Trading Consistency for Availability in Distributed Systems*

Ken Birman Roy Friedman

Department of Computer Science
Cornell University
Ithaca, NY 14853.

April 8, 1996

Abstract

This paper shows that two important classes of actions, *non left commuting* and *strongly non commuting*, cannot be executed by concurrent partitions in a system that provides serializable services. This result indicates that there is an inherent limitation to the ability of systems to provide services in a consistent manner during network partitions.

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

1 Introduction

Being able to continue to provide services despite failures is one of the main advantages that distributed environments claim over centralized ones. It is often assumed that distributed environments can be designed in such a way that if one component of the system fails, or becomes disconnected (partitioned), other components can take over, and hide the effects of this failures from the outside world. Systems that exhibit this kind of behavior are called *highly available*. One of the main questions in the design of distributed systems is how many failures can a given system sustain before the effects of these failures would be noticed.

Group communication is a research area where many aspects of consistency and availability have been investigated, mostly using the virtual synchrony model [8]. One of the main debates that evolved in the research of virtually synchronous systems is regarding their availability during network partitions. The question at issue concerns the conditions under which different partitions of the systems can continue to provide services concurrently, while maintaining their semantics. On one hand, according to the ISIS model [8], only processes that appear in the primary partition are allowed to make progress, while all other processes must wait until they are reconnected with the primary partition. On the other hand, more recent models like *extended virtual synchrony* [2], *strong partial view synchrony* [6], and *strong virtual synchrony* [12], allow different partitions to make progress in parallel, in order to achieve a higher degree of availability.

The motivation behind the original ISIS approach was an intuitive belief that in order for a system to exhibit a consistent behavior to the outside world, it must operate in a coherent way, which cannot be done if different partitions are allowed to act on their own. In this paper we formalize this intuition by identifying two important classes of actions, i.e., *non left commuting* and *strongly non commuting*, which cannot be performed by more than one partition in parallel. This result formally proves that if a system allows different partitions to take arbitrary actions in parallel, then it may exhibit inconsistent behavior to the outside world.[1]

The model used in this paper assumes that the only possible failures are message losses. This model is therefore simpler than models that allow other forms of failures, but the impossibility result holds in these more complex models as well. Finally, note that although this work was inspired by a debate in the virtually synchronous systems research community, our impossibility result is general and applies to other systems as well.

[1] Our impossibility result does not indicate that in order to maintain consistency, one has to use an ISIS like primary partition solution. In particular, Keidar has shown a solution in which all actions can be serviced even if a primary partition is never installed [15]. However, in [15], processes are not allowed to perform an action before they know that at least half of the processes in the system were notified about this action. Thus, minority partitions must interact before they can perform actions.

1.1 Related Work

The most prominent impossibility result regarding availability in distributed systems is the lower bound of Fischer, Lynch, and Paterson [10], stating that a distributed system cannot reach a consensus among its members even if it is known that at most one process may fail during the computation. Our result is different from the Fischer, Lynch, and Paterson impossibility result [10], since we do not require that all processes will eventually make any progress. In particular, some processes may not be aware at all of some of the actions taken by the system. Also, satisfying a sequential specification of a service does not necessarily mean being able to reach a consensus [3].

Our proof method was largely influenced from lower bounds in the area of distributed shared memories, stating that certain memory operations cannot be executed faster than the network delay. The *application program* in [4, 5, 16] can be viewed as a client which has one (and only one) reliable link to a process where no process can crash and the network is reliable, but has no known bound for message delays. In that sense, our work can be viewed as a generalization of the results in the area of distributed shared memory without network partitions, to results about general services in a distributed environments prone to network partitions.

2 The Model

2.1 A Distributed System

We would like to model a distributed system that provides some kind of a service to the outside world. Examples of such systems include control applications (e.g., air traffic control and mission control) and distributed data-bases (e.g., banking systems and brokerage systems). However, no matter what type of service is provided by the system, we would like to be able to express the fact that actions taken by the system have some external effect, so once an action has been taken, there is no way to reverse it.

In our model, a distributed system consists of a set of *system processes* (or simply *processes*) and a set of *clients*. (See illustration in Figure 1.) Processes are connected by some interconnection *network*, and are allowed to communicate with each other only by sending and receiving messages through the network. However, there is no bound on the delivery time of the network and messages can get lost. If the set of processes are divided into distinct sets such that all messages sent from processes in one set to any process that is not included in their set are lost, then we say the network is *partitioned*, and each of these sets is called a *partition*. We assume that processes and clients do not crash, which makes our impossibility result even stronger.

Each client may have several *external links* to processes.[2] Clients can send *requests* to processes and receive *replies* from processes only over the external links. They are otherwise

[2] We distinguish between the internal communication channels (network) that connects processes to one

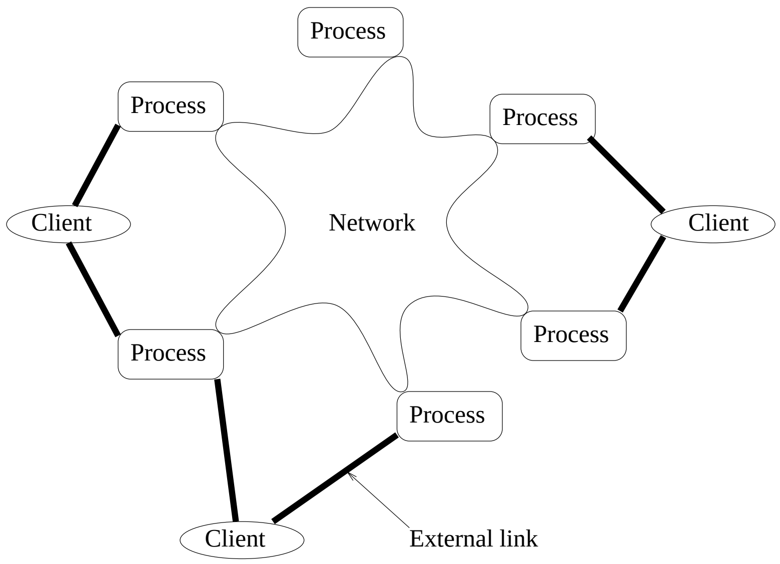


Figure 1: System illustration

not allowed to communicate with each other or with processes to which they do not have external links. We do not make any assumption about the delay of external links, but assume that they are fully reliable.

More formally, processes and clients can be viewed as (non deterministic) automata which accept zero or more events of a certain kind, do some local computation, and then generate zero or more events of another kind:

- A process can accept `message-receive` events from the network and `request-receive` events from its external links, and generate zero or more `message-send` events for the network and zero or more `reply-send` events for its external links. A `message-send` event consists of the process that generates it, the message to be sent, and the process for which the message is intended. A `message-receive` event consists of the process that receives the message, the message itself, and the process that sent the message. A `reply-send` event consists of the process that generates it, the reply to be sent, and the client to whom the reply is intended for. A `request-receive` event consists of the process that received it, the request itself, and the client that sent it.

another and the external communication links by which the distributed system interacts with the outside world. In our terminology, an external link is any device or actuator by which the system takes an externally visible action. Such links have physical visibility outside the world of the distributed system itself. The network is the medium by which processes within our system interact with one another. This distinction is an important one, and in the discussion below, the reader is cautioned to keep in mind that external links are not the same as the network.

- A client can generate zero or more `request-send` events for its external links and accept zero or more `reply-receive` events from its external links. A `request-send` event consists of the client that generates it, the request to be sent, and the process to whom the request is intended for. A `reply-receive` event consists of the client that receives it, the reply itself, and process that sent it.

A *history* of a process or a client is a list of events that occurred in that process or client, ordered in the sequence of their occurrence. An *execution* of the system is a collection of histories, one for each process and one for each client in which the following holds: (a) there is a one-to-one correspondence between `request-send` and `request-receive` events, (b) there is a one-to-one correspondence between `reply-send` and `reply-receive` events, (c) there is a mapping from `message-receive` events to `message-send` events, and (d) there is a mapping from `reply-send` events to `request-receive` events. Hence, the external links are fully reliable. On the other hand, the network may drop messages, but is not allowed to generate spurious messages, or to duplicate messages. Also, processes can only reply to requests they have received and there can be at most one reply for each request. Note that each execution implies an ordering $\xrightarrow{\sigma}$ on the events in it; an event ev appears before another event ev', denoted by $ev \xrightarrow{\sigma} ev'$, if one of the following holds:

1. both ev and ev' appear in the same history h and ev appears before ev' in h,
2. ev is a `message-send` event, ev' is a `message-receive` event, and ev' maps to ev,
3. ev is a `request-send` event, ev' is a `request-receive` event, and ev' corresponds to ev,
4. ev is a `reply-send` event, ev' is a `reply-receive` event, and ev' corresponds to ev, or
5. there exists another event ev'' such that $ev \xrightarrow{\sigma} ev'' \xrightarrow{\sigma} ev'$.

In this paper we consider only executions for which $\xrightarrow{\sigma}$ is an acyclic relation.

2.2 Services and Consistency Requirements

We assume that clients' requests are unique. A pair of matching `request-send` and `reply-receive` events which corresponds to the same request forms an *action*. The `request-send` event is called the *invocation* of the action, the `reply-receive` event is called the *point of notification* of the action, and the corresponding `reply-send` event is called the *point of decision* of the action. Note that not every `request-send` event needs to have a corresponding `reply-send` event. Hence, the term action refers only to those `request-send` events that have a matching `reply-send` event.

The processes try to implement a *service*. A service can be any abstract entity that has a *sequential specification* (cf. [14]), defining the allowed sequences of actions on this service.

Given a sequence of actions τ and a service $\mathcal{E}$, we say that τ is *legal* w.r.t. $\mathcal{E}$ if τ appears in the sequential specification of $\mathcal{E}$.

A typical request (but not the only possible request) can be a simple update request, e.g., "set a specific object to a given value", or an inquiry request, e.g., "what is the balance of a bank account". It can also be a more complex read-modify-write type of request, e.g., "sell a certain collection of stocks, assuming the account still have them", or even "transfer a certain amount of money from one account to the other". A typical reply (but not the only possible reply) can be a confirmation of whether the action actually took place, or *performed*, or if it was *aborted*, and if it was performed, a returned value. So for example, in most cases, any sequence of aborted actions, as well as any sequence of performed inquire and update actions in which every inquire returns the value of the last previous update, will be considered legal. An example of a service that is not a database arises in air traffic control systems. In this case, airplanes request directions from the air traffic controller. All sequences of such directions which do not violate safety constraints and do not cause accidents are considered legal.

Given a sequence of actions τ, we denote by $\tau \mid i$ the restriction of τ to actions invoked by client c_i. Given an execution σ and a client c_i, we denote by $acts(c_i)$ the sequence of actions that were invoked by c_i ordered in the order in which their invocations appear in the history of c_i. A sequence of actions τ is a *serialization* of an execution σ if for every client c_i, $acts(c_i) = \tau \mid i$.

Definition 2.1 (Serializable Execution) *Given an execution σ, we say that σ is serializable if there exists a legal serialization τ of σ.*[3]

A set of processes P *implements* a service $\mathcal{E}$ if every execution generated by the system is serializable. We say that P implements $\mathcal{E}$ *in a responsive way* if for every request for which a `request-send` event is sent, one of the processes generate a corresponding `reply-send` event.

Given a pair of actions A_1 and A_2, we say that A_2 is *left commuting* with A_1 if for every sequence of actions τ such that both $\tau \cdot A_1$ and $\tau \cdot A_2$ are legal, $\tau \cdot A_2 \cdot A_1$ is legal too. A_2 is *non left commuting* with A_1 if it is not left commuting with A_1. A_1 and A_2 are said to be *commuting* if A_1 is left commuting with A_2 and A_2 is left commuting with A_1. A_1 and A_2 are said to be *strongly non commuting* if A_1 is non left commuting with A_2 and A_2 is non left commuting with A_1.

So, for example, if we denote by $update(x, v)$ an action which updated an object x with the value v, and by $inquire(x, u)$ an action that inquired the value of an object x and returned u, where $v \neq u$, then $update(x, v)$ is non left commuting with $inquire(x, u)$. Similarly, if we denote by $fetch\&add(x, v, w)$ an action which adds a value v to an object x, and returns its old value w, and by $fetch\&add(x, u, w)$ an action which adds a value u to an object x, and

[3] Note that this definition is not exactly the same as the definition of serializability in data-bases [7]: On one hand, we assume that each action is composed of only one request and one reply. On the other hand, our definition applies to any kind of objects which have sequential specification, and not just to read/write objects.

returns its old value w, where $u, v \neq 0$, then *fetch&add*(x, v, w) is strongly non commuting with *fetch&add*(x, u, w). In the air-traffic control example, non commuting actions could be instructions that may direct two different airplanes to the same air segment.

Two actions A_1 and A_2 are said to be *executed concurrently* in an execution σ if the point of decision for A_1 does not follow the invocation of A_2 in $\xrightarrow{\sigma}$ and the point of decision for A_2 does not follow the invocation of A_1 in $\xrightarrow{\sigma}$. In particular, if A_1 and A_2 are executed concurrently, then the invocation of A_1 and the invocation of A_2 are not ordered by $\xrightarrow{\sigma}$. (They are also concurrent w.r.t. $\xrightarrow{\sigma}$.)

3 Impossibility Result

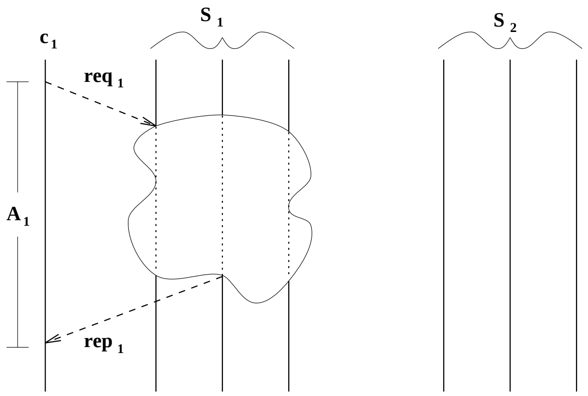
Theorem 3.1 *No set of processes that implement a service in a responsive way can execute strongly non commuting actions concurrently.*

Proof: Assume, by way of contradiction, that there exists a set of processes $P = \{p_1, p_2, \ldots, p_n\}$ that implement a service $\mathcal{E}$ in a responsive way and can execute strongly non commuting actions concurrently. Denote processes $\{p_{i_1}, p_{i_2}, \ldots, p_{i_k}\}$ by S_1 and processes $\{p_{i_{k+1}}, p_{i_{k+2}}, \ldots, p_{i_n}\}$ by S_2, and consider the following execution σ_1 of the system, as depicted in Figure 2(a). During σ_1, all the messages sent from processes in S_1 to processes S_2 or vice versa are lost, but all messages sent from processes in S_1 to processes within S_1 are delivered after a finite delay. Let c_1 be a client which has external links to some processes in S_1. At some point in σ_1, c_1 generates a `request-send` event ev_1 with a request req_1 on one of its external links with processes in S_1, which results in a corresponding `request-receive` event at some process in S_1. Since this is the only request in the system, at a later point in the execution, some process in S_1 generates a `reply-send` event with the reply rep_1. This eventually results in a `reply-receive` event ev'_1 in c_1. Denote the action formed by ev_1 and ev'_1 by A_1.

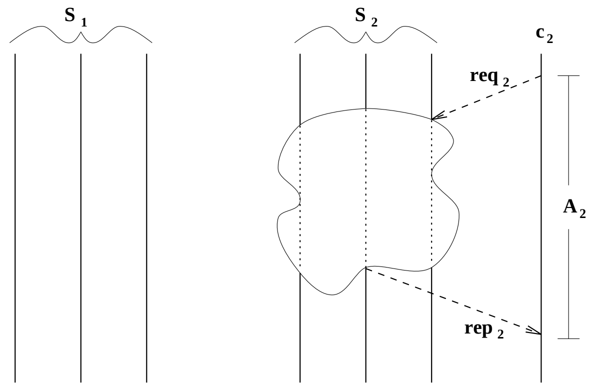
Now consider a symmetric execution σ_2 of the system, as depicted in Figure 2(b). During σ_2, all the messages sent from processes in S_2 to processes S_1 or vice versa are lost, but all messages sent from processes in S_2 to processes within S_2 are delivered after a finite delay. Let c_2 be a client which has external links to some processes in S_2. At some point in σ_2, c_2 generates a `request-send` event ev_2 with a request req_2 on one of its external links with processes in S_2, which results in a corresponding `request-receive` event at some process in S_2. Since this is the only request in the system, at a later point in the execution, some process in S_2 generates a `reply-send` event with the reply rep_2. This eventually results in a `reply-receive` event ev'_2 in c_2.

Assume also that according to the sequential specification of $\mathcal{E}$, A_1 is strongly non commuting with A_2. In particular, neither the sequence $A_1 \cdot A_2$ nor $A_2 \cdot A_1$ are legal.

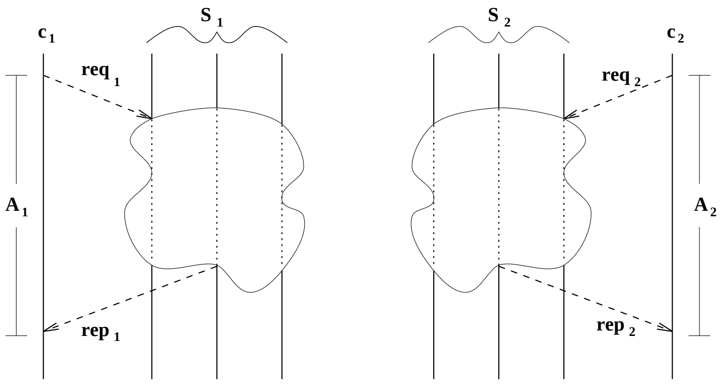
Recall that in both σ_1 and σ_2 all messages sent from processes in S_1 to processes in S_2 or vice versa are lost. Hence, σ, the result of replacing c_2's history and the the histories



(a) Execution σ_1: C_1 interacts with S_1 which are disconnected from S_2.



(b) Execution σ_2: C_2 interacts with S_2 which are disconnected from S_1.



(c) Execution σ: obtained by combining the histories of c_1 and S_1 from σ_1 with the histories of c_2 and S_2 from σ_2.

Figure 2: Proof of Theorem 3.1

of all processes in S_2 in σ_1 with their histories in σ_2 is a possible execution of the system. (See illustration in Figure 2(c).) This is because as far as c_1 and the processes in S_1 are concerned, they cannot distinguish between σ and σ_1, and as far as c_2 and the processes in S_2 are concerned, they cannot distinguish between σ and σ_2.

However, σ is not serializable: since A_1 and A_2 are the only actions in σ, there are only two possible serializations of σ, $A_1 \cdot A_2$ and $A_2 \cdot A_1$. However, by assumption, both serializations are not legal. A contradiction to the assumption that P implements $\mathcal{E}$. ■

Before we state the following theorem, we need to introduce some new definitions: A pair of actions $\langle A, A' \rangle$ such that the point of decision of A is ordered in $\xrightarrow{\sigma}$ before the point of notification of A' is called an *ordered pair of actions*. We say that an ordered pair of actions $\langle A_1, A'_1 \rangle$ is *executed concurrently* with another ordered pair of actions $\langle A_2, A'_2 \rangle$ if neither the point of decision of A'_1 is ordered in σ after the point of notification of A_2 nor the point of decision of A'_2 is ordered in σ after the point of notification of A_1.

Theorem 3.2 *No set of processes that implement a service in a responsive way can execute an ordered pair of actions $\langle A_1, A'_1 \rangle$ concurrently with an ordered pair of actions $\langle A_2, A'_2 \rangle$ if A_2 is non left commuting with A'_1 and A_1 is non left commuting with A'_2.*

The proof of Theorem 3.2 is similar to the proof of Theorem 3.1. The main difference is that in the proof of Theorem 3.2, σ_1 includes both A_1 and A'_1 and σ_2 includes both A_2 and A'_2. The details are omitted for brevity.

Note that in our proof, the network delivers all messages that are sent from processes in S_1 to processes in S_1 and all messages sent from processes in S_2 to processes in S_2 reliably and after a finite delay. Also, we do not assume anything about the protocol used by the processes to perform actions, or their specific semantics. This makes the theorems very strong. In particular, they hold for executions in which more severe failures occur.

Actual implementations of serializable services have to explicitly deal with a more realistic model in which processes may crash and external links may drop requests and replies. In such implementations, clients may resend requests, perhaps on more than one external link, and replies may be sent by the system on more than one external link. As a result, the system must make sure that either all replies to the same request are identical, or that clients have a way of picking the "correct" reply to a request they have submitted. These issues are further investigated in [1, 11, 13].

4 Discussion

The availability limitations on distributed systems that seek to maintain consistency or related safety properties have long been a topic of discussion. Our results demonstrate that there are essentially three ways to implement systems with these properties. In the first class of system, only non-conflicting operations are initiated within a partitioned component. As an example, a component of a system might update a variable that it "owns", or direct an aircraft into a sector of the air space over which it maintains exclusive control. Other components may learn of such actions, but do not initiate conflicting actions of their own.

In the second class of system, conflicting operations can be initiated concurrently in disconnected components. Such systems must delay action until a sufficient degree of global communication is reestablished to permit an unambiguous event ordering to be determined. Keidar's work [15], and that of Amir [1], are examples of techniques that implement this methodology. Inevitably, however, this approach exposes actions to high latencies.

The Isis primary-partition approach [8] represents a third "approach" that mixes elements of the other two. In this system, a single primary component has the right to initiate potentially conflicting actions, and other components are understood to be potentially inconsistent; they must shut down and restart by state transfer from the primary later.

Our results thus shed light on a fundamental design option for distributed systems development. Some modern distributed computing environments such as Horus [19] and Transis [9] permit the developer to configure system properties to precisely match the needs of the application, as hence to balance the tradeoff between availability and consistency on a per-application basis. Our results represent a tool for making such decisions. Alternatively, many distributed computing environments offer just a single consistency option that all user's must accept [17, 18]. Our work is also applicable to this class of systems, illuminating the application-level consequences of such design decisions.

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, S. Kramer, and D. Malki. Membership Algorithms in Broadcast Domains. In *Proc. of the 6th International Workshop on Distributed Algorithms, Lecture Note in Computer Science #647*, pages 292–312, November 92.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message Passing Systems. In *Proc. of the 9th ACM Symposium on Principles of Distributed Computing*, pages 363–375, 1990.

[4] H. Attiya and R. Friedman. A Correctness Condition for High-Performance Multiprocessors. In *Proc. of the 24th ACM Symp. on the Theory Of Computing*, pages 679–690, May 1992. Revised version: Technical Report #767, Department of Computer Science, The Technion. Submitted for publication.

[5] H. Attiya and J. Welch. Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[6] 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.

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

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

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

[10] M. Fischer, N. Lynch, and M. Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[11] R. Friedman, I. Keidar, D. Malki, K. Birman, and D. Dolev. Deciding in Partitionable Networks. Technical Report TR95–1554, Department of Computer Science, Cornell University, March 1995.

[12] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical Report TR95–1491, Department of Computer Science, Cornell University, March 1995.

[13] R. Friedman and A. Vaysburd. Implementing Replicated State Machines Over Partitionable Networks. Technical report, Department of Computer Science, Cornell University, April 1996.

[14] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

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

[16] M. Kosa. Time Bounds for Strong and Hybrid Consistency for Arbitrary Abstract Data Type. Technical report, Dept. of Computer Science, Tennessee Technological University, 1995.

[17] L. Moser, P. M. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.

[18] M. Reiter. Distributed Trust with the Rampart Toolkit. *Communications of the ACM*, 39(4):70–74, April 1996.

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