The Transis Approach to High Availability Cluster Communication

Dalia Malki, Yair Amir, Danny Dolev, Shlomo Kramer

Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

Technical Report CS94-14

Abstract

This paper presents the design and implementation of the Transis communication subsystem. Is starts with indicating typical classes of fault tolerant applications supported by Transis. It shows the design decisions made in order to provide useful services for these applications. Then, it describes the protocols developed for supporting these services efficiently. The important novel aspects are: *Supporting partitionable operation; employing the hardware multicast for high throughput communication; utilizing a novel network based flow-control mechanism; providing simple and easy to use group semantics.*

1 Introduction

This paper presents the design and implementation of the Transis communication subsystem. Transis supports reliable group communication for high availability applications. Transis contains a novel protocol for reliable message delivery that optimizes the performance for existing network hardware and tolerates network partitioning.

The communication needs of distributed applications have increased considerably in recent years. Today, applications require high speed advanced communication services that support intricate interactions. A growing number of areas require interaction among *process groups*, *i.e.* multiple processes that (may) span multiple machines. To illustrate this point, we name a few areas:

- The known way to increase the availability of an information service is through replication of the information service on a number of machines. Various replicated information services are already in wide spread use, such as replicated file systems (*e.g.* [55, 42, 7, 61, 54, 39, 52]), and replicated databases (*e.g.* [1]). Increasing the availability through replication means that every replica performs all the updates to the information base, and any replica can provide up-to-date snapshots of the information. To this end, the group of replicated servers needs to communicate in order to maintain the consistency of all the replica.

- The recent increase in computing power and in graphics and audio capabilities of every workstation has led to an enormous interest in *multimedia* applications. Currently, much effort is invested into the research and development of tools such as video and audio conferencing, shared editors, interactive full screen discussion tools, and so on. On-line conferencing in any form requires high bandwidth fast communication among the participants.

- Cluster parallel processing is starting to emerge, and the potential in this area is vast. Since the parallel supercomputers of today are internally network computers, the programming environment they support (*e.g.* MPI, CCL, PVM) can be emulated on standard network computing environments. Support for these standards within network clusters can provide a software platform to port intensive numerical applications onto. In some cases, this can provide an attractive alternative for (parallel) supercomputers.

The application programmers of all of the above applications require an advanced tool, that disseminates information reliably and efficiently to multiple destinations. The tool must sustain high communication rates while automatically regulating the communication flow. Such a tool must be optimized for each available hardware, and relieve the programmer from adapting the program to different communication platforms. It needs to specify its guarantees to the programmer accurately, and define the behavior that will occur upon each possible failure. So, the important issues that we focus on in this work, are the semantics of services required by communication-based distributed applications, and the provision of optimized protocols for supporting them.

Sometimes, as we shall see below, the programmer also relies on ordering guarantees provided at the communication level. Ordered services come in addition to reliability, high performance and flow control. Given the latter requirements, we will see that ordering comes at low cost.

Our work on the Transis project started at 1990. At that time, there were already several projects under way in this area: notably, the Isis system [13, 16] from Cornell University, has been in wide spread use since around 1987. Also, Psync [49], Lazy Replication [37], the Trans and Total protocols [43]. Independently and roughly at the same time, two other projects in this area were launched: The Horus project [59, 58] rebuilds Isis from scratch, and the Amoeba operating system project integrates a group communication facility within the kernel of the operating system [34, 32, 33]. It is, therefore, crucial to explain first, why the Transis project was needed. For our convenience, we refer in the sequel to the work that existed prior to the start of the Transis project as the *first generation systems*, and to current projects under development as the *second generation systems*.

The first and foremost problem with all the first generation systems is that if partitions occur in the network, those systems made no guarantees about the situation. Isis, for example, designates one of the components[1] as *primary*, and relieves all consistency strains off the non-primary components. Thus, it is possible for a non-primary component in Isis to continue operation shortly after a partition occurs, and before the partition is detected, and perform operations inconsistently with the primary component. Moreover, if the primary component is lost (as, provably, cannot be prevented [28]), then the entire system blocks. Another approach, taken *e.g.* by the Total protocol, allows the system to continue operation only if a majority of the processors are operational and connected [43].

Our approach to handling partitions is different. We incorporate partitions into the model, and provide the strictest semantics possible in face of partitions. The Transis approach is unique in allowing *partitionable operation* and in its support for consistent *merging* upon recovery. Further work done in the Transis project [35, 5] shows how the understanding of partitionable operation can lead to the development of algorithms that provide long term guarantees despite failures, avoid the end-to-end acknowledgment needs, and come up with algorithms for long term replication that are more fault tolerant than prior work.

Other, second generation systems, consider the possibility of network partitions: In the Amoeba system, a partitioned group may continue operation within multiple components, unless the user specifies otherwise. However, the system provides no means for merging the components upon recovery. The Horus system intends to adopt the Transis approach to partitionable operation, and to collaborate with us on the development of high level mechanisms and methodologies for partition-recovery [41].

The second issue that trigerred the work on Transis is, that the implementation of the

[1]Sometimes in the literature, a partition refers to a disconnected network component. We use the term *partition* to denote the detaching of the network, that causes several *components* to become disconnected.

first generation systems mentioned above fail to utilize the available hardware multicast.[2] Because of this limitation, the first generation systems were built on top of point-to-point communication protocols. This approach is not sufficient, because high communication rates among multiple machines cannot scale using point-to-point links. The need to sustain high bandwidth group communication within local clusters will become crucial in many applications. Moreover, platforms supporting hardware multicast will apparently prevail in future networks.

As an exception, the Trans protocol [43] is a reliable multicast protocol that proposes to use the available hardware multicast. The principles of the Trans protocol underlie the reliable multicast protocol in Transis. We note that the Psync protocol [49] may be used on top of available hardware multicast.

The Transis project is pioneering in the development of high performance multicast protocols that exploit the available multicast hardware. From the outset, we proposed to model the system as composed of a collection of *multicast clusters*, as depicted in Figure 1. Each multicast cluster represents a domain of machines that are capable of communicating via hardware multicast. In reality, such a cluster could be within a single Local Area Network (LAN), or multiple LANs interconnected by transparent gateways or bridges. Other forms of communication are used between interconnected clusters.

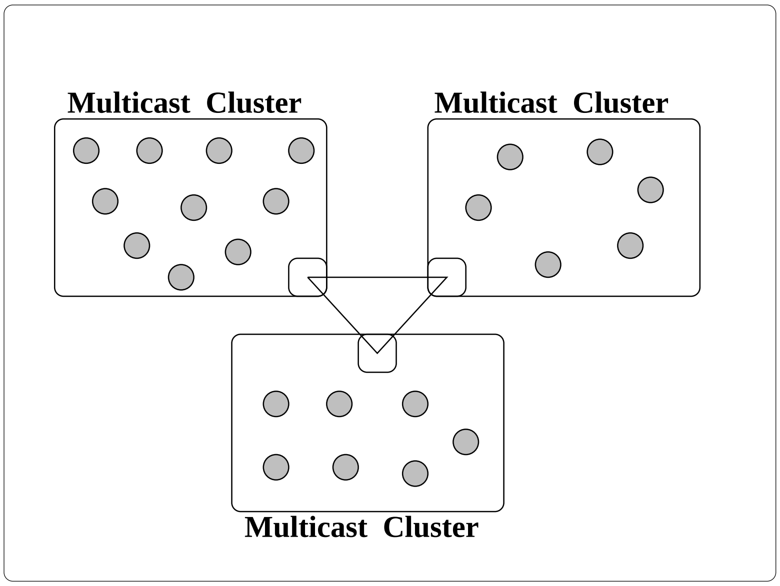

Figure 1: The Transis Communication Model

Today, there are several systems, as well as the new revisions of the Isis system, that also

[2]We use the term multicast for denoting the dissemination of messages to multiple destinations done at the network level, whether to a selected subset or not. Ethernet, for example, supports both broadcast and multicast, and the Deering multicast extensions to IP [25] extend IP to support multicast at the Unix Ethernet drivers' level. Similarly, the emerging ATM standard will support multicast dissemination at the network level.

exploit hardware multicast where available [6, 34, 58, 16]. As shown below, Transis achieves comparably good performance results, due to its highly distributed and efficient protocols.

A crucial issue in the performance of high rate communication is the flow control mechanism. Another advantage of the Transis hierarchical approach is that the network flow is regulated within each multicast cluster on a network-basis, rather than on a point-to-point basis. Transis contains a flow control mechanism that takes into account the network traffic of the entire cluster. Our experience shows that this property is crucial in achieving high communication throughput among a large number of machines in a cluster.

The paper presents the classes of applications supported by Transis. It shows the design decisions made in order to provide useful services for these applications. Then, it describes the protocols developed for supporting these services efficiently.

2 Related Work

There is a great deal of research in the area of fault tolerant distributed systems, both in theory and in practice. In many high availability systems, the service layers of multicast communication and membership maintenance are intertwined together (*e.g.* [46, 47, 16, 13, 33, 24]). Therefore, in this section, we relate our work to several types of efforts:

1. Multicast communication protocols.
2. Membership algorithms.
3. General high availability projects.

Multicast Communication Protocols

The V system [21] was the first to introduce the concept of process-group communication via multicast. V supports group communication as an operating system primitive. It implements *best-effort* multicast semantics, and does not make any guarantees about the delivery or the order of delivery of multicast messages. Similarly, VMTP [19] and the IP-multicast extension [25] support unreliable multicast, but do so in a portable way, for more broad systems.

Chang et al. [18] provide a family of totally ordered multicast protocols using a token revolving within a *token list*. Messages can be broadcast by any process. In order to deliver a message, the sequencing token needs to be transferred L times after its transmission, where L is a system parameter, and L token holders need to acknowledge the message. The protocol guards consistency by blocking the system in case that the token holder and its L followers in the token-list are unreachable.

The Isis system contains a suite of protocols for supporting various orderings in reliable multicast services [15]. In the causally ordered multicast service (CBCAST), the causal order among messages is preserved by piggybacking vector-timestamps onto each message.

The causal ordering of messages is also maintained across overlapping process groups. Isis employs a clever technique for reducing the amount of information sent on messages, in the case that many overlapping groups interact. The totally ordered multicast service in Isis (ABCAST) is supported by a central coordinator scheme, that sets the order of transmitted messages within each process group. The order of multiple messages may be set by a single message from the coordinator. Total ordering is not maintained in Isis across different process groups.

Psync is a multicast communication substrate developed at the University of Arizona [49]. It provides several multicast communication services among a group of processes. It provides the user with a history *context* of messages, and allows the user to define semantical-dependent multicast ordering. Failure notifications are ordered causally with respect to regular messages in the system [45]. The underlying message recovery protocol in Psync may be used on top of available hardware multicast.

The Amoeba system contains support for ordered group multicast within the operating system kernel [34, 33]. The ordered multicast protocol in Amoeba uses a centralized coordinator for distributing the messages and for setting the total order on them. Amoeba lets the user trade resiliency to failures for performance: The user may choose parameters to allow fast message delivery on the one hand, or to require higher message stability before delivery.

The Trans and Total protocols [43] are genuinely distributed protocols for reliable multicast, preserving causal and total order, respectively. The basic mechanism in Transis for causal reliable delivery of messages is based on the Trans protocol, with several important modifications that adapt it for practical use (see [4]). The Total protocol reaches agreement on the total order of messages among a majority of the processes in the systems. The protocol is prone to deadlock in rare situations (see proof in [36]).

The Xamp protocol of [60] relies on special network hardware that orders messages. The guarantee of message delivery in Xamp is based on a 2-phase acknowledgment protocol.

An early delivery distributed protocol for totally-ordering multicast messages appears in [26], and is employed in Transis. This protocol is guaranteed to terminate when no failures occur. In case of failures, its operation on top of the Transis membership service guarantees progress.

Another totally ordered protocol is described in [6], and was also implemented within the Transis environment. Their protocol uses a *revolving token holder* that coordinates the transmission and regulates the flow of broadcast messages. One of the drawbacks of the latter method is that the system has to reconfigure whenever the token is omitted.

For an excellent formal study of the reliable broadcast problem, we refer the reader to [29].

Membership Algorithms

The maintenance of the membership of machines is a basic building block of many fault-tolerant applications.

Cristian introduces the problem of processor membership maintenance in [23]. This work formulates the problem in a synchronous environment, and provides protocols for solving it, assuming that the clocks of different processors are synchronized within some known time skew, that the relative speeds of processors are known, and that message transmission times are bounded. In this environment, Cristian places bounds on the time required for reaching agreement on the process group membership; he requires that processors requesting to *join* the configuration succeed in doing so within limited time; and that processor failures are detected, and consequently faulty processors are removed from the configuration, within a known time bound.

To the best of our knowledge, the first formal definition of the requirements of membership in asynchronous environments is given in [51]. Their paper defines a *primary-component* membership service, that maintains the *local views* of all the operational machines in agreement. In the application requirement in [51], at most one component may exist, and machines outside the primary component are either dead or give up. The implementation of this membership protocol within the Isis system guarantees another important principle, called *virtual synchrony*, that ensures that all the processes deliver regular messages in the same order with respect to membership changes [10]. In this way, membership changes induce a consistent context on regular messages in all the processors (see also [14]).

Moser et al. describe another primary-component membership service [44], that is based on a reliable, totally ordered communication service [43]. The underlying total ordering protocol guarantees agreement on the order of messages throughout the system, and induces agreement on process joins and removals by placing *join* and *remove* messages in the stream of totally ordered messages. Since the membership agreement relies on the total ordering layer, it suffers the same limitations exhibited by it: A majority of the processors need to be operational and connected in order to reach agreement on the total order, and there is a small chance that agreement cannot be reached forever even when no failures occur. On the other hand, the advantage of this approach is that membership changes are consistently ordered with respect to all regular messages.

The Psync system possesses a membership mechanism based on causally ordered messages [45]. This membership protocol preserves causal-order between membership changes and regular messages, but does not guarantee virtual synchrony.

Jahanian et al. [31] provide a suite of membership protocols, unrelated to multicast message ordering. In the Weak Membership of [31], there is no guarantee that all the members see a certain membership installation. This protocol simply assures that if the communication is timely and there are no faults, then the membership will be in consensus. In periods of instability, the weak membership view might diverge at different processes. For purposes such as object replication, this membership is too weak. Therefore, [31] also provides the Strong and the Hybrid memberships for consistent replication.

Two membership protocols that are closely related to our work are [3, 6]. In [3], we have introduced the concept of *partitionable* membership service, and extended the definition of virtual synchrony to such a service. [6] utilizes the principles of this membership protocol, and extends it to form a ring of machines with an initial token holder.

There are other membership services that allow partitioned operation. The membership service of the Amoeba system [32] lets the user determine the minimal size with which the system can continue operating. If the user determines a majority threshold, the result is a primary-component membership service. On lower thresholds, the system may partition. The user has the control in trading resiliency for consistency. The protocol presented in [32] does not provide any solution for merging operational components upon reconnection.

High Availability Projects

Borg et al. describe a fault tolerant network Unix system [17]. Fault tolerance is achieved through duplicating all the processes in the system, and by routing their external communication through a special multicast communication layer. In this way, every process' external interaction, whether to other processes, or to system services such as DISK I/O, is recorded at its replica and may be recovered in case of failure. The system relies on special network hardware and on a network protocol implemented at the operating system's kernel.

Isis is a tool that was especially designed to support fault tolerant distributed applications on standard distributed platforms. Isis is widely used in industrial and in academic settings. The Isis package supports a variety of multicast communication services, and provides the application programmer with the *virtually synchronous* model of programming [10, 14]. Isis mandates the primary-component consistency model.

The Gossip project at MIT [40, 37] is based on a novel approach called *lazy replication*. It is suited for replicating a single service, and lets the client of the service specify the potential causal dependency among requests. In this way, unrelated requests of different clients do not incur any latency delay due to communication, and the requests propagate "lazily" (off-line) to the replica. This approach can be very beneficial in a restricted type of applications.

The AAS system is a fault tolerant application, developed by IBM Corp for the Federal Aviation Administration to support the air traffic and control system [24]. The system is intended to overcome both hardware and software errors that manifest themselves as *performance* errors. The basic mechanism for increasing the fault tolerance of the system is through server group replication on independent redundant hardware. Server replication is supported by underlying synchronous services, such as group communication, membership maintenance and clock synchronization.

The Horus project, currently under development at Cornell University [59, 58], undertakes to rewrite Isis from Scratch. Horus is a modular system that employs micro-kernel programming methodologies and allows great flexibility in services and semantics. The bottom layer of Horus, called MUTS, supports multicast communication on top of a variety of software and hardware platforms. Where available, it uses hardware multicast for disseminating messages to multiple destinations. Horus implements a clever packing technique that sends multiple messages within one communication packet, and thus achieves extremely high message throughput for short messages. Horus has adopted the principles invented by Transis for supporting partitionable operation.

The Delta-4 project [50] provides services for constructing dependable software. The project has two tracks: one providing services for active replication in an *open system architecture*. The open system architecture of Delta-4 allows incorporating standard workstations, but requires special purpose network interfaces. The second track provides real-time services for semi-active replication, within an *extra performance architecture*. In the extra performance architecture, special homogenous machines are utilized, as well as special purpose network hardware.

The Consul project of the University of Arizona [47, 46] supports object replication. The system is implemented in a mudular object oriented approach, on top of the X kernel. It employs the Psync communication substrate for multicast communication, and thus allows the user to define semantically dependent multicast orderings. The system supports automatic logging of messages and replaying on recovery. It guarantees that when the communication is stable, all the processes in the system converge to have consistent views of the system.

3 Matching the Application Needs

We believe that communication is going to play such an important role (and already is) that no **one** protocol will provide an answer for all needs. On the contrary: performance will be pushed to the limits in such a way, that only a suite of protocols, each one optimizing for its needs, will sustain the communication needs of all different applications.

In this section, we attempt to provide the reader with the motivations and guidelines we followed when designing the Transis services. We justify the Transis design "top down", *i.e.* starting from a class of distributed applications, we show services that can benefit the application developer.

3.1 Application Classes

Active replication using the State Machine approach [53], is widely used in replicated information systems (*e.g.* Deceit [55], Afic [1]). In this method, the information base itself resides on multiple sites. The operations that modify the information (updates) are introduced to the sites in a consistent, global order. This is done by passing all the updates through a special layer, a *reliable and totally-ordered communication layer*, that passes messages reliably and in the same order to all the sites. Each site can perform any delivered update independently. In this approach, requests for retrieving information from the information base (reads) can be made to each replica separately. In this way, any available replica can service read requests, and thus the availability of the service is increased, as well as its overall throughput.

Another class encompassing a wide range of applications is one we call *a replicated display*. In this application class, a display of some sort is set up in multiple locations to depict the current state of the system. Each display allows either human examination or computerized analysis or both. Examples of this kind are military command and control displays, and radar control systems. The characteristic of this type of applications is that the most up to date state of each object needs to be known at each replica, but among the updates made to a specific object, only the *last* one is significant. The replica exchange messages containing snapshots of objects (but not of the entire system). To some extent, a similar property is characteristic of on-line conferencing applications. There is no ordering requirement on messages, because previous updates to any specific object are simply *canceled* by later ones.

The third application type we list includes numerical programs that can be "parallelized" to run on a network of machines. Experience in parallel processing has shown that often, such applications require information exchange among groups of processes, that entail all-to-all exchange, one-to-all or all-to-one scatter-gather operations, and so on (see PVM, MPI). In such group operations, the messages must be disseminated reliably and efficiently to all the destinations, overcoming any communication failures.

3.2 Reliable Group Communication Services

The construction of such applications can be made substantially easier with a strong communication substrate. The communication substrate can be made to work on new networks, making the porting of the applications easy. The optimizations required for sustaining high throughput communication are made by the architects of the communication layer, and allow for specialized optimizations, that any application programmer can benefit from them.

We now address the question: What are the properties required of such a communication substrate?

The State Machine replication approach, message passing packages for parallel processing, and a wide class of other applications, require that the same message is delivered to a set of destinations *reliably*. In point to point communication, the definition of reliability is obvious: If both parties stay operational, then a message from the first party will eventually reach the second party. In the context of group communication, there may be several possible interpretations to the reliability requirement. Having identified the kind of applications that we want to support, we choose an *all or none* guarantee: if any member of the destination set delivers a message, then all the other members of the set will deliver it, despite some pre-defined failures (but not despite all failures).

Often, the all-or-none guarantee is misunderstood, or oversimplified: the fact is that, in an asynchronous environment, no protocol can guarantee true all-or-none semantics despite arbitrary failures without saving messages on stable storage, because, otherwise, the act of message-delivery would require common-knowledge, which is impossible to attain in an asynchronous environment [30]. In Transis, we therefore provide two levels of service:

Atomic: An Atomic message delivered at any member is guaranteed to be delivered at all the currently operational members of the application, despite message omissions. However, if any member crashes or disconnects soon after the atomic message is transmitted or delivered, then it might not deliver the message.

Safe: A Safe message delivered at any member is guaranteed to be delivered at all the currently operational members of the application, provided that they do not crash. Note that, even if they detach (from the originator, or from any member that is known to deliver the message), the message is guaranteed to be delivered to them.

Note how these two services are natural to use in the applications we described above: The Atomic service would be used for parallel processing applications and in the distributed display application, and would avoid the delay incurred by making the messages Safe. Within each connected component, the application will benefit from the advanced communication substrate because it provides efficient and optimized dissemination of messages. In addition, in the distributed display application, recovery of disconnected or failed members is made easier by the reliability guarantees made within each component: They allow each component to act as a unit upon recovery, and optimize state-merging considerably.

The Safe service, on the other hand, would be used in a replicated information service based on the State Machine approach. Each server of the replicated service acts upon an update message after it is Safely delivered, thus guaranteeing that even if the system partitions, the update will be carried out by any operational replica (at the same order). Using this service, end-to-end acknowledgment is needed only upon a membership change event such as processor crash, processor recovery, network partition and/or network merge (details on this can be found in [5]).

At first glance, it appears that a reliable (and moreover ordered, as we see below) multicast service would be an "overkill" for the distributed display application, because it needs the most recent snapshot of each object, and not the entire history of changes to the object. So let us examine the application needs in detail: This application requires that whenever communication is possible among two displays for a sufficient duration, every object has the most up-to-date state among the two. Therefore, a mechanism that guarantees the delivery of the "last update" to each object is required. If the "last update" to an object is not followed immediately by another update to the object, then it is desirable to detect the loss of this update as soon as possible. Therefore, some means for recovering lost messages is desirable even in this case, provided that the cost is not too high. Inter-message relations, such as total order or other orders, may speed up the detection of a lost message by the lower layers, in the case that the source of the message does not send additional messages for a while. Thus, message ordering at the low layers may assist in converging the system quickly to a consistent state.

Moreover, the loss rate exhibited by today's networks is extremely low. Therefore, the overhead in recovering lost messages is not high. On the other hand, the message loss rate may increase significantly under heavy network loads. Therefore, in order to prevent excessive message loss, the system must regulate the flow of messages in the entire network, and coordinate the transmission of messages by different processors.

Finally, the distributed display application needs to be reported about failures.

We will see in the design of Transis below, that in order to implement a flow-controlled communication layer, with early omission-detection and failure notification, one already needs a reliable multicast layer. Thus, the fact that the distributed display application does not utilize the message ordering services, does not imply that removing them optimizes the implementation in any way.

3.3 Multicast Message Ordering

Some applications, notably, applications based on the State Machine approach, require that all the members of a group deliver the messages in total order. For example, when messages are totally ordered, the database replication layer can perform updates locally as soon as they are delivered from the communication layer, without a need for further coordination. Transis provides the following ordered multicast service:

Agreed: The Agreed multicast service guarantees that any two messages m, m' delivered to multiple destinations, will be delivered in the same order everywhere. This means that either m is delivered before m' or vice versa, but there is no mixing of their orders at different sites.

Agreed messages in Transis maintain this requirement even when partitions occur. Thus, in case detached components continue operating shortly after a partition occurs, consistency is preserved. Lacking this guarantee, the replicated information service, for instance, would have to wait for an additional round of acknowledgments after message delivery, before it could operate on it.

For applications that do not utilize totally ordered multicast, we thought it useful to provide a weaker form of ordering:

Causal: A causally ordered multicast service preserves the *causal* order of messages (see [38]). Intuitively, causal communication guarantees that a response to a certain message will never be delivered before the message.

Causal-preserving delivery order makes programming somewhat easier, much in the same way that FIFO guarantees make it easier to program two-party interaction. There is currently a vivid debate in the community about the merit of causally ordered multicast. Cheriton & Skeen [20] hold the position that it is entirely useless. On the other hand, several members of the Isis group [9, 22, 57, 56] quite strongly advocate for causally ordered multicast services, and point to several applications that require and use it. It is certainly beyond the scope of this presentation to enter this controversy. However, we find that:

1. In our protocols, causal ordering comes "for free", as a by-product of efficient and reliable message delivery. We find that in order to detect message losses promptly, to avoid multiple acknowledgments, and to maintain the information about message-stability efficiently, we need to construct a representation of (at least) causal order of messages at the lower layers. Furthermore, we delay the delivery of messages when causal "holes" are created in order to make the acknowledgment mechanism concise.

2. We have built several novel protocols, incorporated within the core of the Transis system, that rely on causal ordering of messages [26, 3, 27]. These protocols utilize the causal structure for performing a completely distributed reasoning about the system

state. Without causal structure, similar protocols use a centralized source of agreement (*e.g.* [51]), and do not use knowledge of the history. As we have found the causal structure useful for internal system protocols, we conjecture that future developers of higher level protocols can likewise benefit from causally ordered message passing for developing sophisticated distributed algorithms.

3. Causal order seems to be the intuitive extension of FIFO order to group communication. If all the interaction in the application is done through the communication system and there are no external communication channels, then a causal communication layer is sufficient for preserving any potential causality at the application level. Furthermore, weird scenarios might occur if it is not preserved.

3.4 Failure Notification

Suppose that machines may crash, and the network may partition. In networking environments, it is currently impossible to distinguish for sure between machine crashes and network partitions; thus, machines may appear crashed, but in fact they are only detached.

In order to preserve consistency, a replicated information service must allow only one active component in the network to perform updates. Therefore, the members of the replicated service must act upon messages only if they are delivered within the context of a *primary component*. Therefore, the communication substrate needs to provide failure notification to the upper layer, informing it when machines in the configuration appear detached. Furthermore, in order to avoid replaying the entire history of update messages upon recovery, the failure notification mechanism should indicate when, in the stream of messages, the process group has detached, and when the partition has recovered. This principle is guaranteed by the *virtual synchrony* programming model, and its extension to partitionable operation, see below (Section 4.1). The works of Keidar [35] and of Amir [2, 5] show how this principle can lead to an efficient merging-mechanism, that supports long term consistent replication.

3.5 Caveat

We have manifested at the beginning of this section that communication protocols need to be tailored to various classes of application needs. We reiterate here, that, for example, if one has in mind an application that requires the most up-to-date message from each source, thereby making previous messages from the source obsolete, then one should not seek a reliable communication protocol, nor, for that matter, one that provides FIFO ordering.

Furthermore, even in applications that can make use of the fast and reliable message delivery of Transis, coupled with the consistent failure notification it provides, the application programmer may still need additional layers implementing, for example, end-to-end acknowledgment or disk logging. Transis is useful in providing the communication needs of these applications; the works of Amir and Keidar [2, 5, 35] show that the usage of Transis leads to simpler and more efficient design of sample applications. We have confidence that in the future we shall see more works detailing the utilization of communication substrates such as Transis in the construction of high availability projects.

Therefore, we do not see in the limitations detailed in [20] a refutation of our approach.

4 The Transis Programming Model

The Transis communication system supports *process group* communication. Groups are conveniently identified by a name (a string) that is selected by the user, such that messages are addressed to the entire group by specifying the group name. Using the group abstraction, the communication subsystem relieves the user from identifying the targets of messages explicitly, and from finding the network routes to them. In addition, it guarantees an *all-or-none* delivery semantics among the currently operational and connected members of a group, and handles message losses and transient network failures transparently.

Process groups are the basic unit of communication in Transis. A process can send a message to a group, and all the currently connected members of the group are guaranteed to receive it. In order to send a message to a single process, one must send a message to a group consisting solely of this process.

Process groups are dynamic: members can leave a group voluntarily, or may be forced to leave a group if they crash or disconnect from it. If the network *partitions*, then multiple components of a group may be formed, each component detecting the other components as presumed failed, or detached. New processes may join a group, and *components* of a group may suddenly become connected and merge. All of these occurrences are reflected to the Transis application through a *group membership* mechanism. The group membership mechanism reports about changes in the group membership to the application. It guarantees that the changes are reported *consistently* to all the members of the group. It supports the concepts of the *virtual synchrony* programming model, while extending it to partitionable operation. Conceptually, this model guarantees that the order of membership-changes is consistent with respect to the delivery of regular messages.

Transis provides several forms of group multicast operations: un-ordered, *Causally* ordered, *Totally* ordered, and *Safely* delivered. The multicast operations differ in their semantics and in their cost (*e.g.* latency). Generally, unordered and Causal communication are the cheapest and fastest. Totally ordered communication incurs a higher delay, and is useful for supporting replicated information of any kind. The Safe multicast guarantees that delivery of a message occurs after all the machines in the network have received a copy of the message. It incurs the largest delay, because it explicitly waits for acknowledgment from all the participating machines. Here is the precise definition of the multicast services:

Atomic Guarantees that all the members of the group will receive the message, unless they fail or disconnect. All the services below are Atomic.

Causal Let us denote by $\text{deliver}_p(m)$ the event in which processor p delivers the message m, and $\text{send}_p(m)$ the event in which p sends a message m. The causal order on messages is defined as the reflexive, transitive closure of (see [38]):

(1) $m \xrightarrow{cause} m'$ if $\text{deliver}_q(m) \rightarrow \text{send}_q(m')$ [3]

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

The Causal multicast service guarantees that the message delivery order of Causal messages preserves the causal order. All the services below guarantee the causal order.

Agreed Let m, m' be two Agreed messages, and p, q be processors that deliver them. The Agreed multicast service guarantees that they deliver them in the same order, *i.e.* :

$$\text{deliver}_p(m) \rightarrow \text{deliver}_p(m') \Rightarrow \text{deliver}_q(m) \rightarrow \text{deliver}_q(m') \ .$$

The Safe service below also maintains the Agreed order.

Safe The Safe multicast service guarantees that a message is delivered only after it is received by the Transis software layer, at every processor in the system. This guarantees that the Transis applications at each processor will deliver this message, unless they crash. The Safe messages are ordered in Agreed order with all other messages of any type.

4.1 Virtual Synchrony and Partitionable Operation

Transis provides the application programmer with a programming environment that is conceptually *Virtually Synchronous*, as defined by Ken Birman et al. in the early work on the Isis system [10], and extended into partitionable environments in [48]. The virtual synchrony model and its extension to partitionable operation encompasses the relation between message passing operations in a process-group, and between control messages provided by the system about process failures and joins in the group.

A process group in Transis is dynamic: A group is created with some initial membership, and subsequently its membership changes as processes join (are added) and leave (are deleted or fail). Whenever the membership of a group changes (and initially, created), all the processes of the new membership observe a *membership change* event. This event is provided as a special Transis message, a *membership change message*.

The essence of the virtual synchrony programming model, is in guaranteeing that membership changes are observed in the same order by all the members of a group. Furthermore, membership changes are totally ordered with respect to all the regular messages in the system. This means that every two processes that observe the same two consecutive membership changes, receive the **same** set of multicast messages between the membership changes. For example, let's consider a group that changes from the configuration $\{A, B, C\}$ to $\{A, B, D\}$. Then processes A and B will first receive the membership-change indication of $\{A, B, C\}$,

[3]Note that '$\rightarrow$' orders events occurring at q sequentially, and therefore the order between them is well defined.

then they both receive exactly the same set of regular messages, and finally they receive the second configuration change, $\{A, B, D\}$. (In this case, C might receive, after the first configuration change, any subset or superset of the regular messages that A and B received). In this sense, membership changes are virtually synchronous, as the processes have identical contexts when messages arrive. This allows the processes to act upon the messages they receive in a consistent way. For a formal definition of the virtual synchrony model, refer to [10].

As an important extension to the Isis virtual synchrony model, Transis allows *partitionable operation*: If a group partitions into two components, such that communication between the components is impossible, then each component continues observing the virtual synchrony model separately. Furthermore, upon re-merging, the merged set will be virtually synchronized starting with the membership change event that denotes the joining. More details on the semantics of the partitionable membership are given below, in Section 7.

5 A High Performance Reliable Multicast Protocol

We experimented with several protocols and ideas for supporting reliable multicast communication. We found out that the way to support reliability with high performance is based on several principles:

- In systems that exhibit low loss rate, it is preferable to use a negative-acknowledgment based protocol. Thus:
  - Messages are not retransmitted unless explicitly asked to, through a negative acknowledgment.
  - Positive acknowledgment, required for determining the stability of messages, are piggybacked on regular messages that go to the required destination. In case no regular message is transmitted, then periodically, an empty message containing only acknowledgment and an "I am alive" indication will go out.

  These ideas are not new, and are utilized also, in various forms, in [49, 43, 34, 8]. Their importance is great in today's networks, that exhibit extremely low message loss rates.

- Detection of message losses must be made as soon as possible. Suppose that machines A, B and C send successive messages. If machine D maintains reliability guarantees against each machine separately, and misses the message from A, then it will not detect the loss until A transmits another message. If, on the other hand, there are additional relationships between messages sent by different processors, then D can possibly detect the loss as soon as B transmits its message. Early detection saves on buffer space by allowing prompt garbage collection, regulates the flow better, and prevents cascading losses.

In the Transis project, this principle led to two separate versions of the system, one relying on causal relationships among messages, and the other on a revolving token that forms a total order on message transmission events. In this paper, we elaborate on the first method (details about the token-based protocol can be found in [6]).

- Our protocols rely on very low message loss rates of the networks. However, under high communication loads, the networks and the underlying protocols can be driven to high loss rates. For example, we conducted experiments using UDP/IP communication, between Sun Sparc machines, interconnected by 10-Mbit Ethernet; Under normal load, the loss rate is approximately 0.1%, but under extreme conditions, the loss rate went up to 30%. Such loss rates would make the recovery from message losses costly. Furthermore, this can cause an avalanche effect, where under high loads, the reliable communication protocols further increase the load to overcome omissions. Therefore, to prevent this situation, it is crucial to control the flow of messages in the network.

- In order to achieve extremely high throughput (approaching the physical limits of the network), the pipelining principle needs to be exploited: One or more machines can "feed" messages to the network in a rate that approximates its maximum capacity. But if the machines delay (*e.g.* for acknowledgment) between successive transmissions, then they fail to utilize the full network capacity.

Generally, we note that, optimizations done at the low levels of the system usually (not always) make the system much faster than it would be, had the entire role been that of application developers. Thus, even though the system (as really, any service) may provide *more* than what is needed by some specific application, it may well still lead to an overall better implementation of the useful parts of the service. In addition, it employs system-wide flow control considerations, instead of being limited to a per-application basis.

5.1 Details of the Message Recovery Engine

Transis handles message dissemination and recovery at a per-machine basis. Within each machine, a Transis *multiplexor* supports multiple clients (see Figure 2). The Transis multiplexor can be lowered down to the operating system's kernel, in systems that allow dynamic loading of modules (*e.g.* Solaris, MACH, X-kernel). In the current implementation, the Transis multiplexor is implemented as a user level process that remains operational at all times (a *daemon* process).

The principle idea of reliable message delivery within a multicast cluster in Transis is motivated by the Trans algorithm [43] and resembles also the Psync algorithm [49].

Messages are transmitted via a single transmission, using the available network multicast. The "blobs" in Figure 3 represent multicast messages. Each machine tags its messages with increasing serial numbers, serving as message-ids. For example, in the figure, machine

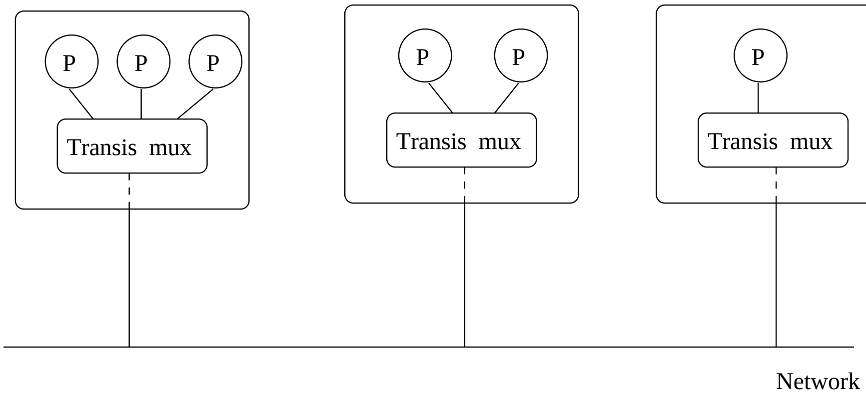

Figure 2: The Transis Multiplexor

A emits the first message A_1, machine B emits its first message B_1, and so on. Acknowledgments to messages are piggybacked onto the next multicast messages. The full arrows represent acknowledgments: from message B_2 to A_1[4] and to B_1, from C_1 to B_2, etc. An ACK consists of the sending machine-id and the serial number of the acknowledged message. Thus, the message B_2 contains the acknowledgments $\stackrel{A_1}{\hookrightarrow}\stackrel{B_1}{\hookrightarrow}$. A fundamental principle of the protocol is that each ACK needs only be sent once. Further messages, that follow from other machines, form a "chain" of ACKs, which implicitly acknowledge former messages in the chain. For example, Figure 3 could depict the following scenario on the network:

$$A_1\ ,\quad B_1\ ,\quad \stackrel{A_1}{\hookrightarrow}\stackrel{B_1}{\hookrightarrow} B_2\ ,\quad \stackrel{B_2}{\hookrightarrow} C_1\ ,\quad \stackrel{C_1}{\hookrightarrow} D_1\ ,\quad \ldots$$

Machines in a multicast cluster might experience message losses. They can recognize it by analyzing the received message chains. For example, machine A recognized that it lost C_1 after receiving the sequence: A_1, B_1, $\stackrel{A_1}{\hookrightarrow}\stackrel{B_1}{\hookrightarrow} B_2$, $\stackrel{C_1}{\hookrightarrow} D_1$. Therefore, A emits a *negative-ACK* on message C_1, requesting for its retransmission. In this case A acknowledges B_2 and not D_1, since messages that follow "causal holes" are not incorporated for delivery until the lost messages are recovered. In this way, the acknowledgments form the causal relation among messages directly.

The delivered messages are held for backup by all the receiving machines. In this way, retransmission requests can be honored by any one of the participants. Of course, messages cannot be kept for retransmission forever. When all the machines have acknowledged the reception of a message, it can be safely discarded.

If a multicast cluster runs without losses, then it determines a single total order of the messages. Since there are message losses, and machines receive retransmitted messages, the original total order is lost. The piggybacked acknowledgments are used for reconstructing the original partial order of the messages.

[4]Note, that the direction of arrows is consistent with the causal order determined by acknolwedgements; thus, the arrow from A_1 to B_2 represenets the acknowledgment from B_2 to A_1.

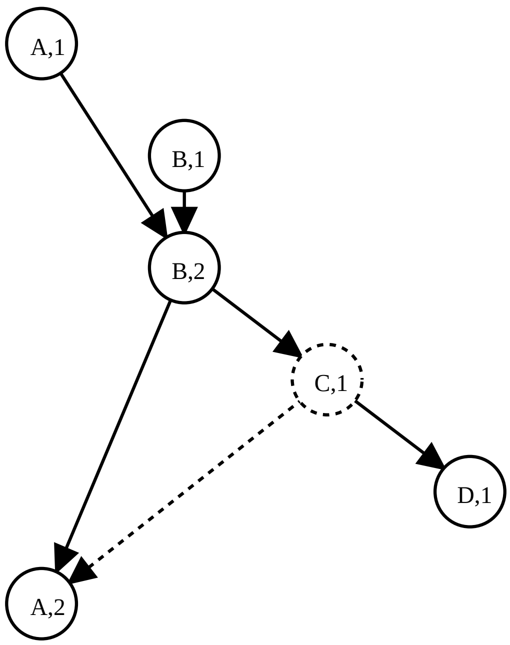

Figure 3: A Transis Scenario

5.2 Flow Control

Transis employs a novel method for controlling the flow of messages and for bounding the amount of memory consumed by the protocol. Define a *network sliding window* as consisting of all the received messages that are not acknowledged by all the machines yet. Each machine computes this window from its local information. Note that this window contains messages from *all* the machines in the current Transis configuration, unlike the traditional sliding-window that maintains only the machine's own 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 anything but "I am alive" messages) when the window exceeds a maximal size. The window cannot remain stuck for long, because the background membership algorithm will remove from the configuration machines that do not participate (see the Membership Section below). This releases the sliding-window block and the flow of messages resumes. The network sliding window roughly determines the maximum number of messages that need to be kept for retransmission.

5.3 Agreed Multicast

One of the characteristics of the Trans and the Transis protocols, is that they allow completely spontaneous transmission of messages by any machine. Consequently, two machines may send messages within a small interval apart, none receiving each other's message first. In this case, there will be no acknowledgment between these messages. This means that additional processing is required if there is a requirement to deliver the messages in the same total order at all their common destinations.

The Agreed multicast service guarantees that messages arrive reliably and in the same total-order to all their destinations. Since we currently have three versions of Transis that differ in their implementation of the Agreed multicast service [26, 6], we chose to present in the section the tradeoffs in this issue.

There are several completely distributed algorithms that build a total order from the local information and reach agreement [43, 26, 49]. It is perhaps easiest to understand the *all-ack* algorithm of Transis, that is also completely distributed. The above referred algorithms are essentially optimizations on this principle. The all-ack idea is:

- Wait until at least one message is received from each machine.
- Then go through the machines in ascending order, and deliver the first message from each machine unless it directly acknowledges another message.

The common characteristic of these algorithms, is that they do not incur any extra message exchange for achieving agreement on the total order. They have *post-transmission* delay, from the time a message is transmitted and received until it is ordered in the right place. Interestingly, this cost is most apparent when the system is relatively idle, and waiting for responses from all (or some) of the machines incurs the worst-case delay. On the other hand, these methods can sustain steady transmission loads that are close to the network limits, when all the machines are fairly uniformly active.

A different family of protocols orders the messages in a total order by contending for an ordering capability to order messages [15, 6, 34]. The Isis ABCAST protocol [15] employs a *token-holder* within each group of communicating processes. ABCAST messages are multicast at will, and their delivery is delayed by all the receiving processes except for the token holder. Periodically, the token holder sends a message indicating its order of delivery for all received ABCAST messages, and all the other processes comply with it. The Token may also migrate to the sender.

The Amoeba system contains a different variation of this scheme, implemented within the operating system kernel [34]. A sequencer kernel is designated as the central controller. Every message is sent to it via point to point communication, and the sequencer multicasts it to all the machines. The FIFO order of sequencer-transmissions determines a total order for all the messages.

The Totem protocol [6] uses a revolving token that holds a sequence-number for messages. The holder of the token can emit one or more multicast messages, and update the token sequence accordingly. In order to transmit a multicast message, a processor must obtain the token. The token itself regularly revolves among all the processors.

The cost in these protocols is in obtaining access to the ordering capability, be it migratable or static. This cost is apparent both in the delay occurring until the control is obtained, and in extra messages exchanged. Once it is obtained, transmission and ordering is done immediately. Therefore, we say that they have a *pre-transmission* delay. The advantage of a control scheme like the revolving token of [6] is that it regulates the flow of messages efficiently.

It is not entirely clear what are the tradeoffs between pre-transmission ordering and post-transmission ordering in these protocols. In particular, the behavior of these protocols when the communication pattern is "chaotic" is an active area for future research.

5.4 Group Semantics

There are several important issues in the semantics of groups in Transis:

- Each multicast cluster constitutes a single communication group at the lower layer of the communication substrate. The group members are the participating *machines*, and reliable message delivery is maintained among *all* the participating machines. The *process* groups visible to the Transis programmer are implemented by a process-group maintenance layer on top of Transis. This layer translates *machine* failure and recovery notifications into *process* failure and recovery notifications, within every relevant process group. In addition, it supports voluntary process join and leave operations, and reports these changes to the group members.

  By this design, we achieve the following desirable properties:

  1. Machine failures and recoveries, and network partitions and re-merges are handled once by the machine-level membership protocol. The process-group maintenance layer easily computes the induced process-group changes in a completely distributed way.
  2. Process-failures and recoveries are detected within their host machine, and do not require extraneous communication, *e.g.* for keeping a periodical "live" signal. A periodical "live" signal is maintained only among the machines participating in the application.
  3. The reliable message delivery engine benefits from the increased redundancy of messages, since messages can be recovered by any participating machine (even if there are no processes interested in these messages within that machine). Furthermore, the piggybacking acknowledgment mechanism can utilize any Transis message going in the "right direction", and save on overall network traffic.
  4. The flow control mechanism regulates the flow in the entire multicast cluster (as far as message traffic is related to Transis). This automatically coordinates the flow of all the communication groups, whether of one or more applications.
  5. Ordering guarantees are made globally within the multicast cluster. This means that inter-group ordering is automatically maintained, and is done once only. Admittedly, in some applications, global ordering might incur a higher price than per-group ordering, especially for Agreed multicast. However, our experience shows

that Transis currently maintains extremely high communication rates, which are close to the network limits, and therefore we believe that this cost is bearable.

The alternative to this approach would be to maintain the membership of operational processes *per process group*, and likewise maintain message ordering and recovery within each process group. This means that every machine failure would have to be detected by each process group separately and would incur a membership-agreement procedure within each process group. When there are many process groups, the price would be enormous. Additionally, each process group would incur its own communication overhead for message acknowledgment and for maintaining liveness, whereas in our approach this cost is amortized over the whole multicast cluster. Finally, applications often need to maintain inter-group message ordering, which complicates the implementation in the per-group ordering scheme.

Truly, there cannot be a clear cut for choosing one way or another, and the tradeoff depends at large on the type of applications required. Having identified the classes of applications that we plan to support, the approach we chose is appropriate. Note that it is possible to simply run several logical Transis multicast domains within one multicast cluster, in case the application benefits from this. On the other hand, in systems like Horus, it is just as easy to implement a *light-weight-group* emulation layer, such as ours, and let the programmer designate the "real" group used by the lower layer.

- Groups in Transis are *open*, meaning that any process may send messages to the group, regardless of whether it is a member of the group or not. The reason for the open group semantics is that external communication into a group is highly common: First, a typical usage of a reliable multicast substrate would be to let *clients* communicate with a group of *servers*, in such a way that the composition of the servers-group is transparent to the clients. Another common usage for group communication is in building fault tolerant distributed applications, in which every object is maintained by a group of processes, and the communication among objects is replicated among their representative groups (see for example [17]). Other systems have also realized the importance of an open group service, and provide an emulation layer for specific needs of open groups: For example, Isis provides a *coordinator-cohorts* mechanism, for client-server interaction [11].

- Groups are automatically created by the first member that *joins* them. Thus, there is no special "create group" operation, and there is no member of a group that is distinguished in any way. This simplifies programming greatly; it avoids the need to check (atomically) whether a group exists already, in order to determine whether to create or to join it, and allows each member to start up by joining its required group(s).

6 Performance Results

We conducted several performance tests and measured the maximal message-throughput per second transferred by Transis in these tests. All the tests were performed on a cluster of Silicon Graphics Indigo machines, interconnected by a 10-Mbit Ethernet. Tests were performed multiple times, and the best throughput results were recorded.

Figure 4 shows how message-throughput varies with message sizes. The graph shows the total throughput measured when ten processes on ten different machines send messages (according to the various scenarios), and all the processes receive all the messages. The different scenarios are:

1. All the processes constantly send messages, completely independently.
2. The processes take turns in a round-robin fashion, each one sending a burst of messages in its turn.
3. Only a single process sends messages (the remaining processes receive all the messages).

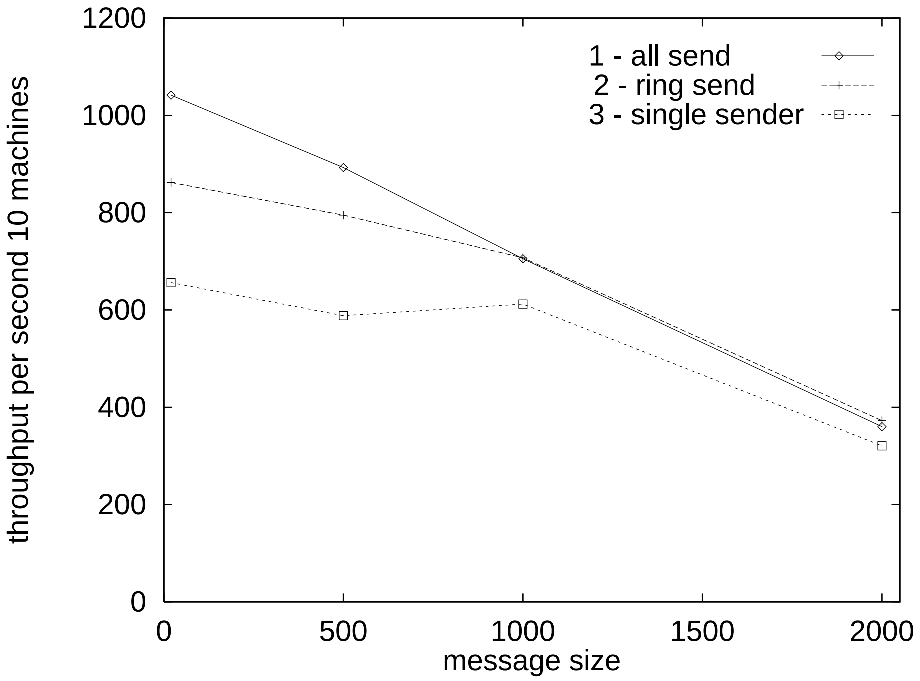

Figure 4: Throughput among Ten Machines for different message sizes

We note that the sharp drop in throughput between 1K messages and 2K messages is due to the Transis internal packet size, which has been chosen to be 1K. In order to send larger messages, Transis needs to packetize the message and emits multiple packets per message. We also note that we currently do not employ packing of multiple small messages into a single packet, which could result in a significant increase in message-throughput for small message sizes.

Figure 5 shows how message throughput mildly degrades as processes are added to the system. The test depicts the round-robin scenario (no. 2 above), in which 1K messages are sent among all the machines.

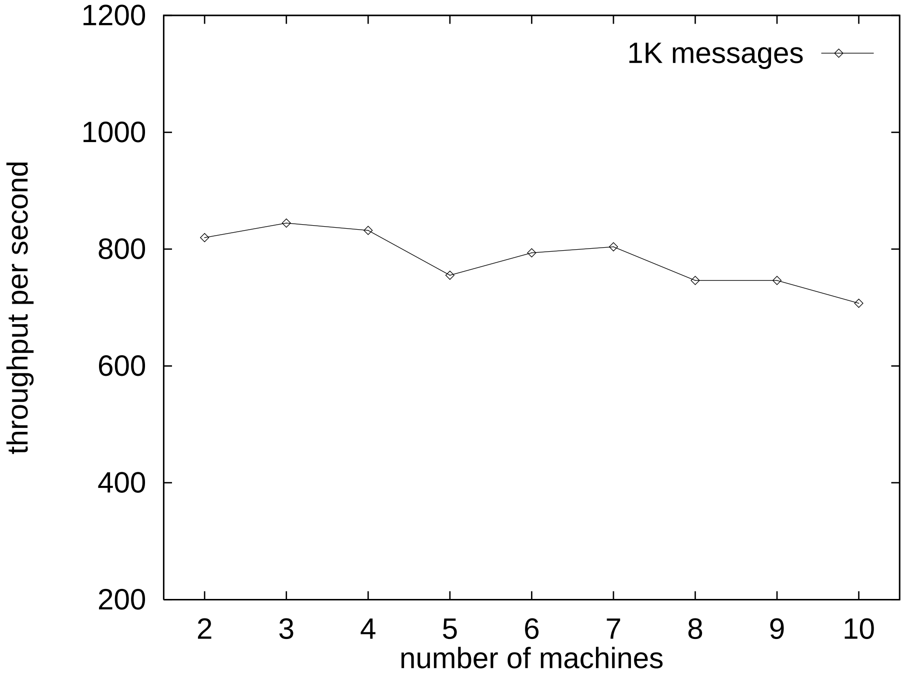

Figure 5: Throughput among varying number of machines

7 Membership

A fundamental issue in the design of a reliable multicast layer is the maintenance of the *membership* of operational machines. Transis contains a membership protocol that is integrated in the communication system, such that the notifications of *membership changes* are delivered to the application among the stream of regular messages [3, 27]. Changes to the membership are coordinated with the delivery of regular messages in the system.

The problem of maintaining a consistent view of the reachable operational machines is fundamental in the design of distributed systems. In [23], a formal definition of the problem (and solution) is given for synchronous systems. To the best of our knowledge, the first formal definition of the requirements of membership in asynchronous environments is given in [51]. Their paper defines a *primary-component membership service*, that maintains the *local views* of all the operational machines in agreement. Since it is well known that reaching agreement in asynchronous environments is impossible [28], the membership agreement must circumvent this difficulty somehow. The membership protocol in [51] uses an inaccurate failure detector, based on timeout: When a machine is presumed *faulty*, it is taken out of the view, such that further messages from the faulty machine are *discarded*. In reality, failure detection can be fine-tuned to avoid false-detection almost entirely. This approach is practical, and we adopt it, with an important modification: a presumed failed machine can rejoin the membership, and does not need to *give up*.

In the application requirement in [51], at most one component may exist, and machines outside the primary component are either dead or give up. In large and critical systems, this approach is not realistic, and it is essential to enable operation in face of partitions. In the case that partitioned operation is not desired, it is easy to add a layer on top of our membership that disallows all but one component to operate. Thus, our protocol does not mandate partitioned operation, but provides more flexibility.

When the network partitions, each component continues operating separately. The machines in each component are in agreement about membership among themselves, but not with the machines in other components. This might sound chaotic, at first. However, we require that:

- Every pair of machines that go through two consecutive membership changes, receive the same set of messages between the two changes. (This is a generalization of the principle called *virtual synchrony*, see [12]).

- Upon re-merging, all the machines in the new membership start with a consistent view of the membership, and agree on the messages that immediately follow it.

In this way, the membership service associates a membership-context with each message. The application can use this to perform consistent operations on received messages, and to merge the histories of joined components. For example, [2] describes a replicated mail server that exploits the Transis membership for efficiently implementing a partitionable service.

Intuitively, at the basis of our membership protocol, there are two stages: *(1) suggest a new membership set, (2) wait for agreement from every machine in the set.* This simple protocol is complicated by the following issues:

- All the machines in a new membership set need to terminate the preceding membership in a consistent way. The problem is that if a certain machine, q, is taken out of the previous membership, the adversary might cause the "last" message from q to reach only a subset of the new membership.

  We use the following rule: Let m_q be a message from q. If **any** machine in the new membership set receives m_q before committing to the new membership, **all** of them deliver m_q. Otherwise, all of them discard it.

- Our design allows the flow of regular messages to continue during transition to a new membership. This design goal is important for handling cascading membership changes, during periods of instability or frequent changes. It also makes the protocol flexible to support messages from external sources (an *open* group communication), because auxiliary sources are not part of the protocol and cannot cease sending messages during its operation.

  In our protocol, the context of regular messages that are sent during membership transitions is determined by their order with respect to the messages used within the membership protocol.

In [27], we present the full solution that supports partitioned operation and rejoining. Joining is done multi-way, in a completely symmetrical fashion. In this way, the joining provides a solution to the *startup* problem as well: Each machine starts up as a singleton set on its own. Then, all the machines that start up *merge* into a larger set.

8 Conclusions

Computer communication networks play a crucial role in today's computer environments. Using advanced communication facilities, one can replicate information cheaply, conveniently and more quickly.

The Transis approach to advanced group communication has acquired a wide recognition in the academic community, mainly due to the following desirable properties:

1. It employs a highly efficient multicast protocol, based on the Trans protocol [43], that utilizes available hardware multicast.

2. It can sustain extremely high communication throughput due to its effective flow control mechanism, and its simple group design.

3. Is supports partitionable operation, and provides the means for consistently merging components upon recovery.

As of approximately two years ago, Transis has been operational at the Hebrew University, and supports various applications as well as the "Distributed Algorithms" course.

References

[1] Afic Inc. MSO: A Replicated Database. private communication, 1993.

[2] O. Amir, Y. Amir, and D. Dolev. A Highly Available Application in the Transis Environment. In *Proceedings of the Hardware and Software Architectures for Fault Tolerance Workshop, at Le Mont Saint-Michel, France (LNCS 774)*, June 1993.

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms for Multicast Communication Groups. In *6th Intl. Workshop on Distributed Algorithms proceedings (WDAG-6), (LCNS, 647)*, pages 292–312, November 1992.

[4] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[5] Y. Amir, D. Dolev, P. Melliar-Smith, and L. Moser. Persistent Order Maintanence in a Partitioned Network. in preparation, 1994.

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

[7] A. Bhide and S. P. Morgan. A Highly Available Network File Server. RC 16161, IBM Research, May 1990.

[8] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12), December 1993.

[9] K. P. Birman. A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication. *ACM Operating Systems Review*, 28(1):11–21, January 1994.

[10] K. P. Birman. *Reliable Distributed Computing with the Isis Toolkit*, chapter Virtual Synchrony Model. IEEE Press, 1994. to appear.

[11] K. P. Birman, R. Cooper, and B. Gleeson. Design Alternatives for Process Group Membership and Multicast. TR 91-1257, Dept. of Computer Science, Cornell University, Dec 1991.

[12] K. P. Birman, R. Cooper, and B. Gleeson. Programming with Process Groups: Group and Multicast Semantics. TR 91-1185, dept. of Computer Science, Cornell University, Jan 1991.

[13] K. P. 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.

[14] K. P. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *11th Ann. Symp. Operating Systems Principles*, pages 123–138, Nov 87.

[15] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comp. Syst.*, 9(3):272–314, 1991.

[16] K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Press, 1993. to appear.

[17] A. Borg, W. Blau, W. Oberle, and W. Graetsch. Fault Tolerance in Distributed Unix. In B. Simons and A. Spector, editors, *Fault-Tolerant Distributed Computing, Lecture Notes in comp. sci. #448*, pages 224–243. Springer Verlag, 87. A revised paper of A. Borg, J. Blaumbach and S. Glazer, 'A Message System Supporting Fault Tolerance', 9th sosp, oct 83.

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

[19] D. R. Cheriton. VMTP: Versatile Message Transaction Protocol. RFC 1045, SRI Network Information Center, February 1988.

[20] D. R. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *14th Symposium on Operating Systems Principles*, pages 44–57, December 1993.

[21] D. R. Cheriton and W. Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Trans. Comp. Syst.*, 2(3):77–107, May 1985.

[22] R. Cooper. Experience with Causally and Totally Ordered Communication Support, A cautionary tale. *ACM Operating Systems Review*, 28(1):28–31, January 1994.

[23] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–187, April 1991.

[24] F. Cristian, B. Dancey, and J. Dehn. Fault-Tolerance in the Advanced Automation System. In *20th Annual International Symposium on Fault-Tolerant Computing*, pages 6–17, June 1990.

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

[26] D. Dolev, S. Kramer, and D. Malki. Early Delivery Totally Ordered Broadcast in Asynchronous Environments. In *23rd Annual International Symposium on Fault-Tolerant Computing*, pages 544–553, June 1993.

[27] D. Dolev, D. Malki, and H. R. Strong. An Asynchronous Membership Protocol that Tolerates Partitions. submitted for publication. Available as CS TR94-6, Institute of Computer Science, the Hebrew University of Jerusalem, 1994.

[28] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.

[29] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In S. Mullender, editor, *chapter in: Distributed Systems*. ACM Press, 1993.

[30] J. Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. In *3rd Annual ACM Symp. on Principles of Distributed Computing*, pages 50–61, 1984.

[31] F. Jahanian and W. Moran. Strong, Weak and Hybrid Group Membership. In *IEEE Workshop on Management of Replicated Data*, pages 34–38, 1992.

[32] F. S. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije University, 1992.

[33] M. F. Kaashoek and A. S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *11th Intl. Conference on Distributed Computing Systems*, pages 882–891, May 1991.

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

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

[36] S. Kramer. Total Ordering of Messages in Multicast Communication Systems. Master's thesis, Inst. of Computer Science, The Hebrew University of Jerusalem, 1992.

[37] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Lazy Replication: Exploiting the Semantics of Distributed Services. In *9th Ann. Symp. Principles of Distributed Computing*, pages 43–58, August 90.

[38] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, July 78.

[39] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 226–238, Oct. 1991.

[40] B. Liskov and R. Ladin. Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Collection. In *5th Ann. Symp. Principles of Distributed Computing*, August 86.

[41] D. Malki and R. van Renesse. The Replication Service Layer. internal manuscript, 1994.

[42] K. Marzullo and F. Schmuck. Supplying High Availability with a Standard Network File System. In *4th Intl. Conf. Distributed Computing Systems*, pages 447–453. IEEE, June 1988.

[43] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Trans. Parallel & Distributed Syst.*, 1(1):17–25, Jan 1990.

[44] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Membership Algorithms for Asynchronous Distributed Systems. In *Intl. Conf. Distributed Computing Systems*, pages 480–488, May 91.

[45] 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*, pages 137–145, Feb 1991.

[46] S. Mishra, L. L. Peterson, and R. L. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. TR 91-32, dept. of Computer Science, University of Arizona, 1991.

[47] S. Mishra, L. L. Peterson, and R. L. Schlichting. Experience with Modularity in Consul. *Software Practice and Experience*, 23(10):1059–1076, October 1993.

[48] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended Virtual Synchrony. In *Intl. Conference on Distributed Computing Systems*, June 1994. to appear. Also available as technical report ECE93-22, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.

[49] 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.

[50] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing.* Springer-Verlag, 1991.

[51] A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *proc. annual ACM Symposium on Principles of Distributed Computing*, pages 341–352, August 1991.

[52] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE trans. on Computers*, 39(4):447–459, April 1990.

[53] F. Schneider. Implementing Fault Tolerant Services Using The State Machine Approach: A Tutorial. *Computing Surveys*, 22(4):299–319, December 1990.

[54] A. Siegel. *Perfomance in Flexible Distributed File Systems.* PhD thesis, dept. of Computer Science, Cornell University, Feb 1992. (TR 92-1266).

[55] A. Siegel, K. P. Birman, and K. Marzullo. Deceit: A Flexible Distributed File System. TR 89-1042, dept. of computer science, Cornell University, Ithaca, NY, Nov 89.

[56] R. van Renesse. Causal Controversy. In *5th ACM SIGOPS Workshop*, 1992.

[57] R. van Renesse. Why bother with CATOCS? *ACM Operating Systems Review*, 28(1):22–27, January 1994.

[58] R. van Renesse, K. P. Birman, R. Cooper, B. Glade, and P. Stephenson. Reliable Multicast between Microkernels. In *Proceedings of the USENIX workshop on Micro-Kernels and Other Kernel Architectures*, pages 27–28, April 1992.

[59] R. van Renesse, R. Cooper, B. Glade, and P. Stephenson. A RISC Approach to Process Groups. In *Proceedings of the 5th ACM SIGOPS Workshop*, pages 21–23, September 1992.

[60] P. Verissimo, L. Rodrigues, and J. Rufino. The Atomic Multicast Protocol (AMp). In D. Powell, editor, *Delta-4: A Generic Architecture for Dependable Distributed Computing*, pages 267–294. Springer-Verlag, 1991.

[61] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *9th Symp. on Operating Systems Principles*, pages 49–70, 1983.