Transis: A Communication Sub-System for High Availability

Yair Amir, Danny Dolev, Shlomo Kramer, Dalia Malki

The Hebrew University of Jerusalem, Israel

Abstract

This paper describes Transis, *a communication sub-system for high availability. Transis is a transport layer that supports reliable multicast services. The main novelty is in the efficient implementation using broadcast. The basis of Transis is automatic maintenance of dynamic membership. The membership algorithm is symmetrical, operates within the regular flow of messages, and overcomes partitions and re-merging. The higher layer provides various multicast services for sets of processes.*

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 is a transport layer package that supports a variety of reliable message passing services within arbitrary topology networks. Messages are being addressed (multicast) to several destinations. The topology is divided into broadcast domains. The focus of this paper is the dissemination of information within a broadcast domain. Transis is responsible for the delivery of messages at all the designated destinations. It *receives* messages from the network, handles all recovery and consistency requirements, and *delivers* messages for processing at each processor.

The main difficulty facing the designer of a distributed application is the consistency of the information disseminated, and the control over the dissemination of that information. Experiments show that current broadcast protocols are not sufficient for reliable information dissemination; at an average to high load on a LAN, many messages do not arrive to all targets. Thus, the designer of a distributed system would wish for a trasport layer that provides a guaranteed delivery–and–consistency of messages sent to multiple targets. Having such a layer, most distributed applications become much easier to implement and to maintain.

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 model assumes hardware and software support for non-reliable broadcasts (multicasts) within a broadcast domain. The *Lansis* building block provides the Transis transport layer services within a broadcast domain. 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 ([15]), 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 [4]. 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 ([7, 6]). Our service definitions are greatly influenced by the ISIS experience and the virtual synchrony concept.

Lansis offers the following set of services:

1. **Membership**: maintains the membership of processors.
2. **Basic** multicast: guarantees delivery of the message at all the active sites. This service delivers the message immediately to the upper level.
3. **Causal** multicast: guarantees that delivery order preserves causality (defined precisely below).
4. **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 ([14, 17]), others involving a central coordinator ([8, 5]). We developed a symmetrical algorithm that achives this (see [2]).
5. **Safe** multicast: delivers a message after all the active processors have acknowledged its reception.

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.

Melliar-Smith et al. suggest in [14, 15] a novel protocol for reliable broadcast communication over physical LANs, the *Trans* protocol. Similar ideas appear in the Psync protocol ([17]). These protocols use a combined system of ACKs and NACKs to detect message losses and recover them. Melliar-Smith et al. provide the *Total* protocol for total ordering of messages over Trans ([14]), and show how to maintain agreed membership using this total order ([15]). 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 membership and message-ordering services they provide.

Transis contains a new membership algorithm that handles processor crashes and joining, network partitions and merges. 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. The dynamic membership enables the simple and efficient solutions of the rest of the control services, such as the agreed-multicast.

2 Rationale: Utilizing Broadcast

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 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, a BD can encompass multiple LANs that are connected by transparent gateways, or may also be only portions of LANs.

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.

The benefit of utilizing broadcast is enormous: All the desired propeties of point-to-point protocols (reliability, flow control, connection service) and more are provided in our multicast protocols, with superior performance. The 'Lansis performance' section provides performance details.

Transis is a transport layer that supports reliable multicast. At the transport layer, there should be no long term guarantee for message delivery, and it is not possible to log messages onto secondary storage. Therefore, Transis is responsible for (a) the delivery of messages to all the currently active and connected processors, and (b) to report accurately about the system state and success of delivery to the user.

3 Lansis: Distributed Services in a BD

The basic building block of the Transis package is *Lansis*. Lansis operates within a single *broadcast domain* (BD). A BD is best suited for a single LAN, or several LANs transparently connected via bridges, as seen below. However, Lansis can encompass diverse topologies if desired, though the cost and performance are likely to suffer. [1]

The Lansis package consists of three logical layers:

1. **Membership.**
2. **Multicast services.**
3. **Session and name service.**[2]

We define the *current configuration set* (CCS) that changes dynamically and consists of the active processors. The basis layer of Lansis, *membership*, automatically maintains the CCS in consensus among all the members of CCS.

The Membership section below sheds some intuition on the membership algorithm. The full details of the algorithm and proof of its correctness are found in [1].

The next layer on top of the dynamic membership is responsible for reliable delivery of messages within the CCS. It delivers both regular messages and special *configuration change* messages to the upper level.

Lansis supports various primitives for coordinating the delivery of multicast messages in different processors (see below).

Configuration-change messages are delivered within the regular flow of messages. Lansis 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 [3, 7] as *virtual synchrony*: It allows distributed applications to observe all the events in the system in a single order. In this way, it creates

[1] 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 unless it is missed. This can be emulated over general topologies, but is best suited for LANs.

[2] The session and name service are not part of the transport layer, but are provided for the Lansis users.

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.

The upper layer is the session layer. It provides the interface for multiple user processes on each processor. Note that only a single Lansis process executes on each processor. The name-service at this layer allows addressing of messages to *processes-sets*, which can be any subset of the active processes. Process sets are formed dynamically by processes that 'join' them. A process that joins a set receives all further messages sent for this set.

Multicast Services

The underlying communication system model of Lansis 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, Lansis provides various multicast services enabling the user to correlate delivery event with other events.

The additional multicast atoms do not incur extraneous message passing, but bear a cost in the latency of the transport protocol. The various services 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 2 presents the Lansis services and their indices of synchrony. Note that n is a *variable* that marks the varying size of the current configuration set.

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 2: Services Hierarchy

Basic

The basic multicast is the elementary 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 [3].

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

Causal

The causal multicast 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 ([13]), The causal order of message delivery is defined as the transitive closure of:

(1) $m \xrightarrow{cause} m'$ if $\text{receive}_q(m) \rightarrow \text{send}_q(m')$

(2) $m \xrightarrow{cause} m'$ if $\text{send}_q(m) \rightarrow \text{send}_q(m')$

Note that '$\rightarrow$' orders events occurring at q sequentially, and therefore the order between them is well defined. The causal multicast atom guarantees that if $m \xrightarrow{cause} m'$, then for each processor p that receives both of them,

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

The index of synchrony here is 1 as well.

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 specific message is received by all the destined processors before taking an action. The safe multicast provides this information, and delivers the message to the upper level

[3] The range of time is system configurable.

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 because of the dynamic membership. The index of synchrony here is n.

3.1 Lansis over LAN

The principle idea of reliable message delivery in Lansis is not new. We are motivated by the Trans algorithm ([14]) and the Psync algorithm ([17]). For completeness of this paper, we repeat the basics of the algorithm here. There are some novel aspects in our implementation and the flow control handling. However, the main contribution of this paper is in showing how to combine these ideas in an aggregate of distributed services in a dynamic environment.

The Lansis protocol exploits the network broadcast capability for disseminating messages to multiple destinations via a single transmission (if a multicast capability is supported, as in [10], Lansis can exploit it to address only the participating machines). There are various multicast protocols that utilize broadcast hardware, such as UDP ([18]), and the IP extended multicast protocol ([10]). However, they provide best-effort delivery only, and are not completely reliable. 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 participating processors. Lansis uses a combined systems of piggybacked *positive-ACKs* and *negative-ACKs* in order to gaurantee delivery of messages to all the processors.

Every processor transmits messages with increasing serial numbers, serving as message-ids (*e.g.* P_A emits 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:

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

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:

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

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. 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. The piggybacked information is user for reconstructing the original 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 [14]. 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. 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 different orders.

Implementation of Services

The multicast 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.
The delivery criteria are as follows:

1. Basic: Immediate delivery.
2. Causal: When all direct predecessors 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 delivery criterion are beyond the scope of this paper.
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 Lansis is described in a separate section below.

Implementation Considerations

Since Lansis 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 then slows down when losses occur. Define a *network sliding window* as consisting of all the received messages that are not acknowldeged by all the processors 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 maintain only their sent-out messages. The network 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 from 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.

Our first, non-optimized prototype of Transis on top of UDP broadcast sockets over a 10 Mbit Ethernet exhibits encouraging performance results. 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. Thus, the performance of Transis for the communication of three or more machines is superior to utilizing point to point protocols for the same purpose. Lansis exhibits only a slow degradation in performance as the number of participating machines increases.

Lansis is a useful tool when used carefully. It is important to remember that it bears a cost in the extraneous communication and interrupts when messages are carried over to non-interested destinations; and in the complexity and space overhead of the trasport layer.

4 The Membership in Lansis

Each processor holds a private view of the *current configuration* that contains all the processors it has established connections with. This view is denoted *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. 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 problem of maintaining processor-set membership in the face of processor faults and joins is described in [9]. As noted by others ([12, 11, 14]), solving the membership problem in an asynchronous environment when faults may be present is impossible. Transis contains a new membership protocol that handles any form of detachment and re-connection of processors, based on causally ordered messages. This extends the membership protocol of Mishra et al. ([16]). Our approach never allows blocking but rarely extracts live (but not active) processors unjustly. 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 ([14]). The dynamic membership enables simple and efficient solutions for the rest of the control services, such as the agreed-multicast.

The Lansis membership protocol achieves the following properties:

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

4.1 Faults Handling

The faults algorithm handles departure of processors from the set of active ones. Recognizing faulty processors is important for two reasons: First, this knowledge is valuable for the upper level applications. Second, it prevents indefinite waiting for responses from failed processors. More specifically, the delivery of Lansis *agreed* and *safe* message might block if the DAG does not contain messages from faulty processors. To prevent a deadlock, the faults should be detected and considered. The object of the algorithm is to achieve consensus among the set of active and connected processors about the failed processors.

The faults algorithm is initiated every time the communication with any processor breaks. Each processor identifies failures separately. A processor that identifies a *communication-break* with another processor emits a special message called an FA message declaring this processor faulty. The specific method for detecting communication-breaks is implementation dependent and irrelevant to the faults algorithm. For example, in the Lansis environment, each processor expects to hear from other processors in its membership set regularly. Failing this, it attempts to contact the suspected failed processor through a channel reserved for this purpose. If this fails too, it decides that this processor is faulty.

The faults algorithm operates within the regular flow of Transis messages. The algorithm relies on Lansis' ability to reliably deliver messages. Since the special *FA* message represent *configuration changes* in the system, they must be delivered in a consistent order at all the processors. The main difficulty is to reach an agreement on the identity of the last messages received from failed processors.

In order to achieve this agreement, the delivery of FA messages is delayed until all the remaining live processors agree on *all* the failures. Neglecting this, there may be scenarios where some processors deliver the last message of a failing processor before the configuration change, while it is delayed on other processors.

4.2 Joining

A full membership algorithm must allow for processors to join the membership set dynamically. Typically, this problem has been solved for the case that a single processor joins an existing membership set (*e.g.* [15, 16, 9, 7]). However, in reality processors might temporarily detach. When they reconnect, they are oblivious to the detachment and continue sending regular messages. Another difficulty arises from partitions and reconnections of the network. In this case, there are two *sets* of processors that need to be joined together. Finally, even without any mishaps, our system starts-up spontaneously. Each processor comes up as a singleton-set, and then two or more merge into larger sets.

Thus, in a practical membership algorithm, there is no joining-side and accepting-side. It must handle the joining of two or more sets of any size. The algorithm must operate correctly in face of faults occurring during the joining process.

The join algorithm is trigerred when a processor detects a "foreign" message in the broadcast domain. Every active membership publishes its current state through an *Attempt Join* (AJ) message.

The join algorithm works in two logical stages:

stage 1 The purpose of the first stage is to collect as many foreign join-attempt (AJ) messages, without committing. A timer is set to bound the execution of this stage. Let us denote J the set of all the collected processors at the end of stage 1.

stage 2 In the second stage, the processor is committed to the join set J. It emits a special JOIN message containing J. The processor is allowed to shift to a new join set only when it is "safe", *i.e.* when one of the required members of J sent a different suggestion, not contained in J. This assures that this member will never concur on J, and the new suggestion can be safely incorporated into J. When all the processors in J agree on it, J is accepted.

Faults occurring during the join procedure are handled in a similar way to the above faults algorithm. However, since not all the joining processors belong to the same membership set, the join procedure must dynamically determine which faults occur in the 'previous' setting, and which occur effectively *after* the join.

The dynamic membership of Lansis allows partitioned operation. Assume there are 50 workstations in the computer science department that execute a distributed application. If the network is partitioned into two halves, such that each half contains exactly 25 workstations, each half will gradually remove all the processors in the other half out of its membership, and continue operation normally. When the network reconnects, the membership algorithm will re-join the partitions, providing the upper level with the **exact point** in the processing when the re-join occurs. It is up to the high level application designer to implement a consistent re-joining of the data and the application.

5 Transis

The *communication domain* (CD) of Transis comprises of a collection of broadcast domains and provides the multicast message passing services throughout it. The connection between different DBs is mediated via the *Xport* mechanism. This mechanism provides a reliable, selective port for transferring messages between Lansis domains. The Xport mechanism allows the Transis services to be supported over arbitrary topologies. It also allows the partitioning of a single LAN into sub-domains that are interconnected through an Xport link.

The reason for partitioning a CD into BDs is that the Lansis protocol might be too demanding on a large environment, because it requires each processor to observe all the messages and maintain mutual backup. Therefore, Transis provides the multicast services over a broader range of systems.

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

- Scalability: In a collection of BDs, the messages overate and space overhead is kept within the smaller sets of BDs and therefore can be kept reasonable.
- Tailoring the services 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.
- Enabling partitioning according to performance 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 logical link between the two domains, which is easier to handle. Of course, this additional breaking is only beneficial if the communication between the domains is infrequent.
- The application structure may suggest partitioning into communication clusters which are best served by different BDs. The application may be best served by a hierarchical communication domain.

Each BD is represented by a logical Xport node (that can be implemented by a single or multiple processors). The Xport nodes may be connected in any arbitrary topology. In order for messages to get outside the local BD, they must pass through the Xport interface.

The inter-BD multicast operations are performed only by the Xport processors. All other processors 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 selectively exports the message to all relevant BDs. The message is sent only once on each Xport connection.
- *Import*: Messages received from remote BDs are disseminated locally on behalf of their senders.

There remains the problem of guaranteeing the order of messages between BDs. To the best of our understanding, this problem requires global information about all the BDs' representative Xport nodes. We can live with this fact and perform causal, agreed and safe ordering protocols among the Xports. Note that causality order can be preserved in the entire CD simply by maintaining causality among the Xports' set.

The agreed and safe multicast within a CD are more complicated. Another approach is to leave this matter for the higher levels. This is currently an active research issue in the Transis project.

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 primitives. The efficient implementation is facilititated through the use of broadcast. 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.* [15, 16, 9, 7]) 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] 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.

[4] 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.

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

[6] 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.

[7] 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'.

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

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

[10] S. E. Deering. Host extensions for ip multicasting. RFC 1112, SRI Network Information Center, August 1989.

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

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

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

[16] 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.

[17] 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.

[18] J. B. Postel. User datagram protocol. RFC 768, SRI Network Information Center, August 1980.

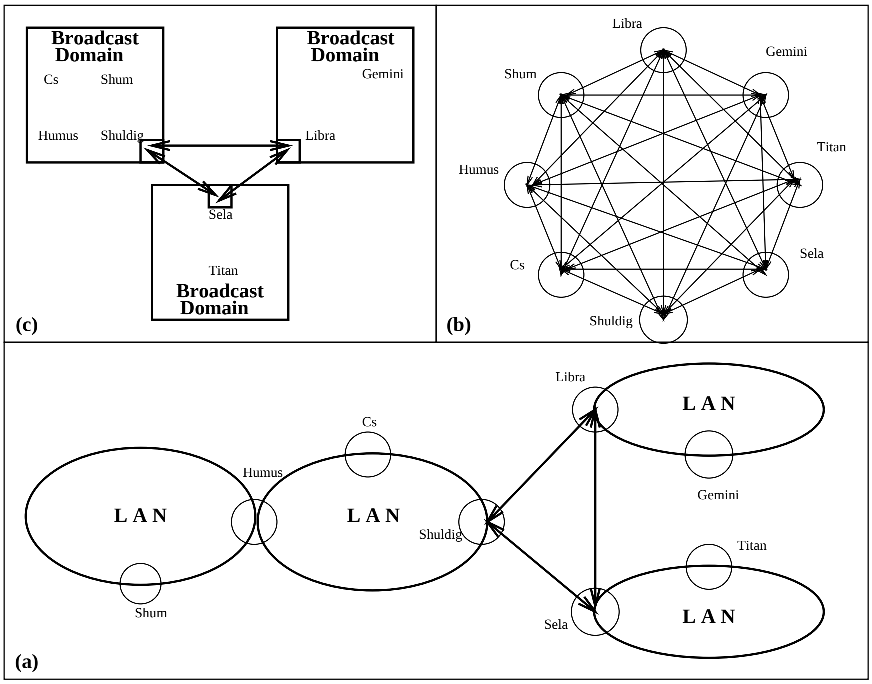

Figure 1: Communication Structure: Reality and Model