Implementing Replicated State Machines Over Partitionable Networks*

Roy Friedman   Alexey Vaysburd
Department of Computer Science
Cornell University

{roy,alexey}@cs.cornell.edu

April 5, 1996

Abstract

This paper presents an implementation of a replicated state machine in asynchronous distributed environments prone to node failures and network partitions. This implementation has two appealing properties: It allows minority partitions to continue providing service for idempotent requests, and it guarantees that progress will be made whenever a majority of replicas can communicate with each other.

*This research was supported by by ARPA/ONR grant N00014-92-J-1866

1 Introduction

Process 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 [8, 13] as a way to provide such a consistent behavior: With this approach, all replicas run identical state machines, and all communication in the system is performed via totally ordered broadcasts. Since all replicas start in the same state, and deliver exactly the same set of messages in the same order, they proceed through the same sequence of states. This property allows to ensure that actions taken by the system appear to the outside world as if they were performed by a single processor.

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.*, [3, 9], tend to deliver messages fast, yielding good performance, although individual replicas may need to roll their state back under certain failure scenarios. On the other hand, pessimistic implementations, *e.g.*, [1, 2, 6], deliver messages only when it is safe to do so, thereby avoiding the risk of having to roll back, although this is achieved at the cost of additional communication rounds.

Existing implementations of replicated state machines either ignore the issue of network partitions [4, 12], or have a limited support for partitions but can block forever even after all partitions have been reconciled [2, 3], or are pessimistic [1, 6, 10]. 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 a replicated state machine that can tolerate network partitions and is optimistic and thus efficient. In 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 do *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 updated state.

Note that as indicated in [5], 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 does 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. Also, the implementation we present uses one phase communication for executing an operation, which yields good performance.

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

Finally, our solution is implemented in Horus, and therefore our discussion sometimes deals with specific details of the Horus implementation. However, the proposed approach can easily be applied to any system that supports partitioned operation, including Transis and Relacs.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 presents the 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. Finally, we conclude with a discussion in Section 5.

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 [12]. In a recent paper, Bressoud and Schneider presented a hardware-based implementation of a replicated state machine using off-the-shelf workstations, called Hypervisor [4]. This solution is workable only in tightly coupled systems, and is definitely not suitable for wide-area distributed environments. On the other hand, 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 [3] provides the `abcast` primitive and the notion of a *primary partition* as abstractions for implementing a replicated state machine semantics in asynchronous distributed environments. In Isis, processes that are removed from the primary partition are forced to terminate. Hence, network partitions may cause Isis to lose the primary partition forever, even if these partitions later remerge. Also, by killing processes that are removed from the primary partition, Isis does not allow minority partitions to even execute idempotent operations, even though many applications could benefit from such a feature. Note that Isis also offers primitives for causal and unordered communication, which can be used for improved performance, although if not used with care, they can violate the replicated state machine abstraction.

Phoenix [9] 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, papers describing Phoenix do not sufficiently 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 make sure it is performed in a meaningful way.

Pessimistic implementations of global total ordering that can tolerate network partitions have been proposed in [1, 6, 10]. However, being pessimistic, they require more communication rounds to perform an operation.[2] Also, these solutions require all messages to be logged on a stable storage, which adds a substantial overhead that may be unacceptable for many applications. On the contrary, our implementation only requires logging of a single bit, which is performed by every process only once at the initialization time.

Finally, 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 [2, 7].

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* [8], 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 [14]. It is designed to reside on top of a system which provides *totally ordered broadcast* in a *strongly virtually synchronous* environment, 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 strong virtual synchrony layer. However, this separation is not necessary for our implementation.

3.3 Strong Virtual Synchrony

In the strong virtual synchrony model, each process has a view of the group (list of current members) at any time. Strong virtual synchrony supports a partitionable group model that

[2] Note that [1] claims to deliver messages as soon as they arrive from the transport layer, without end-to-end acknowledgements, but it assumes that the transport layer provides total safe (uniform) delivery, which requires at least 2 communication rounds at the transport level.

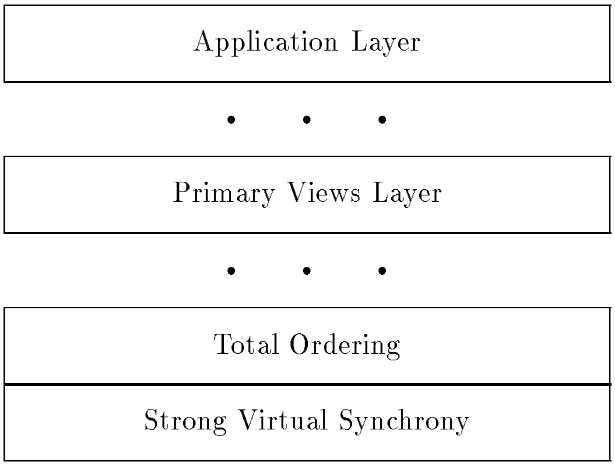

Figure 1: The Layered Protocol Architecture of Horus

allows multiple concurrent views to be installed at different processes. A view is 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 strong virtual synchrony layer guarantees the following properties regarding installation of new views:

Property 3.1 (Validity) *If a process accepts a view, then this view was proposed by some process.*

Property 3.2 (Self-Inclusion) *If a process proposes or accepts a view, it is included in it.*

Property 3.3 (View Causality) *View messages are delivered in causal order.*

Property 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 3.5 (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 strong 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 3.6 (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.*

Property 3.7 (Same-View Delivery) *If a message is delivered, it is delivered in the same view in which it was sent.*

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. Together with view atomicity (Property 3.6), total ordering ensures that processes in a view deliver the same *sequences* of messages, as captured by the following property:

Property 3.8 (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 vice versa.*

In a system with a partitionable group model, such as the one provided by strong virtual synchrony, there are scenarios in which a message may need to be dropped to prevent a violation of the strong prefix property (Property 3.8). This problem does not arise in Isis, where potential violators are forced to terminate.

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

Property 3.9 (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 3.9 provides the replicated state machine semantics for processes in a view between two consecutive view changes. However, we need to provide a replicated state machine semantics for the entire execution of the system, not merely within one view. This task is complicated since strong virtual synchrony allows multiple views to be executing concurrently. 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. In other words, processes in different views may be exchanging different sets of messages and producing different outputs. Although each particular view may satisfy Property 3.9, 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

Recall that a replicated-state-machine execution, as represented by a group state history, should be indistinguishable from an execution of a single process with an equivalent message history. In the rest of this section, we will discuss the properties of primary views and their relation to the replicated state machine semantics. We will also describe our implementation of primary views in Horus.

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 nonempty) intersection. By Property 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. This is called *state transfer*. In particular, we assume that if processes from the most recent primary view V merge with processes from another view, then processes that were not in V adopt the state and message history of processes in V.

We say that a process *accepts* a message either if it actually delivers it, 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 never be rolled back.

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 (implicitly) 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.

Note that due to Property 3.8, and our assumption regarding the way state transfers are done, 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 3.9. 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 this 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 limitation 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 now to formulate the following property:

- **Replicated State Machine Property:** Suppose $M = (m_0, \ldots, m_n)$ is the *group message history* and $S = (S_0, \ldots, S_n)$ 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 goal of our protocols is to merge concurrent views (partitions), to form larger views, notify a primary view that it is primary, and do state transfer such that the initial state of every newly formed primary view will be 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. 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.

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 2: The Primary Views layer: Process Initialization

When the contact of a certain view V receives an `I-am-alive` message, it replies by sending a `merge_request` to the contact p 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 p receives this `merge_request` message, then if p's view is not already merging with another view, and if the state version number of q is smaller than the state version number of p, then a new view U, which is a union of both V and W, can be installed.

In order to install U, p proposes U to all members of U, including itself, by sending a `view` message to them, which specify the last authoritative state of V. When proposed members of U accept it, they install this view by delivering a **view** event to the application. At this point, they may need to do state transfer in order to reconcile their state with the state of more advanced members. After completing state transfer, a member sends an `xfer_done` message to p. When p receives `xfer_done` messages from all processes that need to do state transfer, it reinstalls U as a primary view.

An additional 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, in which there are 3 processes in the system, p, q, and r, such that p and q form a primary view at some point. 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. 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. This allows the process to determine at 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 will 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.

4.2.1 Pseudocode

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 2. The pseudocode description is given is Figures 3 and 4.[3]

4.3 The Client's Point of View

Our implementation guarantees that all members of the primary view will go through the same set of state changes in the same order and will accept the same set of messages in the same order which obeys the definition of a replicated state machine. However, one of the purposes of the replicated state machine approach is to present an outside observer (client) of the system with an illusion of a single process' execution. The interaction with outside clients is performed through group outputs, as discussed in Section 3.1. However, an output produced by a single process may have to be rolled back. In order to ensure that an action performed by a group member will not be rescinded later, an external observer should gather (presumably equivalent) outputs from a majority of the members, before it accepts that output.

An output produced by a majority of processes is called *authoritative*. If an external observer considers authoritative outputs only, the behavior of the group will be indistinguishable from an execution of a single process with an equivalent message history, with the following exception: An output triggered by the delivery of an authoritative message may never be reproduced by a *majority* of group members, and thus this output may never become authoritative. This can happen if the message has never been delivered by a majority of processes and only became authoritative as a result of state transfers. Thus, the sequence of authoritative outputs produced through an execution of the group is a (possibly proper) subsequence of

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

```
broadcast I-am-alive:

    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 [to 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 3: The Primary Views layer implementation

```
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 4: The Primary Views layer implementation (continued)

outputs produced by a single process with the identical message history. In practice, an application may choose not to wait for an output to be reproduced by a majority of members, in which case it must be able to rescind the consequences of accepting such an output, if required. Alternatively, a client can be designed to sustain a possible loss of an output generated by an authoritative message.

With the authoritative output approach, the cost of maintaining external consistency is shifted to outside observers (the clients of the group), which are required to wait for an output from a majority before using this output. Another way to solve the external consistency problem is to assume *uniform delivery* of messages with respect to the primary view [11]. That is, if a member of a primary view delivers a message, then all members of the view that also appear in the next primary view deliver the message as well. (There is a layer in Horus that provides uniform delivery.) In this case, an authoritative action can simply be defined as an action taken by any member of the primary view after it delivered the corresponding message in totally uniform order.

Thus, uniform delivery guarantees that if a member of a primary view delivers a message, then this message is included in the message history of the primary view. Therefore, an action triggered by a delivery of such a message will not be rolled back. This allows an external observer to only wait for an output from one member, rather than wait for a majority. From a practical point of view, the cost of this solution is unacceptable for many applications, since uniform delivery requires running an agreement protocol for each message.

5 Discussion

We have developed an optimistic implementation of a replicated state machine that can operate over partitionable networks. Our implementation 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, messages are delivered within one phase communication. Our solution also deals explicitly with maintaining the state of the primary view consistent. In particular, it addresses the issues of doing state transfer in a way that maintains the semantics of a replicated state machine, and doing 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 [9], 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 describe in a follow-up paper. Another approach would be to follow the Isis implementation [3], 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.

The implementation of primary views discussed in this paper is a part of the Horus system. To get more information about Horus, you may visit the Horus home page at `http://www.cs.cornell.edu/Info/Projects/Horus`.

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

References

[1] 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.

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

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

[4] T. Bressoud and F. Schneider. Hypervisor-base Fault Tolerance. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 1–11, December 1996.

[5] 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*, 1996. To appear.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] G. Neiger and S. Toueg. Automatically Increasing the Fault-Tolerance of Distributed Algorithms. *Journal of Algorithms*, 11(3):374–419, September 1990.

[12] 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.

[13] 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.

[14] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A Framework for Protocol Composition in Horus. In *Proc. of the 14th ACM Symposium on Principles of Distributed Computing*, pages 80–89, August 1995.