Transis: A Communication Sub-System for High Availability

Yair Amir, Danny Dolev, Shlomo Kramer, Dalia Malki

Computer Science department
The Hebrew University of Jerusalem
Jerusalem, Israel

Technical Report CS91-13

April 30, 1992

Abstract

This paper describes *Transis*, a communication sub-system for high availability. Transis is a transport layer package that supports a variety of reliable multicast message passing services between processors. It provides highly tuned multicast and control services for scalable systems with arbitrary topology. The communication domain comprises of a set of processors that can initiate multicast messages to a chosen subset. Transis delivers them reliably and maintains the *membership* of connected processors automatically, in the presence of arbitrary communication delays, of message losses and of processor failures and joins. The contribution of this paper is in providing an aggregate definition of communication and control services over broadcast domains. The main benefit is the efficient implementation of these services using the broadcast capability. In addition, the membership algorithm has a novel approach in handling partitions and remerging; in allowing the regular flow of messages to continue; and in operating symmetrically and spontaneously.

1 Introduction

This paper provides an overview of *Transis*, a communication sub-system for high availability. Transis is developed as part of the High Availability project at the Hebrew University of Jerusalem. Transis supports a variety of reliable multicast message passing services between processors. The communication domain comprises of a set of processors that can initiate multicast messages to a chosen subset. The Transis layer is responsible for their delivery at all the designated destinations. The environment is dynamic and processors can come up and may crash, may partition and re-merge. Transis maintains the *membership* of connected processors automatically. This maintenance of correct membership is a fundamental service for constructing fault tolerant applications that use the Transis communication services.

Transis is based on a novel communication model that is general and utilizes the characteristics of available hardware. The underlying model consists of a set of processors that are clustered into *broadcast domains*, typically a broadcast domain will correspond to a LAN. The communication facility within a broadcast domain is a nonreliable broadcast. The broadcast domains are interconnected by point-to-point (non-reliable) links to form the *communication domain*.

Transis provides the communication and membership services in the presence of arbitrary communication delays, of message losses and of processor failures and joins. However, faults do not alter the message contents. Furthermore, messages are uniquely identified through a pair $< sender, counter >$. This requires that processors that come up are able to avoid repeating previous message identifiers. As Melliar Smith et al. noted ([17]), this may be implemented by using an incarnation number as part of the message identifier; The last incarnation number is saved on a nonvolatile storage.

One of the leading projects in this area is the ISIS system [5]. ISIS provides services for constructing distributed applications in a heterogeneous network of Unix machines. The services are provided for enhancing both performance and availability of applications in a distributed environment. ISIS provides reliable communication for process-groups and various group control operations. It supports a programming style called *virtual synchrony* for replicated services: The events in the system are delivered to all the components in a consistent order, allowing them to undergo the same changes as if the events are synchronous ([8, 7]). Another system providing high availability services is described in [13, 15]. They show how to replicate a service efficiently using a 'lazy' asynchronous form of updating. However, the information required for the ordering of updates is carried by the user requests. Our service definitions are greatly influenced by the ISIS experience and the virtual synchrony concept.

Transis offers the following set of services:

1. **Atomic** multicast: guarantees delivery of the message at all the active sites. This service delivers the message immediately to the upper level.

2. **Causal** multicast: guarantees that if the sending of m' causally follows the delivery of m (defined precisely below), then each processor delivers m before delivering m'.

3. **Agreed** multicast: delivers messages in the same order at all sites. There are various protocols for achieving the agreed order, some not involving additional messages ([16, 19]), others

involving a central coordinator ([9, 6]). We are investigating this problem and propose insights into it in ([2]).

4. **Safe** multicast: delivers a message after all the active processors have acknowledged its reception.

5. **Membership**: maintains the membership at each processor in such a way that all connected processors agree on the series of configuration changes and deliver the same set of messages before installing each configuration change.

The user can use any type of multicast service for each message. These services resemble the ISIS approach, however the design and implementation differ. The main benefit of the Transis approach is that it operates over nonreliable communication channels and makes an efficient use of the network broadcast capability. A full description of all of the Transis' services is beyond the scope of this paper. In this paper, we give an overview of the system structure and its basic protocols. The interested reader is referred to [1, 2] for more details.

Melliar-Smith et al. suggest in [16, 17] a novel protocol for reliable broadcast communication over physical LANs, the *Trans* protocol. Similar ideas appear in the Psync protocol ([19]). These protocols use the hardware broadcast capability for message dissemination and a combined system of ACKs and NACKs to detect message losses and recover them. They work efficiently in broadcast networks with marginal loss rates. Melliar-Smith et al. provide the *Total* protocol for total ordering of messages over Trans ([16]), and show how to maintain agreed membership using this total order ([17]). The basic building block of Transis - *Lansis* - is motivated by these ideas and provides all the services over a single broadcast domain. However, it differs from the upper level membership and message-ordering services they provide.

The problem of maintaining processor-set membership in the face of processor faults and joins is described in [10]. As noted by others ([12, 11, 16]), solving the membership problem (or the equivalent problem of total ordering of messages) in an asynchronous environment with faults is impossible. Transis contains a new membership algorithm that handles any form of detachment and re-connection of processors, based on causally ordered messages. This extends the membership algorithm of Mishra et al. ([18]). Our approach never allows blocking but rarely extracts live (but not active) processors unjustfully. This is the price paid for maintaining the membership in consensus among all the active processors and never blocking. This overcomes the main shortcoming of the Total algorithm which may block with small probability in face of faults ([16]). The "approximate" membership enables the simple and efficient solutions of the rest of the control services, such as the agreed-multicast. The Transis membership algorithm achieves the following properties:

- Handles partitions and merges correctly.
- Allows regular flow of messages of all the supported types while membership changes are handled.
- Guarantees that members of the same configurations receive the same set of messages between every pair of membership changes.

The contribution of this work is in providing an aggregate definition of communication and control services over a broadcast domain. It demonstrates how to implement all the services efficiently over LANs. It tackles the practical aspects of these protocols as well; Lansis is implemented and fully operative, and achieves encouraging, preliminary performance results. Another important contribution is the membership algorithm. It has a novel approach in handling partitions and re-merging; In allowing the regular flow of messages to continue; And in operating symmetrically and spontaneously.

2 Rationale

Distributed systems are becoming common in most computing environments today. Figure 1(a) shows a standard distributed environment, consisting of various LANs interconnected via gateways and point-to-point links.

The fast evolution of distributed systems gave way to a "broadcast fright," namely the fear of relying on broadcast information in the system. Indeed, a scalable distributed system should never attempt to maintain *globally replicated* information. Various systems use various kinds of *multicast* services, which are mostly implemented via point-to-point dissemination of messages to all the destinations. In other words, most existing systems model their communication environment as a set of processors interconnected in an arbitrary topology by point-to-point links (see figure 1(b)). This model has the benefit of generality, allowing most to all existing environments to be mapped onto this model. Though general, this model fails to utilize the strongest characteristics of existing communication hardware: all local communication is done through an *exclusive broadcast media* (Ethernet, FDDI, etc.). The use of point-to-point multicast incurs an enormous overrate of messages when the underlying communication system has broadcast capabilities. Furthermore, imposing a logical point-to-point connection renders these systems non-scalable, since the n^2 interconnectivity grows rapidly.

This leads us to believe that the only reasonable communication structure is hierarchical, one that carefully utilizes local clusters (such as LANs) where possible. Figure 1(c) shows our system model comprising of a collection of *broadcast domains* (BDs), interconnected by (logical) point-to-point links. The BDs typically correspond to the physical LANs. However, as the figure indicates, they can encompass multiple LANs that are connected by transparent gateways, or can also be portions of LANs.

Note that when you think of an information system at any level, you will probably have a picture like Figure 1(c) in mind. This suggests that the user should be able to map his application easily onto such a model.

In supporting the broadcast domain services, we certainly do not advocate nondiscriminatory use of broadcast for all purposes. Broadcasting bears a price in interrupts to non-interested processors and in protocol complexity. The goal of this project is to provide services built around broadcast services, as well as guidelines on when and how to use them.

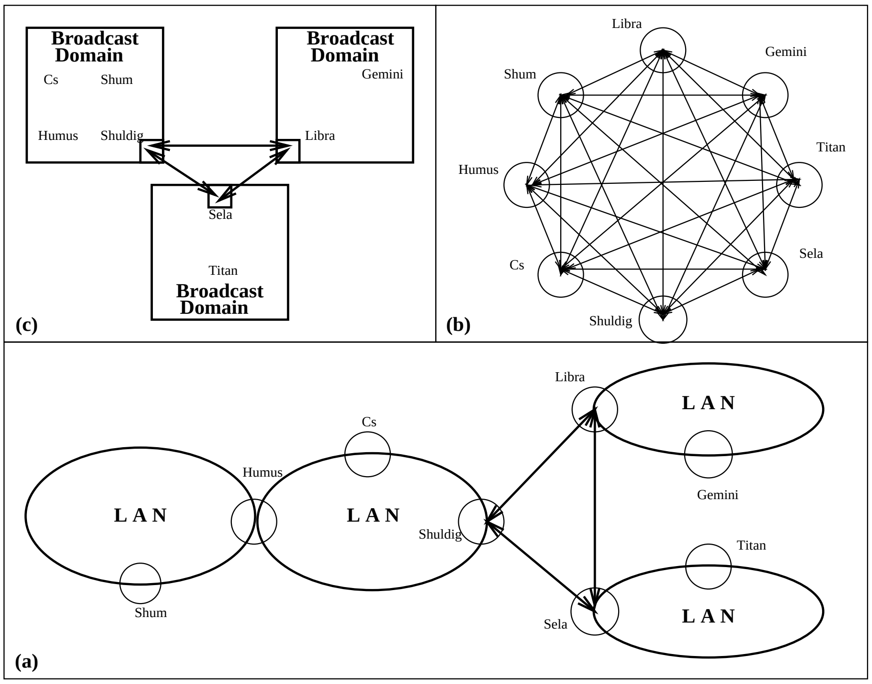

Figure 1: Communication Structure: Reality and Model

3 Transis

Transis supports reliable multicast message passing between processors. The communication domain comprises of a set of processors that can initiate multicast messages to a chosen subset. The service is responsible for their delivery at all the designated destinations.

Transis is a transport layer service that supports processors that are presently connected. At this layer, there is no knowledge or guarantee of delivery of messages to pre-defined processor sets. Therefore, we define the *current configuration set* (CCS) that changes dynamically and consists of the active processors. In order to guarantee message delivery within this set, the transport level acts upon two types of events:

1. Configuration change: either the current configuration is augmented with new processors or a processor is taken out of it.

2. Message delivery: messages are sent by any one of the processors in the current configuration.

Configuration-change events are delivered within the regular flow of messages. Transis guarantees to deliver configuration-change events in a consistent order with messages at all sites. More formally, each processor receives the same set of messages between every pair of configuration-change events. Birman et al. describe this concept in [4, 8] as *virtual synchrony*: It allows distributed applications to observe all the events in the system in a single order. In this way, it creates the illusion of synchronous events. Caveat emptor: in case of a network partitioning, each partition sees a different, non-intersecting set of configuration-change events and messages.

3.1 General Architecture

The software architecture of the services provided by Transis is depicted in Figure 2. The uppermost layer of Transis provides general multicast services for a hierarchy of *Lansis* domains (broadcast domains). The user has control over the mapping of Transis onto the physical system and can use locality and clustering knowledge in order to avoid unnecessary propagation of messages between different domains.

Lansis is the underlying layer that provides a coherent, logical broadcast environment. All the processes participating in the Lansis domain behave alike, whether they physically belong to the same LAN or not. Thus, Lansis can be either implemented over a single LAN, or "simulated" over a general WAN.

The connection between different Lansis domains into the global Transis domain is mediated via the *Xport* mechanism. This mechanism provides a reliable, selective port for transferring messages between Lansis domains.

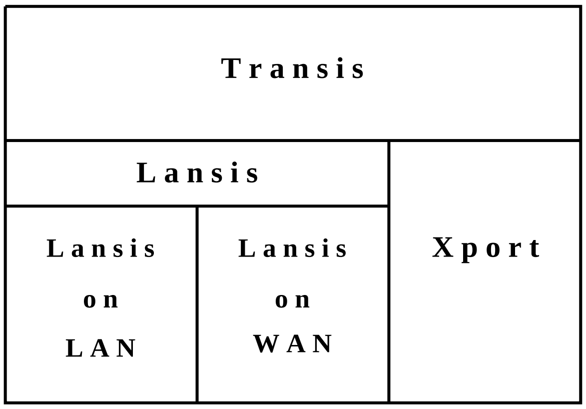

Figure 2: Transis Architecture

3.2 Services Types

Reliable multicast operations guarantee delivery at all destined sites. The underlying communication system model is completely asynchronous and assumes arbitrary communication delays and losses. Therefore, messages arrive at different times and order to distinct processors. In order to coordinate the delivery of messages at different sites, Transis provides various multicast atoms enabling the user to correlate delivery event with other events. For example, one of the multicast atoms provides *causal order delivery* of messages: if the event of sending a message m_2 follows the delivery of another message m_1 at the sending processor, then all the destinations of both m_1, m_2 will deliver m_1 before m_2.

In addition communication problems, there is added difficulty incurred by the crashing and recovery of processors in the system. Transis provides a membership service and reports to the upper level about changes in the current configuration set.

The additional multicast atoms do not incur extraneous message passing, but bear a cost in the latency of the transport protocol. The various atoms can be ranked by the delaying they inflict on the protocol. We define the *index of synchrony* as the number of processors that must acknowledge reception of the message before the protocol delivers the message to the upper level. Figure 3 presents the Transis services and their indices of synchrony. Note that n is *variable* that marks the size of the current configuration set; n changes when the configuration undergoes changes. These changes occur during operation, which further complicates the implementation of the services' protocols. The following sections give the details of the hierarchy of services.

Service-Type	***Index of Synchrony***
S A F E	**n**
A G R E E D	**n/2+1**
C A U S A L	**1**
B A S I C	**1**

Figure 3: Service Hierarchy of the Communication Domain

Atomic

The atomic multicast is the basic service. It guarantees delivery of the message at all the active destined sites. This means that the sites that are active at a time-range around the message-posting time will receive the message [1].

Every processor that receives an atomic message delivers it immediately to the upper level. Thus, the index of synchrony of the atomic service is 1 (including the sending processor).

Causal

The causal multicast atom disseminates messages among all the destined processors such that *causal order* of delivery is preserved. Motivated by Lamport's definition of order of events in a distributed system ([14]), The causal order of message delivery is defined as follows:

If m, m' are broadcast messages sent by p and q respectively, then

$$m \rightarrow m'$$

$$if \quad \text{delivery}_q(m) \rightarrow \text{send}_q(m')$$

Note that $\text{delivery}_q(m)$ and $\text{send}_q(m')$ are events occurring at q sequentially, and therefore the order between them is well defined. The causal delivery order relation for messages is the transitive closure of the above relation. If $m \rightarrow m'$ we say that m' *follows* m, or alternatively that m is *prior* to m'. We say that m, m' are *concurrent* if $m \not\rightarrow m', \quad m' \not\rightarrow m$. The causal multicast atom guarantees that if $m \rightarrow m'$ as defined above, then for each processor p that receives both of them,

$$\text{delivery}_p(m) \rightarrow \text{delivery}_p(m')$$

[1]The range of time is system configurable.

The index of synchrony here is 1 as well [2].

Agreed

The agreed multicast delivers messages in the same order at all their overlapping sites. This order is consistent with the causal order. The difference between the causal-multicast and the agreed-multicast is that the agreed-multicast orders **all** the messages. This includes messages that are sent concurrently, *i.e.* there is no causal relation between them. Thus, while the causal-order is a partial order, the agreed-multicast needs to concur on a single total order of the messages. Note that a majority decision does not achieve the agreed order, since the environment is asynchronous and exhibits crashes. The agreed multicast is implemented via the ToTo algorithm ([2]). The index of synchrony in ToTo is $\frac{n}{2} + 1$.

Safe

Sometimes the user is concerned that a specified message is received by all the destined processors. The safe multicast provides this information, and delivers the message to the upper level only when all the processors in the current configuration set have acknowledged reception of the message. The safe service does not block despite processor crashes. The index of synchrony here is n.

Membership

Transis is designed to operate in a dynamic environment where processors can come up and may crash, may partition and re-merge. The Transis system preserves a *locality* principle, guaranteeing its services for the *currently live* processors. The set of currently live processors is automatically maintained via the membership algorithm (described below in the membership section). Changes in the membership are delivered to the upper level as special *configuration-change* events. The membership algorithm determines the exact range of locality.

The membership algorithm has the following properties:

- Handles partitions and merges correctly.
- Allows regular flow of messages of all the supported types while membership changes are handled.
- Guarantees that members of the same configurations receive the same set of messages between every pair of membership changes.

3.3 Lansis

This section defines the architecture and protocols of Lansis in a broadcast domain. Intuitively, we think of a broadcast domain (BD) as a logical broadcast LAN, which provides reliable and diverse broadcast operations. Every message posted to Lansis by one of the processors is seen by all the processors. All the internal ACKs and NACKs employed by Lansis are seen by all of them too.

[2] In practical implementation, we recommend that the sending processor wait for acknowledgement from at least another processor before delivery, in order to confirm that the message is successfully posted on the network. Therefore, in practice, the index of synchrony is 2.

However, unlike broadcast LANs, messages in the BD do not necessarily arrive at the processors in the same order. The order of arrival depends on the type of service used, as defined in the previous section. The BD is a logical structure, that may be implemented over general topologies.

Lansis is an environment for disseminating broadcast messages. It is best suited for LANs, as explained below in *Lansis on LAN*. However, Lansis can encompass diverse topologies. The logical role of processors is the same in WANs, though the cost and performance are likely to suffer.

Lansis on LAN

In order to understand the need for Lansis over LAN let us examine the following example of a replicated database: there are n identical replicas of data files, $d_1 ... d_n$. The *update* operations are immediately propagated to all the replicas, and the *query* operations are serviced locally by each replica, giving the most up-to-date values. In most system implementations (*e.g.* [3]), each update operation involves contacting with each replica via point-to-point communication and transmitting the update message to it (in addition to ordering-messages sent by the coordinator in order to set a total order of the update messages). If all the replicas reside on a physical broadcast LAN, this method incurs an enormous overate of messages: the same message is posted $n-1$ times over the network, where theoretically it can be seen by all the replicas, but all but one ignore it because it is not destined for them. Also, $n-1$ ACKs need to be collected by the sender.

The reason that the network broadcast is not used as is for this service is that it is not completely reliable, and messages get lost. We have identified three causes of message losses:

1. Hardware faults incurred by the network.
2. Failure to intercept messages from the network at high transfer rates due to interrupt misses.
3. Software-buffers overflow resulting from the protocol behavior.

While the first cause is almost marginal and is expected to become extinct when technology improves, the last two reasons will remain and even become more acute when newer, faster networks (such as FDDI) are used. Therefore, it is up to the software protocols to handle message losses and control the flow of message dissemination.

The Lansis Protocol

The Lansis protocol is based on the principle that messages can be heard by all the processors. If a message is lost by some of the processors, there are many other processors in the network that have heard it and can retransmit it. Lansis uses a combined systems of piggybacked ACKs and NACKs in order to deliver messages to all the processors. This principle idea of Lansis is motivated by the Trans algorithm [16] and the Psync protocol [19]. However, it varies considerably in its implementation considerations, in the variety of services it provides, and in the membership control.

Every processor transmits messages with increasing serial numbers, serving as message-ids. We mark the messages transmitted by P_A: A_1 , A_2 , A_3 , $\cdots$. An ACK consists of the *last* serial number of the messages delivered from a processor. ACKs are piggybacked onto broadcast messages. A fundamental principle of the protocol is that each ACK need only be sent once. The messages that follow from other processors form a "chain" of ACKs, which implicitly acknowledge former messages in the chain, as is the sequence:

$$A_1\ ,\quad A_2\ ,\quad a_2B_1\ ,\quad B_2\ ,\quad B_3\ ,\quad b_3C_1\ ,\quad \ldots$$

Processors on the LAN might experience message losses. They can recognize it by analyzing the received message chains. For example, in the following chain, a receiving processor can recognize that it lost message B_3:

$$A_1\ ,\quad A_2\ ,\quad a_2B_1\ ,\quad B_2\ ,\quad b_3C_1\ ,\quad \ldots$$

The receiving processor here emits a *negative-ACK* on message B_3, requesting for its retransmission. The delivered messages are held for backup by all the receiving processors. In this way, retransmission requests can be honored by any one of the participants. Thus, once a message is posted on the network, the role of carrying it to its destinations becomes the network's responsibility. Obviously, these messages are not kept by the processors forever. The 'Implementation Considerations' section below explains how to keep the number of messages for retransmission constant.

If the LAN runs without losses then it determines a single total order of the messages. Since there are message losses, and processors receive retransmitted messages, the original total order is lost. We cannot expect two different processors to observe the same message order. Thus, it is the piggybacked information that determines the partial order of message passing.

In Lansis, a new message contains ACKs for all the causally deliverable (non-acked) messages. This is an important difference between Trans and Lansis, where the ACKS in Lansis acknowledge the deliverability of messages rather than their reception. Therefore, they reflect the user-oriented cause and effect relation *directly*. In Trans, on the other hand, the partial order does not correspond to the user order of events and is obtained by applying the OPD predicate on the acknowledgements [16].

It is easy to see that this difference does not introduce deadlocks (a message will not be delayed forever) nor does it render its correctness (atomicity and causality are preserved). From the practical point of view we prefer to delay the ACKs when delivery is delayed, allowing us to control the progress of the system. Furthermore, the delivery criteria in Lansis is significantly simplified by this modification.

We think of the causal order as a directed acyclic graph (DAG): the nodes are the messages, the arcs connect two messages that are directly dependent in the causal order[3]. The causal graph contains all the messages sent in the system. The processors see the same DAG, although as they progress, it may be "revealed" to them gradually in a different order.

Implementation of Services

All the Transis services are provided by Lansis. The services are provided by delivering messages that reside in the DAG. They differ by the criteria that determine when to deliver messages from the DAG to the upper level. These criteria operate on the DAG structure and they do not involve external considerations such as time, delay etc.

[3] An arc from A to B means that B acks A. Therefore A "generated" B.

The delivery criteria are as follows:

1. Atomic: Immediate delivery.

2. Causal: When all direct dependents in the DAG have been delivered.

3. Agreed: We have developed a novel delivery criterion called ToTo that achieves best case delay of $\frac{n}{2} + 1$ messages [2]. The ToTo criterion is strictly better than the 'all-ack' criterion *i.e.* at extreme cases, it always delivers messages that have n ACKs, but typically it requires less than n ACKs. ToTo is based on a dynamic membership, therefore it admits messages in a bounded delay determined by the underlying membership algorithm.

4. Safe: When the paths from the message to the DAG's leaves contain a message from each processor. The safe criterion changes automatically when the membership changes.

The membership algorithm in Transis is described in a separate section below.

Implementation Considerations

Since Transis is a practical system, it also concerns itself with the implementation requirements and feasibility of the protocols. The transport protocol needs to keep the retransmission buffers finite by discarding messages that were seen by all the processors. Furthermore, it needs to regulate the flow of messages and adapt it to the speed of the slowest processor. Waiting for NACKs is not good enough. We observed by experimenting a naive implementation that recovery from omission is costly and the system may fall into a cascade of omissions due to this belated response.

Lansis employs a novel method for controlling the flow of messages. This method attempts to avoid 'buffer-spill' as much as possible in order to prevent message losses, and further slows down when losses occur. Define a *network sliding window* as consisting of all the received messages that are not acked by all yet. Each processor computes this window from its local DAG. Note that this window contains messages from *all* the processors, unlike synchronous protocols like TCP/IP which preserve only sent-out messages. The sliding window determines an adaptive delay for transmission by the window size, ranging from the minimal delay at small sizes and slowing up to infinite delay (blocked from sending new messages) when the window exceeds a maximal size. The system does not block indefinitely though. If the window is stuck for a certain period of time, the membership algorithm interferes and removes faulty processors form the configuration. This releases the sliding-window block and the flow of messages resumes.

Performance of Lansis

This section gives preliminary performance results of Lansis over a LAN. Lansis can operate correctly over a general WAN using any routing algorithm. However, the performance will be different; Our main interest is in the operation of Lansis over LANs.

Lansis is a small package implemented on top of of UDP broadcast sockets. Should information be disseminated to more than two parties, Lansis already performs better than TCP/IP. For example, it achieves a throughput of 160 1K-messages per second in an Ethernet network of ten Sun-4 workstations. This throughput is achieved in the most requiring conditions, when all the participants emit messages concurrently and receive all the messages. In comparison, the transmission rate via TCP/IP in one direction between two parties in this network is about 350K/sec. Moreover, the Lansis protocol exhibits only marginal degradation in performance as more machines are added to the broadcast domain.

Lansis is a useful tool when used carefully. It is important to remember that it bears a cost:

- Extraneous communication when messages are carried over to non-interested destinations. Also, completely noninterested processors on the LANs are interrupted by the broadcast traffic.
- Increased processing time of the transport layer, concerned with maintenance of the mutual backup data structures and the protocol flow control.
- Space overhead used for the messages backup.

4 Xport

The *communication domain* (CD) comprises of a hierarchy of broadcast domains and provides the multicast message passing services throughout it. This section describes the mechanism and protocols by which Transis extends the scope of its services outside the broadcast domains. The mechanism is called the *Xport* mechanism.

In general, the Lansis protocol might be too demanding on the environment, requiring each processor to observe all the messages and maintain mutual backup. This may be unsuitable for very large systems. The Transis protocol defines the same set of multicast services over a broader range of systems.

Using a hierarchy of BDs instead of one bigger BD may be advantageous in the following ways:

- The first advantage of the Transis is scalability: in a hierarchy of BDs, the messages overate and space overhead is kept within the smaller sets of BDs and therefore can be kept reasonable.
- Secondly, the services are tailored to the system structure. For example, it might be best to maintain each BD within a physical LAN where it benefits the most from the underlying network. The external communication outside the LAN employs the Xport mechanism.
- The Transis protocol enables partitioning the set of communicating processors according to other considerations. For example, our experience shows that it is difficult to balance the Lansis protocol when a LAN contains processors of different speeds. Instead, the slow computers may be coupled into a BD, and the fast computers constitute a separate BD. This reduces the task of controlling flow in the system to the link between the two domains, which is easier to handle.
- Lastly, the application structure may suggest partitioning into communication clusters which are best served by different BDs. The application at large should not suffer from the overheads exhibited by in a single BD, and is served best by a hierarchical communication domain.

The Communication Domain

Figure 4 presents an architecture of a CD comprised of three BDs. A multicast message initiated in a BD may be *exported* to one or more BDs at which it will be *imported*. All export/import activities are handled by a designated member of the BD called the *xport* node. The xport nodes of different BDs are connected by pairs of uni-directional point to point links, called *xlinks*.

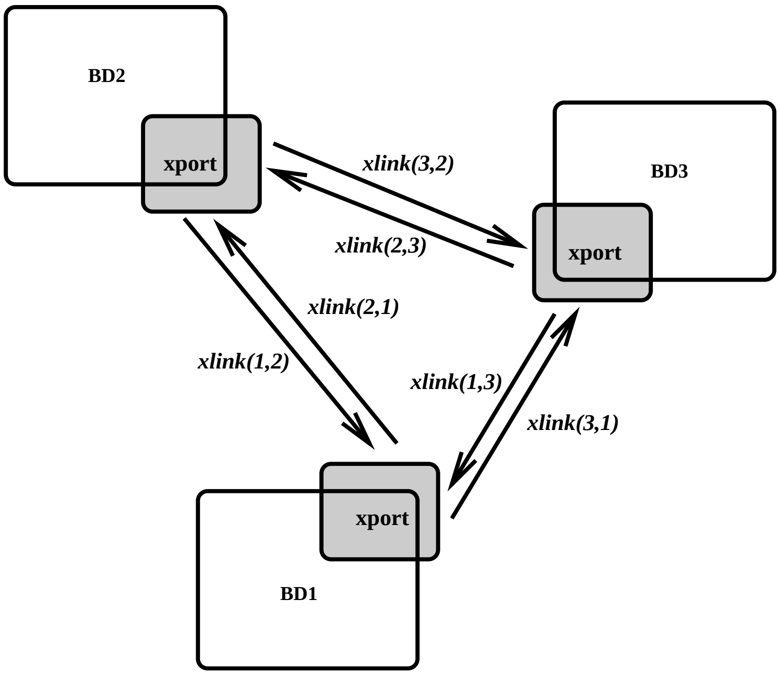

Figure 4: The Communication Domain - a hierarchy of broadcast domains connected by the Xport mechanism.

The CD is mappable to general communication structures in a way which efficiently utilizes hardware characteristics and application structure. Though supplying all of the multicast services with the same semantics as in a broadcast domain, the CD differs in performance and low level behavior. Within a BD each processor obtains all the acknowledgment information that passes in the system, while outside, this information is inaccessible. The performance of message dissemination within a single BD might be different than in a broader CD. The Transis protocol is also more susceptible to faults than Lansis since it depends on gateway connections. In addition, processors only backup other processors within their BD.

The problem of making the Xport mechanism resilient to crash and disconnection faults is solved by replicating xport nodes and xlinks. A service can be replicated using an agreement atom. In Xport, the agreement atom is implemented using the agreed multicast service in the set of xport nodes. Thus, one additional multicast is needed per export or import of an event. In the following sections we will assume reliable xlinks and xport nodes.

Atomic and Causal Multicast in the CD

This section describes the *inter-BD* protocol that extends the atomic and causal multicast services to the communication domain. The inter-BD causal multicast protocol is performed only by the xport nodes, all other nodes perform only the regular intra-BD protocol and are totally oblivious to the fact that messages cross BD boundaries.

The inter-BD causal multicast protocol entails two types of activities:

- *Export*: The xport node exports the message to all relevant BDs.
- *Import*: Messages received from remote BDs are disseminated locally while preserving causal order requirements.

Each xport node maintains a vector of counters v with one entry per xlink in the system[4], $v = (v_1, ..., v_n)$ where n is the number of xlinks in the CD. When an xport node engages to export a message on some xlinks, it increments these links' fields in the vector. It sends the full vector with the counters of all the xlinks in the system. Similarly, when an xport node imports a message, it delays handling it until all prior messages on connected links are received. This is done by comparing the vector components that correspond to all the incoming links. It updates the local vector by taking the pairwise maximum of all the xlinks. Upon importing a message, the xport transmits it in its local BD. This method relies on the continuity of xlink counters, and in case of partitions, the upper level needs to update the counters.

Note that the vector contains an entry per xlink and not per BD (unlike ISIS vector time stamps). The reason for this is that messages may be destined to any number of BDs. In this way non-interested BDs are not concerned with messages not destined to them.

The import and export algorithms are sketched below:

- Import: Upon receiving a message m, stamped with a vector time stamp mv, through xlink inl, wait until for every **incoming** xlink l:

  $$\begin{cases} mv[l] = v[l] + 1 & l = inl \\ mv[l] <= v[l] & l \neq inl \end{cases}$$

  When the condition holds, update v to the pairwise maximum of (v, mv), and multicast m within the BD.

- Export: Upon delivery of a message m initiated in the local BD such that $destinations(m)$ contain external nodes:

  1. Strip m of inner BD piggybacked information (ACKs, NACKs);
  2. Let $outlinks$ contains all the outgoing links to the external destinations of m.
     For every xlink $l \in outlinks$, increment $v[l]$.
  3. Stamp m with v and send it on all outlinks.

It can easily be verified that this protocol extends the causal order across BDs.

[4] n can be reduced by applying considerations similar to those described in [8]

Agreed and Safe Multicast in the CD

The intra-BD agreed multicast extends the agreed multicast in the CD. If messages m_1, m_2 initiated at BD_1, BD_2 respectively, are multicast to processors in BD_1 and BD_2, then all destinations of $(m_1,\ m_2)$ in both BDs will deliver m_1 and m_2 in the same order. It should be noted that only messages destined to more than one BD need to participate in the inter-BD protocol, all other messages are internal to the BD from which they were initiated. The correctness of the agreed multicast service is preserved if these messages participate only in the intra-BD protocol.

The agreed multicast service in the CD is implemented on top of the causal multicast protocol. It is implemented by a cascade of two protocols (see Figure 5):

- An intra-BD agreed multicast protocol. Performed only in the BD from which the message had been initiated. Once local deliverability is reached the message is further delayed until the inter-BD protocol reaches global deliverability.[5] At remote BDs, an imported agreed multicast message is delayed until global deliverability is reached.
- An inter-BD total ordering protocol. Performed only by the xport nodes. The input to the protocol is a stream of partially ordered messages, among which the locally initiated messages are totally ordered. The xport nodes reach an agreement on the order of all relevant messages and broadcast it within their BDs. This agreement can be reached by any algorithm for total ordering of messages.

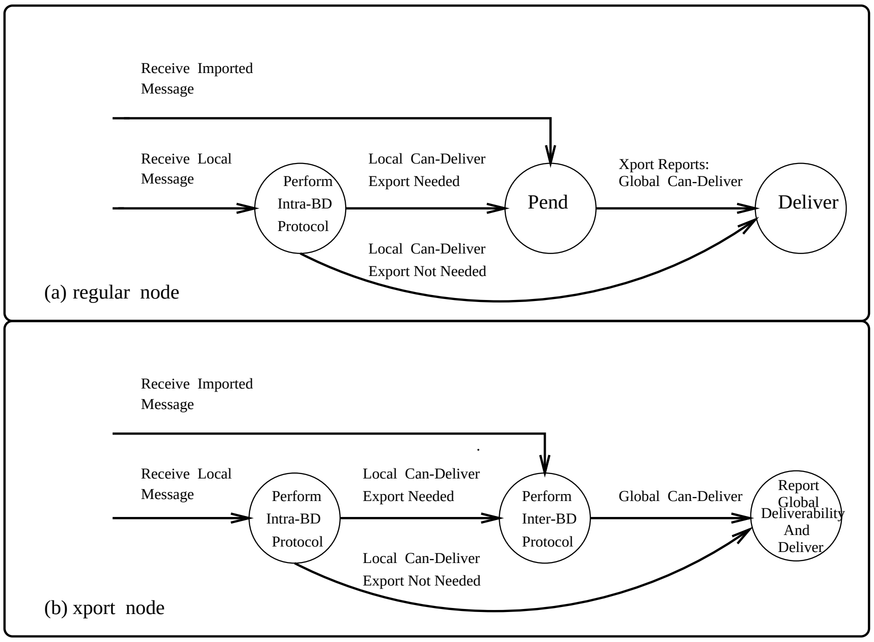

Figure 5: Agreed multicast in the CD: a per message state diagram.

The safe multicast is easily reducible (in a similar way to the agreed multicast) to a cascade of

[5] As noted above, for messages with a destination set contained in the local BD no further delay is needed.

two protocols: an intra-BD all-ack protocol and an inter-BD all-ack protocol in which the relevant xport nodes exchange local deliverability information.

Cost

The concept of a single port through which all messages are imported and exported is natural in many physical topologies. It is often the case that a LAN is connected to other LANs or WANs via a single bridging node. Thus, the Xport mechanism does not introduce any additional communication bottlenecks. Since BDs are typically large and connected by relatively slow links, we expect the communication bandwidth between BDs to be significantly smaller than the communication bandwidth inside a BD. Thus, the relatively simple protocols employed by the Xport mechanism are not expected to be a serious processing burden on the xport nodes.

5 Membership

Every processor holds a private view of the *current configuration* that contains all the processors it has established connections with. We call this view *CCS*, the Current Configuration Set. Note that this is not a user-defined processor-set, but represents the up-to-date knowledge of active processors in the system. All the processors in the current configuration set must agree on its membership. When a processor comes up, it forms a singleton CCS. This initial set is trivially in agreement by all its members. The CCS undergoes changes during operation: processors dynamically go up and down, and the CCS reflects these changes through a series of *configuration changes*. The membership problem is to maintain the CCS in agreement by all its members throughout these changes.

This problem is provably impossible to solve in asynchronous environments with faults([12, 11]). Our membership algorithm circumvents these results by allowing the extraction of live (but not active) processors unjustfully. Consequently, the membership algorithm never allows blocking, and operates within the regular flow of messages. The sections bellow give the essentials of the algorithm and an intuitive claim of its correctness. A full description of the membership algorithm and its proof is provided in [1].

5.1 The Faults Handling Algorithm

This section focuses on a membership algorithm for handling departure of processors from the set of active ones.

Throughout this section, we assume the existence of a starting 'current' membership, *Members*, which is the agreed set of connected members. $Members$ is the lower level's representation of the membership set. Initially, CCS in the upper level contains the same set as $Members$. During the faults protocol, the $Members$ set undergoes changes which might render it temporarily different from CCS. Eventually, these changes are propagated to the upper level and the CCS becomes up-to-date with $Members$.

The fault handling algorithm operates within the regular flow of messages. When processors in the current set fail, the DAG gets filled with messages that require ACKs from the failed processors (such as 'safe' and 'agreed' messages). As a result, the system would block. Therefore the faults need to be detected and considered. The object of the algorithm is to achieve consensus among all the live processors about the failed processors.

Each processor may find out separately about failed processors. The specific method for detecting faults is implementation dependent and irrelevant to the faults algorithm. For example, in the Transis environment, each processor expects to hear from other processors in the CCS within some regular interval. Failing this, it attempts to contact the suspected failed processor through a special safe channel. If this fails too, it decides that this processor is faulty, and emits an F message declaring this processor faulty.

A processor p receiving a set SP of F messages seeks for confirmation from all the remaining processors in the membership. The SP set is ordered by the causal order, and by the lexicographical order among concurrent messages. The *accepting* set, $Accept = Members - Faulties$, must send ACKs for all of SP. This is the deliverability condition for the F messages in SP. When SP is deliverable, the faulty processors are removed from *Members* and the delivery criterion for the regular messages is changed. The block is thus removed, and the next messages from the DAG are delivered to the upper level. Eventually, all the F messages are delivered according to their order in SP. Accordingly, the CCS goes through the configuration changes one by one, that bring it up-to-date with $Members$.

This algorithm assures that the live connected processors agree on every failed processor. Moreover, all processors deliver the same last message from a failed processor, before installing the new configuration. Note that the SP set is dynamic, and may be different at distinct processors. However, the following two properties are preserved:

- All the connected processors deliver the same set of messages before delivering each configuration change. In particular, all the configuration changes are delivered in the same order.
- All the 'safe' messages are delivered with the same safe-set of the processors that acked them.

The following section introduces the data structures of the algorithm and some notations. The protocol is given in pseudo code in an event-driven fashion, describing the handling of each incoming message.

Notation

In addition to regular messages, there are F messages in the form F(p), where F(p) suggests that an existing processor p is faulty.

Each processor maintains a few data structures and a set of operators on them:

1. The direct acyclic graph (DAG^p) of the received messages that are pending to be delivered. It is a common variable to all operations, and we omit it as an explicit parameter further on.
2. $Members^p$ - The current set of connected processors.
3. $SP^p = \{F(q1), \ldots, F(qk)\}$ - The ordered set of non-delivered special (faulty) messages.
4. $Accept^p = Members^p - \{q \mid F(q) \in SP^p\}$.

Below, we neglect to write the '^{p}' superscript when it is obvious from the context.

The Algorithm

The faults protocol specifies the delivery criteria for F messages, and the effect taken by delivering them. The protocol operates on a full DAG without losses. The acking mechanism and the recovery of missing messages are part of the Transis package.

1. When the event of communication-break with processor q occurs, send F message F(q).
2. When receiving a message from a failed sender (*i.e.* F($sender$) $\in SP$) discard it, unless it is a message that the Transis layer asked for recovery.
3. When receiving a regular message insert it to the DAG.

4. When receiving F message F(q) set:

$$SP = SP \cup \{F(q)\}$$
$$Accept = Accept - \{q\}$$

   Stop acking messages from q.

5. When receiving ACK for any message in SP, check the following delivery criterion for SP: All the processors in *Accept* acked all the messages in SP. If the criterion is met, change:

$$Members = Accept$$
$$SP = \emptyset.$$

After each event of the algorithm, it checks if the next message in the DAG is deliverable. If so, the message is delivered to the upper level and removed from the DAG. Note: this includes both regular messages and F messages. The upper level notes the configuration change only when the F messages are delivered to it according to their order in the DAG. F messages are taken last in their concurrency set.

5.2 General Description of the Join Algorithm

In this section we give a non-formal description of the join algorithm, in order to provide intuition on it.

At a normal state, all the connected processors agree on the membership in their current configuration. (And a recovered or a newly started processor is a single member set). The join algorithm is trigerred when a processor detects a "foreign" message in the broadcast domain. The current set attempts to merge with the foreign set or sets. Since it operates in a broadcast domain, we expect this to typically happen at the other set(s) and the algorithm works symmetrically, *i.e.* there is no joining-side and accepting-side. Note that actual simultaneity is not required for correctness. The closer the sets commence, the sooner they will complete the join protocol. The purpose of the join algorithm is to reach an agreed decision on a joined-membership. There are three logical phases (see Figure 6):

1. Each connected set "publishes" its membership through a special attempt-join (AJ) message.

2. AJ messages are observed by foreign processors. Each processor independently suggests a merged configuration in a *JOIN* message. JOIN messages are observed and acked by foreign processors. A JOIN message that is confirmed by all of its members is accepted as the new configuration.

3. The common DAG of messages of the new configuration is rooted at the accepted JOIN message and contains messages that follow it.

A JOIN message that is considered for delivery divides all the messages to those that precede it and those that follow it. Messages that are prior to it are delivered within their original configurations. If the JOIN message is delivered, messages that follow it are delivered in the joined

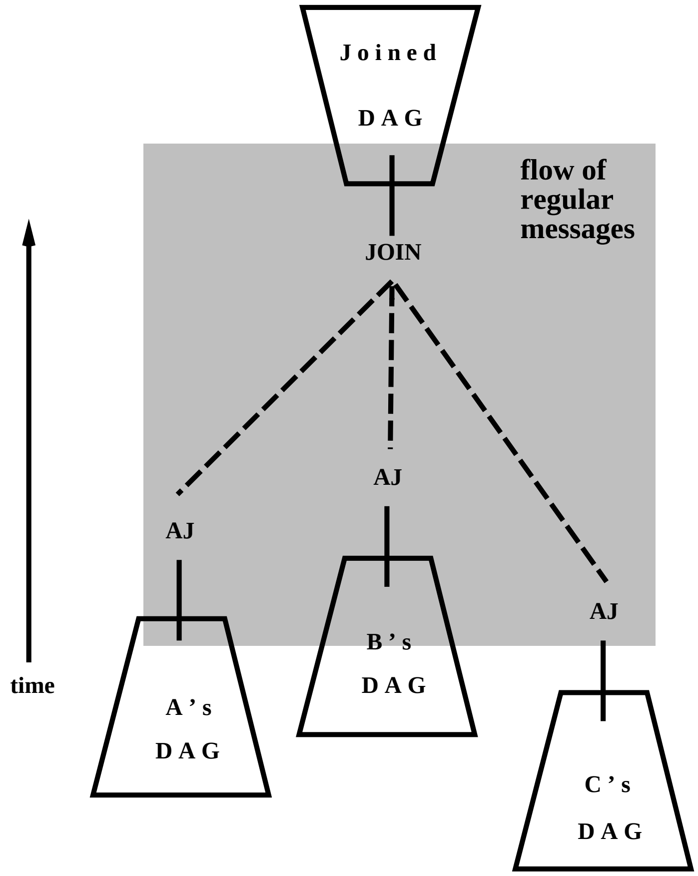

Figure 6: Join Procedure - 3 Logical Phases

set. Thus, the handling of special messages before and after a JOIN message is essentially the same as the faults algorithm. It can be shown that all connected processors deliver the same JOIN message. Naturally, if a partition occurs during the join procedure, two *detached* processors might not maintain this.

A few details are worth noting: Each set must agree on the representing state of the set, for the AJ message. The last message that caused a configuration change is taken as the representing state of the set. The state representation includes the membership and a complete *vector time-stamp*, that holds the counter of the last delivered message from each member processor. Any member can transmit this agreed state when trying to join other sets, provided that there are no pending messages for configuration changes.

After sending a representative AJ message of the set, each processor in each set delays before attempting to join other sets (during this time, regular flow of messages within the set continues). The purpose of delaying is to allow as many other sets as possible to reach an agreement and transmit their AJ message. When the delay completes, each processor independently transmits a suggested JOIN message containing all the AJ/JOIN messages it received. If a JOIN message sent by another processor already contains this suggestion, the processor avoids re-suggesting it and simply acks it.

The join algorithm must guarantee consistency of the join procedure, such that all the connected members agree on the accepted JOIN message. Therefore, every processor *commits* to one JOIN suggestion by either sending it or acking it. However, if any of the processors required to confirm this JOIN message sends a different suggestion before acking it, a new JOIN message combining both suggestions is sent. Note that different suggestions from *foreign* processors must be ignored after committing to one JOIN message. However, if a *required* member of the committed JOIN message sends a different suggestion, it is safe to move to a new JOIN message.

Combining different suggestions such as JOIN, AJ and F messages is done by taking all the live processors in all the messages. There may be a delicate rare situation when a processor *moves* from one configuration to another without the former set's knowledge (yet). The merger recognizes this fact through the time-stamp of this processor in two AJs in the JOIN message. It marks the processor faulty in its former set, *i.e* the one with the smaller counter. When there are multiple, concurrent identical JOIN messages they are considered as one and do not require merging.

Faults occurring during the join procedure are handled in the usual way, where F messages following the JOIN message must be acked by the joined membership set. Faults that are reported concurrently with a JOIN message are a special case of "different suggestions" sent by required members. In the case of concurrent JOIN and F messages, a new JOIN message merges them. This allows a JOIN message to be delivered even if members of its suggested membership fail during the join procedure.

A JOIN message can be delivered when its proposed membership set acks it. There may be faulty processors in this set, in which case the remaining processors must ack both the JOIN message and all the F messages.

6 Conclusions

Most transport-layer packages today provide point to point communication, or non-reliable multicast. We have shown how to generalize methods employed by these layers to support multicast, and how to incorporate them gracefully into existing systems. Our preliminary implementation over a heterogeneous network of Sun-4 and Sun-3 machines shows promising results. Over more than three machines, performance is already better than standard point to point protocols.

Fischer, Lynch and Paterson ([12], and later Dolev, Dwork and Stockmeyer, [11]) have shown that without some sort of synchronization no agreement is possible. Our membership algorithm circumvents these results by introducing a dynamic local group upon which agreement is based. It is true that in some extreme cases, processors may wrongly decide that another processor has failed, but when this is found out, the system recovers. By maintaining membership at the lowest level, we simplify the implementation of all the other services. For example, in [2] we show how to construct the agreed multicast on top of the dynamic membership.

The membership algorithm operates symmetrically and spontaneously. Its novel aspect is the ability to join partitions. To the best of our knowledge all of the existing membership algorithms (*e.g.* [17, 18, 10, 8]) handle the joining of single processors only. This feature is crucial since partitions do occur. For example, when the network includes bridging elements partitions are likely to occur.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms in broadcast domains. Technical Report CS92-10, dept. of comp. sci.., the Hebrew University of Jerusalem, 1992.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Total ordering of messages in broadcast domains. Technical Report CS92-9, dept. of comp. sci.., the Hebrew University of Jerusalem, 1992.

[3] A. Bhide and S. P. Morgan. A highly available network file server. RC 16161, IBM Research, May 1990.

[4] K. Birman, R. Cooper, and B. Gleeson. Programming with process groups: Group and multicast semantics. TR 91-1185, dept. of Computer Science, Cornell Uni., Jan 1991.

[5] K. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS System Manual.* Dept of Computer Science, Cornell University, Sep 90.

[6] K. Birman and T. Joseph. Realiable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, February 1987.

[7] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Ann. Symp. Operating Systems Principles*, number 11, pages 123–138. ACM, Nov 87.

[8] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. TR 91-1192, dept. of comp. sci., Conrell University, 91. revised version of 'fast causal multicast'.

[9] J. M. Chang and N. Maxemchuck. Realiable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, August 1984.

[10] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. Research Report RJ 5964, IBM Almaden Research Center, Mar. 1988.

[11] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *J. ACM*, 34(1):77–97, Jan. 1987.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *J. ACM*, 32(2):374–382, 1985.

[13] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Lazy replication: Exploiting the semantics of distributed services. In *Ann. Symp. Principles of Distributed Computing*, number 9, pages 43–58, August 90.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 78.

[15] B. Liskov and R. Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Ann. Symp. Principles of Distributed Computing*, number 5, August 86.

[16] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Trans. Parallel & Distributed Syst.*, (1), Jan 1990.

[17] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Membership algorithms for asynchronous distributed systems. In *Intl. Conf. Distributed Computing Systems*, May 91.

[18] S. Mishra, L. L. Peterson, and R. D. Schlichting. A membership protocol based on partial order. In *proc. of the intl. working conf. on Dependable Computing for Critical Applications*, Feb 1991.

[19] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, August 89.