer_Callback(XferID id);
    stateTransferDone(XferID id);
    requestState(XferID id, Message request);
    stateRequest_Callback(ObjectID origin, XferID id, Message request);
    sendState(ObjectID dest, XferID id, Message state);
    rcvState_Callback(ObjectID origin, XferID id, Message state);
    stateTransferTerminated_Callback(XferID id);
}
```

Figure 7: Implementation of Replicated Objects with the `ObjectReplica` class

7.2 Communication within an Object Group

Communication within a group is performed via multicast messages. The `multicast` downcall method is used to send a message. When an object receives a message, the `multicast_Callback` method is invoked by Horus. By default, `multicast_Callback` is defined as a no-op function, however it can be overloaded in a subclass of `ObjectReplica` to implement application-specific functionality. Parameters to `multicast_Callback` specify whether the message is *globally safe*. If the group has been configured for the globally-safe execution mode, messages will *always* be delivered only when they become safe. Otherwise the `safe_Callback` method will be used to notify objects when delivered messages become globally safe. Parameters to `safe_Callback` specify how many messages received from the given member of the object group (in the current view) have become authoritative and therefore can be acted upon.

7.3 Communication between Clients and Replicated Objects

Clients can access an object group with the `sendRequest` method. Following a call to `sendRequest`, the `clientRequest_Callback` method is invoked at one of the object replicas. The replica that received the request may send a reply to the client with the `replyToClient` downcall method. In some applications/execution modes it may be necessary to relay a client request to all object

```
class Client {
    // Communication with the object group
    sendRequest(Message request);
    rcvReply_Callback(ObjectID origin, Message reply);
}
```

Figure 8: Implementation of clients with the `Client` class

replicas (which can be done with the `multicast` method). After an object makes a call to `replyToClient`, the `rcvReply_Callback` method will be invoked at the corresponding client. If the application is running in the optimistic mode, the client may need to collect replies from a majority of object replicas before the reply becomes authoritative.

7.4 State Transfer

Horus invokes the `stateTransfer_Callback` method in order to notify objects that a *state transfer* is being (re)started; the callback specifies the ID of the current state transfer transaction. An object replica can request the state (or a portion thereof) with the `requestState` method. Following a call to `requestState`, the `stateRequest_Callback` method is invoked at an object replica with the up-to-date state. An object can send the state (or a portion of it) with the `sendState` method. A call to `sendState` results in an invocation of the `rcvState_Callback` method at the destination object. Parameters to `rcvState_Callback` include the reply message which can be used to bring the state of the object replica up-to-date. After an object completes state transfer, it invokes the `stateTransferDone` method to notify Horus. If state transfer has to be terminated (due to network partitions or process crashes), the object is notified with the `stateTransferTerminated_Callback` method invoked by Horus.

7.5 Example: Simple Replicated-Data Application

In this section we demonstrate how to apply our interfaces in a simple application using replicated data. Each object maintains a replica of the global state contained in the integer *state*. Clients can request read and update operations to be performed on *state*. We assume that the application in our example will be running in the *globally safe* mode. Therefore, when an object receives a read request from a client, it can reply immediately with the local value of *state*, and the value in the reply is guaranteed to be a valid state of the group. When an object receives an update request from a client, it relays it to other replicas. When an object replica receives a relayed update request, it modifies its local copy of *state* correspondingly. Implementation of object replicas is shown in Figure 9. The client code is shown in Figure 10[6].

[6] Inessential details of C++ syntax and class definitions are omitted.

```
typedef enum { READ, UPDATE } request_type;

class MyObjectReplica:  ObjectReplica {
    // Initialize the state.
    MyObjectReplica() {
        state = 0;
    }
    // Got a relayed update request. Modify the state accordingly.
    multicast_Callback(ObjectID origin, Message msg, Bool safe) {
        msg >> state;
    }
    // Got a client request.
    clientRequest_Callback(ClientID client, Message request) {
        int requestType;
        request >> requestType;
        switch (requestType) {
        case READ:
            // Send reply to the client.
            Message reply;
            reply << state;
            replyToClient(client, reply);
            break;
        case UPDATE:
            // Relay request to all replicas.
            multicast(request);
            break;
        }
    }
    // Start state transfer.
    stateTransfer_Callback(XferID id) {
        Message request;
        requestState(id, request);
    }
    // Got a state request.
    stateRequest_Callback(ObjectID origin, XferID id, Message request) {
        Message stateMsg;
        stateMsg << state;
        sendState(origin, id, stateMsg);
    }
    // Received the state.
    rcvState_Callback(ObjectID origin, XferID id, Message stateMsg) {
        stateMsg >> state;
        stateTransferDone(id);
    }
    int state;
}
```

Figure 9: Example of a Simple Replicated-Data Application – Implementation of Replicated Objects

8 Discussion

We have developed a modular implementation of replicated distributed objects based on the replicated state machine paradigm that can operate over partitionable networks. Our imple-

```
class MyClient: Client {
    // Send a read request. Block until reply arrives. (Not reentrant).
    int read() {
        Message request;
        request << READ;
        sendRequest(request);
        cv.wait();
        return state;
    }
    // Send an update request
    update(int newState) {
        Message request;
        request << UPDATE << newState;
        sendRequest(request);
    }
    // Received a reply to a read request
    rcvReply_Callback(ObjectID origin, Message reply) {
        reply >> state;
        cv.signal();
    }
    ConditionVariable cv;
    int state;
}
```

Figure 10: Example of a Simple Replicated-Data Application – Implementation of Clients

mentation has three appealing properties: First, it allows minority partitions to install views and continue to provide idempotent services. Second, even if the system's ability to make progress is suspended for a while due to loss of connectivity, the system will once again be able to make progress, without any outside intervention, as soon as a majority of members become connected again. Third, it offers the application the option of using both optimistic and safe delivery. In the optimistic case, messages are delivered with one phase communication, and is therefore very fast. However, even in the case of pessimistic (safe) delivery, we have achieved better performance than other implementations known to us, especially in terms of throughput. Our solution also deals explicitly with maintaining global consistency. In particular, it addresses the issues of doing state transfer in a way that maintains the semantics of replicated state machines, and does safe recovery of processes after crashes.

Our work assumes that the number of members in the group is fixed. Such an assumption is also made by the Phoenix system [19], and is quite reasonable for most practical applications. This assumption is reasonable since the replicated state machine model is usually used to implement reliable/highly-available servers, whose number is normally fixed and known in advance. It is possible, however, to slightly modify our protocol to allow dynamic changes to the group size, under certain assumptions. The details of this solution will be described in a follow-up paper. Another approach would be to follow the ISIS implementation [7], which also allows dynamic group sizes. With the ISIS approach, a view would need to have a majority of members of the previous primary view, rather than a majority of all group members, in order to be installed as primary. The drawback of this solution, however, is that a single failure could cause the system to block forever, *e.g.*, if a member of a primary view with only two members crashes.

We have also presented a programming interface supporting replicated distributed object functionality at the application level. The interface provides integrated support for group communication, state transfer, and primary views and can be exploited in distributed CSCW or client/server applications using either the globally-safe or the optimistic execution models.

The protocols for primary views and global safety and the interface to replicated distributed objects described in this paper have been implemented as part of the Horus group communication system. To get more information about Horus, or how to obtain it, you may visit the Horus home page at
`http://www.cs.cornell.edu/Info/Projects/Horus`.

Acknowledgments: We would like to thank Ken Birman and Robbert van Renesse for many helpful comments and discussions.

References

[1] Y. Amir. *Replication Using Group Communication Over a Partitioned Network.* PhD thesis, Institute of Computer Science, the Hebrew University of Jerusalem, 1995.

[2] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and Efficient Replication Using Group Communication. Technical Report CS94-20, Institute of Computer Science, the Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[3] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. In *Proc. of the 13th International Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[4] Ö. Babaoğlu, R. Davoli, L. Giachini, and M. Baker. Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems. Technical Report UBLCS–94–15, Department of Computer Science, University of Bologna, June 1994. Revised January 1995.

[5] Ö. Babaoğlu, R. Davoli, L. Giachini, and P. Sabattini. The Inherent Cost of Strong-Partial View-Synchronous Communication. Technical Report UBLCS–95–11, Department of Computer Science, University of Bologna, April 1995.

[6] P. Bernstein, V. Hadzilacos, and H. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[7] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[8] T. Bressoud and F. Schneider. Hypervisor-base Fault Tolerance. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 1–11, December 1995.

[9] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the Impossibility of Group Membership. In *Proc. of the 15th ACM Symposium of Principles of Distributed Computing*, pages 322–330, May 1996.

[10] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[11] D. Dolev and D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[12] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. In *Proc. of the 15th Symposium on Reliable Distributed Systems*, pages 140–149, October 1996.

[13] R. Friedman and R. van Renesse. Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. In *Proc. of the Sixth IEEE International Symposium on High Performance Distributed Computing*, 1997. To appear.

[14] N. C. Hutchinson and L. L. Peterson. The x-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[15] F. Kaashoek, A. Tanenbaum, S. Hummel, and H. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[16] I. Keidar. A Highly Available Paradigm for Consistent Object Replication. Master's thesis, Institute of Computer Science, the Hebrew University of Jerusalem, 1994.

[17] I. Keidar and D. Dolev. Increasing the Resilience of Atomic Commit, at No Additional Cost. In *Proc. of ACM Symposium on Principles of Database Systems*, pages 245–254, May 1995.

[18] L. Lamport. Time, Clocks and the Ordering of Event in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[19] C. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A Tollkit for Building Fault-Tolerant Distributed Application in Large Scale. Technical report, Department d'Informatique, Ecole Polytechnique Federale de Lausanne, July 1995.

[20] S. Mishra, L. Peterson, and R. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. *Distributed Systems Engineering Journal*, 1(2):87–103, December 1993.

[21] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended Virtual Synchrony. In *Proc. of the 14 International Conference on distributed Computing Systems*, June 1994.

[22] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Asynchronous Fault-Tolerant Total Ordering Algorithm. *SIAM Journal of Computing*, 22(4):727–750, August 1993.

[23] F. Schneider. Paradigms for Distributed Programs. In *Distributed Systems – Methods and Tools for Specification*, pages 343–430. Lecture Notes in Computer Science, Vol. 190, Springer-Verlag, New-York, NY, 1985.

[24] Fred B. Schneider. The state machine approach: a tutorial. Technical Report TR 86-800, Department of Computer Science, Cornell University, December 1986. Revised June 1987.

[25] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[26] Alexey Vaysburd. The Horus Object Tools. http://www.cs.cornell.edu/Info/Projects/HORUS/HOT/hot.html.

A Pseudocode of the Primary View Protocol

The pseudocode we give in this section assumes the existence of a total ordering layer and a partitionable virtual synchrony layer, as discussed in Section 3. These layers are responsible for installing views which obey the properties of partitionable virtual synchrony, and for delivering messages in total ordering. Thus, the code we present here only deals with the problem of merging partitions in a consistent way, providing hooks for the application for doing state transfer, and declaring when a view is primary.

We assume that the interface between the primary views layer and the lower layers include the following downcalls and upcalls:

- propose **view** (*new_view*) – this downcall requests the partitionable virtual synchrony layer to install a new view with parameters specified in the *new_view* argument.
- broadcast (**msg-type**, *msg*) – this downcall requests the lower layers to broadcast a message *msg* of type **msg-type** to all view members.
- send [*to_member*] (**msg-type**, *msg*) – his downcall requests the lower layers to send a message *msg* of type **msg-type** to the group member specified in the *to_member* argument.
- received `msg-type` (*arg_list*) – this upcall delivers to the primary views layer that a message of type `msg-type` with arguments (*arg_list*). The types of messages we are using are `I-am-alive`, `merge_request`, `cast`, and `xfer_done`.
- received `view` (*new_view*) – this upcall delivers to the primary views layer a new view *new_view*.

while the interface between the application and the primary views layer include:

- send `msg-type` (*msg*) – this downcall is used by the application to request that a message *msg* of type `msg-type` be broadcast to the sender's current view. The types of messages we use are `cast` (for messages containing application data), and `xfer-done` (for messages that are sent to notify view members of the completion of a state transfer).
- deliver (**view**, *new_view*) – an upcall to deliver a new view (specified in the *new_view* argument) to the application.
- deliver (**msg-type**, *msg*) – an upcall to deliver a message *msg* of type **msg-type** to the application.

(In practice, the above interfaces may include more downcalls and upcalls, but here we only list those that are used by our code.) Each process maintains the following data structures used by the primary-views protocol:

- *local_endpoint* is a record that contains the globally unique *ID* of the local process, its *incarnation*, the current *state version*, and the *zombie* flag.

  The *incarnation* number specifies how many times the process has been (re)started. Whenever a process is brought back after a crash, its incarnation number increases by one.

  The *state version* number is used to keep track of the process' progress, as described earlier. When a process is initialized, its current *primary-view sequence number* and *message sequence number* are both set to 0, which corresponds to the initial state.

  The *zombie* flag is initially set if the process is restarting after a crash. As discussed earlier, zombie processes do not count when determining whether a view has a majority of group members.

- *group* is a record that contains the current *state* of the process, the *total number of members* (which is fixed for the entire execution of the group), and the *i_am_contact* flag.

  The *state* is set to *merging* when the process is participating in an ongoing view merge; otherwise it is set to *normal*. A process starts in a *normal* state.

  The *i_am_contact* flag is set if the process is responsible for conducting merges on behalf of its view. Initially, the process is the contact of its singleton view.

- *local_view* is a record that contains the current *state version* of the local process, the current list of *members* (the group view), and the *primary* flag.

  The list of *members* initially contains the local endpoint only. The *primary* flag is set if the current view is primary. It will be initially set if the process is not a zombie and it is the only member in the group.

The initialization procedure is shown in full detail in Figure 11. The pseudocode description is given is Figures 12 and 13.[7]

B Pseudocode of the Global Safety Protocol

As discussed before, the Global Safety layer is placed on top of the primary views layer, and delivers messages only after they were acknowledged by a majority of group members. In particular, during view changes it propagates messages that were not seen by all members of the group to the new members.

The pseudocode description of this protocol is divided into a procedural part that appears in Figure 15, and an event driven part in Figure 15. Each process holds the following variables:

- *my_id* holding the id of the local process.

[7] The actual implementation includes a few optimizations on this code, which were removed from this description for the sake of clarity of presentation.

```
initialize:
    if (local_endpoint.incarnation = 1) then
        local_endpoint.zombie := false
    else
        local_endpoint.zombie := true
    endif

    if (not local_endpoint.zombie and group.total_nmembers = 1) then
        local_view.primary := true
    else
        local_view.primary := false
    endif

    local_view.state_version.prim_view_seqno := 0
    local_view.state_version.msg_seqno := 0
    local_endpoint.state_version := local_view.state_version
    local_view.members := {local_endpoint}

    deliver (view, local_view)
    group.state := normal
    group.i_am_contact := true
```

Figure 11: The Primary Views layer: Process Initialization

- *local_view* holding the view of the local process.
- *newunacked* is a boolean variable that indicates whether the process received any messages and has not acknowledged them. It allows a single ack messages to acknowledge several broadcast messages, and therefore reduces the number of messages generated by the protocol.
- *non-auth-seqno* is an integer that counts the sequence numbers of unauthoritative messages received by the process, while *delivered* is an integer counter for the messages that were delivered to the process.
- *non-authoritative* holds the set of non-authoritative messages received by the process. These messages are kept so they can be sent to processes that have not received them, when these processes will eventually reconnect with the primary view. In order to detect which processes have acknowledged each message, each element of these sets includes the following fields: *msg* - the message itself, *seqno* - the value of *non-auth-seqno* that corresponds to this messages, and *acked* - the list of processes that acknowledged this messages.

Most of the code is obvious from the discussion in Section 4. Upon receiving a message in a primary partition mode, a process buffers this message in the *non-authoritative* list, and then

```
every Δ1 time units:
    if (group.i_am_contact and group.state = normal) then
        broadcast (I-am-alive, local_endpoint, local_view.state_version)
    endif

received I-am-alive (contact, state_version):
    if (group.state = normal and local_view.state_version < state_version) then
        send [contact] (merge_request, local_view);
        group.state := merging
    endif

received merge_request (merging_view):
    if (group.state = normal and
        merging_view.state_version < local_view.state_version)
    then
        new_view.members := local_view.members ⋃ merging_view.members

        if (#{m ∈ new_view.members | m.zombie = false} > group.total_nmembers / 2 and
            local_view.state_version = merging_view.state_version)
        then
            new_view.primary := true
        else
            new_view.primary := false
        endif

        if (new_view.primary) then
            for all m ∈ new_view.members do
                m.zombie := false
            done
            new_view.state_version.prim_view_seqno := local_view.state_version.prim_view_seqno + 1
            new_view.state_version.msg_seqno := 0
        else
            new_view.state_version := local_view.state_version
        endif

        propose view (new_view);
        group.state := merging
    endif
```

Figure 12: The Primary Views layer implementation

needs to acknowledge the receipt of the message. However, to save on the number of messages, the acknowledgment is not sent immediately. Instead, the receiver increments a counter *non-auth-seqno* for each message received, and then periodically sends an acknowledgment that contains the current value of *non-auth-seqno*.

Whenever a process receives an acknowledgment with counter *seqno*, it checks which messages

```
received view (new_view):
    local_view := new_view
    if (rank of local_endpoint in local_view.members is 0) then
        group.i_am_contact := true
    else
        group.i_am_contact := false
    endif
    if (local_view.primary or
        #{m ∈ local_view.members | m.zombie = false} ≤ group.total_nmembers / 2)
    then
        group.state := normal
    endif
    deliver (view, local_view)

send cast (msg):
    broadcast (cast, msg)

received cast (msg):
    if (local_view.primary) then
        local_view.state_version.msg_seqno := local_view.state_version.msg_seqno + 1
    endif
    deliver (cast, msg)

send xfer_done (mbr):
    broadcast (xfer_done, mbr)

received xfer_done (mbr):
    local_view.members[mbr].state_version := local_view.state_version
    if ((∀ m ∈ local_view.members:
        m.state_version = local_view.state_version) and
            (#{m ∈ local_view.members | m.zombie = false} > group.total_nmembers / 2) and
            not local_view.primary and group.i_am_contact)
    then
        new_view.members := local_view.members
        new_view.primary := true
        for all m ∈ new_view.members do
            m.zombie := false
        done
        new_view.state_version.prim_view_seqno :=
            local_view.state_version.prim_view_seqno + 1;
        new_view.state_version.msg_seqno := 0
        propose view (new_view)
    endif
```

Figure 13: The Primary Views layer implementation (continued)

that have smaller sequence number in *non-authoritative* are now acknowledged by a majority. These messages are then delivered to the application in an authoritative mode, and are removed

```
initialize:
    newunacked := false
    no-auth-seqno := 0
    delivered := 0
    non-authoritative := ∅
    authoritative := ∅

message_acked:
    foreach m in non-authoritative that was acked by a majority do
        if m.seqno > delivered then
            deliver (auth-cast, m.msg)
            delivered := delivered + 1
        endif
        remove m from non-authoritative
        add m to authoritative
    done
    foreach m in authoritative that was acked by all members do
        remove m from authoritative
    done
```

Figure 14: Procedural code for The Global Safety layer

from the *non-authoritative* list.

In order to guarantee correct state transfer, the *non-authoritative* and *non-auth-seqno* list has to be sent to the new members of each primary view. Since the code in Appendix A guarantees that the first member in each view is a member of the most advanced view, this member sends his data to all new members. New members that receive the `xfer` message adopt it into their state. In particular, they acknowledge all messages in the newly adopted *non-auth-seqno* list, and mark themselves as processes that acknowledged them. Each of these messages that has been majority acknowledged is then delivered locally as authoritative.

Messages that are delivered but were not acknowledged by all members are buffered in the *authoritative* list until all acknowledgements for them are received. In our real implementation, this option can be turned off, at which point the application is responsible for that part of the state transfer. The rational behind this is that the application can usually do more efficient state transfer, since it knows the contents of messages. However, for the sake of completeness we included in the code presented here an explicit handling of authoritative but unstable messages.

```
every Δ₂ time units:
    if newunacked then
        newunacked := false
        broadcast (ack, seqno)
    endif

received view (new_view):
    local_view := new_view
    if (not new_view.primary) and I appear first in the view then
        send (xfer, non-authoritative, authoritative, non-auth-seqno) to all new members
    endif
    deliver (view, new_view)

received xfer (non-auth, auth, non-seqno):
    newunacked := true
    non-authoritative := non-auth
    authoritative := auth
    no-auth-seqno := non-seqno
    foreach m in non-authoritative do
        m.acked := m.acked ∪ {my_id}
    done
    call message_acked

send cast (msg):
    broadcast (cast, msg)

received cast (msg):
    if local_view.primary then
        newunacked := true
        non-auth-seqno := non-auth-seqno + 1
        m.msg := msg
        m.seqno := non-auth-seqno
        m.acked := {my_id}
        add m to non-authoritative
    else
        deliver (non-auth-cast, msg)
    endif

received ack (seqno) from j:
    foreach m in non-authoritative for which m.seqno ≤ seqno do
        m.acked := m.acked ∪ {j}
    done
    call message_acked

send xfer_done (mbr):
    broadcast (xfer_done, mbr).
```

Figure 15: Event driven code for the Global Safety layer