Designing Application Software in Wide Area Network Settings *

Mesaac Makpangou Ken Birman†

October 17, 1990

Abstract

Progress in methodologies for developing robust local area network software has not been matched by similar results for wide-area settings. In this paper, we consider the design of application software spanning multiple local area environments. For important classes of applications, simple design techniques are presented that yield fault-tolerant wide area programs. An implementation of these techniques as a set of tools for use within the ISIS system is described.

Keywords and phrases: Process-groups, ISIS, fault-tolerance, wide-area protocols, network partitions.

1 Introduction

There is growing recognition of the utility of the *process-group* paradigm in distributed computing. In this approach, distributed software is structured into groups of processes that cooperate to implement distributed services, share replicated data, monitor one another, and so forth. The communication facilities for such systems typically extend conventional IPC and RPC to include group multicast protocols.

Many systems have implemented group facilities (process groups in the V system [5] and ISIS [2], port groups in Chorus [11], etc). Likewise, a variety of group multicast protocols have been implemented (atomic multicast, causal multicast, reliable multicast, multiRPC, etc). Unfortunately, although it is widely accepted that local area networks (LAN) and wide area networks (WAN) have different characteristics, most work on process groups and group communication has been restricted to LAN environments. For example, most such systems assume low communication latency, high bandwidth, and that although individual messages may be lost, network partitions do not occur. These assumptions hold in a LAN environment but are not typical of the WAN environment. Consequently, mechanisms and multicast protocols that give acceptable performance in a LAN environment, might perform poorly (or incorrectly) in a WAN environment.

This paper examines wide area applications constructed by interconnecting process groups located in different LAN systems. Such applications may present an integrated interface abstraction, but will typically operate by binding the user to a local representative of the WAN service, which

*This research was funded in part under DARPA/NASA subcontract NAG-2-593, and in part under DARPA contract MDA-972-88-C-0024.

responds to requests using local data whenever possible. Our goal is to identify and implement a suitable collection of WAN tools to assist in this process. These consist of mechanisms and protocols that assume that applications will be long-running and will experience such problems as partitions, network crashes, and long haul connection failures.

Because few WAN applications have been developed, we lack a good model for applications of this sort. To overcome this, we begin by examining problems that arise in a WAN application for capture and analysis of seismic signals. We then turn to the problem of implementing the facilities needed to solve this problem. Finally we discuss a general framework for the support of wide area applications, presenting this in the context of the Isis environment.

The rest of this paper is organized as follows. Section 2 discusses our assumptions about the computing environment. Section 3 discusses the applications we have selected and examines their support requirements. Sections 4, 5 and 6 discuss the mechanisms and long haul protocols that emerge from these case studies and provide performances figures for our initial implementation.

2 Background and assumptions

2.1 The wide area system model

Figure 1 illustrates the overall architecture of a wide area environment. The system is composed of a set of *local area networks*, interconnected by point-to-point long haul links that comprise the *wide area network*. The term *cluster* denotes the set of sites belonging to a single local area network. More than one link may connect two clusters.

Computing within a cluster takes place in *processes* that communicate via messages. A *process group* is a set of processes that are cooperating for some purpose. Our work was done in the context of Isis, a system that provides extensive support for process groups and reliable group communication. Isis process groups do not span multiple clusters.

We say that process groups located in different clusters are *related* if they communicate with one another. A *partitioned wide area application* is one composed of related groups. Figure 1 depicts a situation where we have two partitioned wide area applications represented on each cluster by the process group named respectively G1 and G2.

A *local multicast protocol* designates a protocol used to multicast a message to the members of some process group. A *long haul multicast protocol* designates a protocol used to multicast a message to the members of a set of related groups.

2.1.1 Failure assumptions

We assume that each LAN system "isolates" the effect of a host crash, local connection failure, and LAN partition. This means that only application components located within the affected cluster are involved in the detection and handling of these events. These assumptions hold for our Isis-based implementation, but might limit the applicability of our work to other LAN-based systems.

With regard to wide-area communication, we assume that long haul connection failures, cluster crash, and WAN partition can all occur. Because clusters may be redundantly connected we will say that a *long haul connection failure* occurs when a link connecting two clusters fail, and that a *WAN partition* occurs when all such links fail. It will be useful to distinguish two subcases of WAN partitioning:

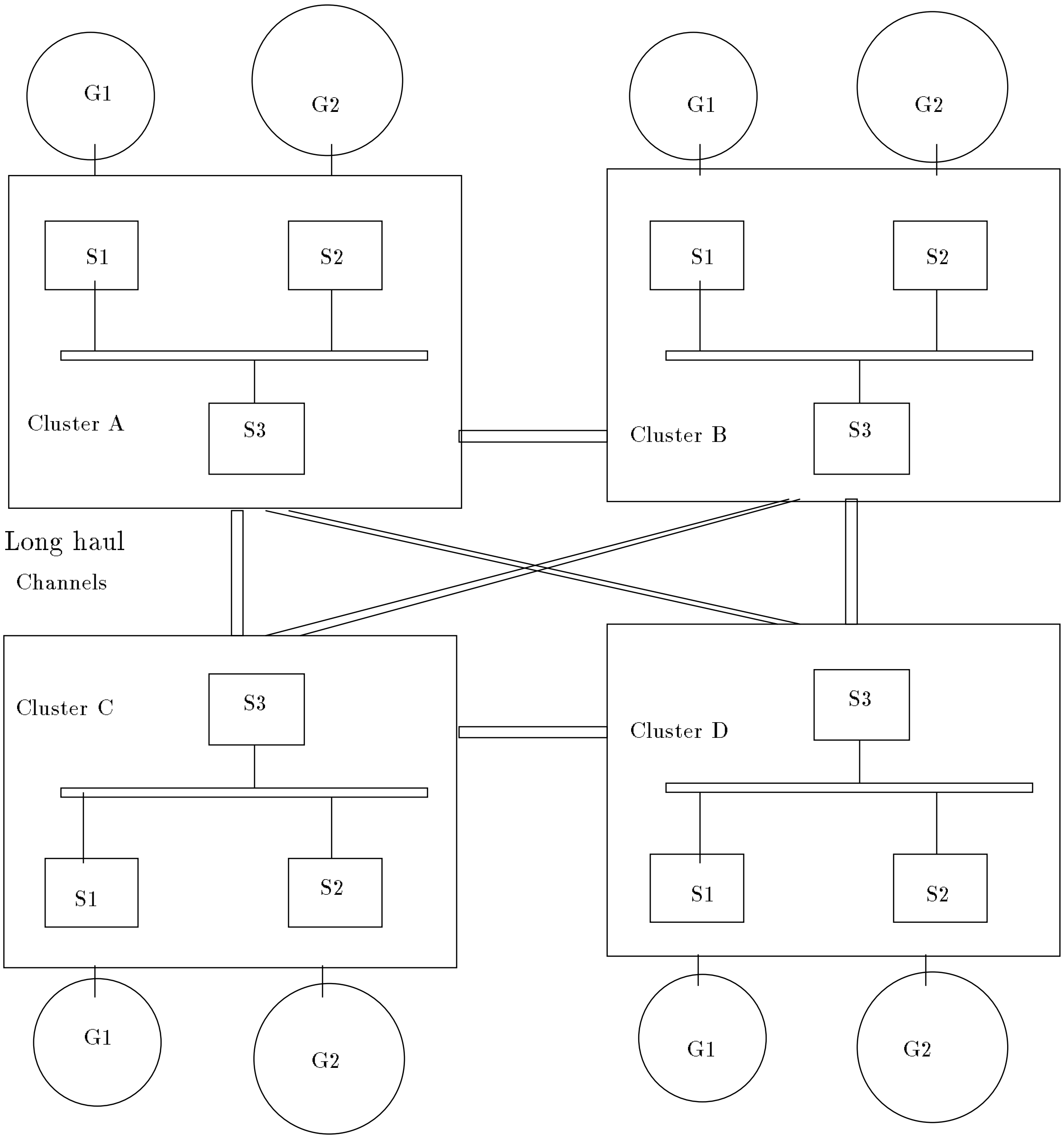

Figure 1: Overall architecture of a wide area system

Controlled WAN partitioning

WAN communication lines may be costly or subject to physical constraints that cannot always be satisfied (i.e., a satelite link will need a line-of-site path to a satelite). For these reasons, many applications use a *periodic* communication model. As needed (or whenever possible), clusters open communication links. Data is shipped across the links, which are then closed. We will refer to this kind of partitioning as controlled partitioning.

Unplanned partitions

A WAN partition is unplanned if it results from an unpredictable event such as the failure of the only communication line linking two clusters or the failure of a machine managing an endpoint of such a line. Such a partition is undistinguishable from the simultaneous failure of all the machines in one of the clusters. Our work assumes that no failure lasts indefinitely and hence that communication will eventually be reestablished. Accordingly, we focus on wide area applications explicitly designed to tolerate the delay introduced by unplanned partitions.

The following additional terminology is used throughout the rest of this paper. A *partition* is a WAN partition. An *application* is a wide-area application, formed of a set of related groups running in separate partitions. And, a *connection* is a single long haul communication channel.

2.1.2 An impossibility result

There exists a substantial body of work on protocols for environments subject to unplanned partitions. The work most relevant to systems like Isis is by Skeen, who proves that protocols having the characteristics of a two- or three-phase commit cannot be terminated safely in the presense of possible partitioning failures [10,6].[1] The LAN implementation of Isis uses multi-phase commit protocols at its lowest levels, to maintain information about the status (operational/failed) of process-group members. This information drives the higher levels of the system.

An implication is that little of the software commonly used by Isis in LAN settings can be modified to work correctly in a WAN environment. In particular, the form of consistency that Isis supports cannot be made tolerant of network partitions without risk of "blocking" when partitions occur. The current version of Isis finesses this issue by shutting down the sites in a "minority" (smaller) partition. Were Isis to be used in a WAN setting, one would sacrifice either consistency (correct, predictable behavior) or availability.

Notice that although Skeen's results preclude any transparent scaling of the existing Isis systems – or any similar system – into a WAN environment, it *is* possible to make LAN systems highly resilient to failure, and the existing Isis toolkit is quite effective at using state replication for this purpose. This justifies our assumption that LAN services will be highly available (recovering rapidly from crashes) and will not lose "committed" state – the property we referred to as failure isolation, above.

[1] Readers familiar with the database literature will be aware of several approaches that yield transactional serializability in the presense of partition failures. Unfortunately, these protocols cannot be extended into protocols for consistent group management and atomic communication, which are the cornerstones of the approach.

2.1.3 Long-haul channels

We initially assume that inter-cluster communication is by a *communication-failure free fifo channel.* Such a channel has the following properties:

- All messages sent from one cluster to another are received in the order sent.
- Inter-cluster communication is not subject to message duplication or packet loss, even in presence of connection failures.

These characteristics are stronger than what a general purpose transport protocol like TCP or any of the five ISO transport classes provides, because we require these properties even when multiple physical communication links exist between a pair of clusters and even when links fail or are restarted during the course of execution. In Sec. 5 the implementation of a communication channel with these properties is shown to be feasible using existing Isis facilities.

2.2 Impact of WAN characteristics on protocol design

For purposes of protocol design, a wide area network (e.g. ARPANET) differs from a local area network (e.g. ETHERNET) primarily in four respects: higher latency, lower bandwidth, point to point connections, and a higher probability of partition. These differences, together with the assumption that the application components located in different LAN systems are loosely coupled (that is, they interact relatively infrequently and most interaction is asynchronous), have a substantial impact on the implementation of long protocols, particularly those involving more than a pair of participants (such as multi-phase commit or reliable multicast):

1. *Network partition must receive more attention.*

   In a LAN environment, the low probability of partition makes it feasible to either ignore these events, or to implement a harsh solution such as the Isis approach cited above. Such a treatment can be justified, at least in moderately small LAN systems, because partitions will be so infrequent and because when LAN failures actually occur, they provoke large numbers of machine failures by separating application programs from resources on which they depend. If large numbers of machines are crippled by a partition failure, simply assuming that these machines have actually failed may not be unreasonable. Isis users have reported little trouble with this restriction.

   In a WAN environment, partition will often be the usual state, with clusters contacting each other periodically so as to minimize the cost of maintaining open connections for long periods of time and to maximize the use of connections when they are opened. Moreover, because applications will be loosely coupled, a WAN partition will generally not trigger large numbers of machine failures. These considerations make it important to limit the impact of a partition and to provide mechanisms by which applications can offer some restricted (or autonomous) level of service in partitioned settings.

2. *Multicasting only when it is really necessary.*

   Systems like Isis often structure applications and services using a collection of small process groups with perhaps 3 or 4 members each. A request on such a group may be implemented as an IPC or RPC to a favored member, or as a multicast[2] to the full set. In this case, either all members perform the request in parallel, or one member performs the request while the others back it up for fault-tolerance. The primary/backup approach is encouraged in Isis because different group members can respond as the primary server for different requests, providing a form of load sharing. This approach is inexpensive because it benefits from the comparatively high speed of communication and because the backup processes for one request will be working actively on other requests. Moreover, the multicast itself may make use of special LAN hardware facilities.

In a WAN environment, casual use of a "large-scale multicast" could lead to poor performance due to the long latency of WAN communication, lower WAN bandwidths, and possible restrictions on establishing and using WAN communication links. Consequently, the Isis style of programming will not map transparently to WAN applications. Instead, such applications will normally communicate with the WAN application through the group representing that application on the local cluster. As much as possible, this group will respond to requests using local information. If information from a remote server is needed, it will most often request it using some form of point-to-point long haul communication. On the other hand, a WAN multicast might remain useful for asynchronous purposes, such as the diffusion of information to the groups in a partitioned wide-area application.

3 Case studies

This section discusses a series of problems motivated by a set of wide-area seismic monitoring applications collectively called the *Nuclear Monitoring Research and Development System*, or NMRD, being developed by Science Applications International Corporation under contract to DARPA. [3] NMRD includes several knowledge-based applications which collect, analyze and archive seismic data from a geographically dispersed network of seismic sensors, and a rich set of tools for selecting and analyzing data in the archive to address seismological issues. The system is extensively automated with rule-based AI techniques.

The largest and most complex element of NMRD is the *Intelligent Monitoring System* or IMS which detects, locates, and identifies seismic events using data from a network of stations in Eurasia. IMS is structured as a collection of LAN clusters, initially placed in Washington, Norway, and San Diego. As the system is developed, there are potential requirements for expansion to include several more LAN clusters.

Our group became involved in developing LAN and WAN software for NMRD and IMS in 1989. The LAN aspects of NMRD are concerned with system fault-tolerance and configuration management, communication, LAN resource scheduling, and related issues. All of these aspects are beyond the scope of the present paper. Below, we focus on WAN use of Isis in the current IMS prototype.

Currently, IMS is structured like a wheel, with a central "hub" in Washington, DC, that performs most of the automated data interpretation functions. A set of "spokes" connect this hub to free-standing LANs which acquire the data and do extensive signal processing to select and

[2] We are using *multicast* in the sense of a software protocol for communicating with the full membership of a dynamically changing group - not in reference to a hardware feature.

[3] DARPA Contract No. MDA972-88-C-0024

characterize data segments which may have signals of interest. The central interpretation done at the "hub" plays a crucial role in this selection. The spokes comprise the WAN communication network, and consist of long-distance TCP channels. Most of the WAN communication consists of automatically initiated data selection and transfer operations, with the hub software issuing requests to the remote subsystems. Because the system is automated, the fault-tolerance of these operations is critical to correct function.

In the future, IMS and other NMRD subsystems may grow to include multiple hubs, supporting seismic researchers as well as automated analysis, and this will make it important to support a number of of additional WAN services. The discussion that follows examines some of these hypothetical issues after briefly commenting on the file transfer problem.

3.1 File transfer and remote notification

The most common of the WAN applications arising in IMS concern inter-LAN event notification and file transfer. The initial signal processing is done close to the data acquisition systems to avoid the requirement that all data be transferred to the hub. All acquired data are processed to detect signals and characterize them in terms of a standard set of parameters which are archived in a local commercial relational database management system (RDBMS). On a regular schedule (e.g., every 15 minutes), the hub initiates a request to transfer data from the remote RDBMS to the central RDBMS at the hub. The automated knowledge-based system (KBS) at the hub analyzes the data from all stations to locate and identify all detected events. Depending on the location and character of the events formed by the KBS, a request is formed for relevant segments of the raw data.

The sequence of steps involved in such a raw data transfer is as follows. First, the ISIS long-haul utility is invoked by an IMS program running on the hub with a message describing the data to be retrieved (station and time interval). The remote portion of IMS receives this message, retrieves the requested data and initiates the file transfer to the hub. When the file transfer takes place, a suitable spooling area is found for the incoming data and notifies the hub process that initiated the retrieval. Finally, after the transfer has completed successfully, the remote file is deleted. This procedure is generalized by replication for additional remote sites. Fault-tolerance is key here: errors such as failure to transfer files, lost or duplicate notification messages, and so forth cause problems requiring later human intervention.[4]

3.2 Resource location

Resource location is the problem of mapping resource names into information about the location and contents of the named data objects. This is the problem solved by so-called "white pages" services, and represents an active research topic. Because the current IMS system is centralized, the problem does not yet arise. However, WAN solutions to the resource naming problem would become important if the system expands to include multiple hubs.

Imagine an IMS-like system running with many integrated computational hubs. Each of these hubs would have the ability to request information *(new_data)* from outer clusters (data that was not provided as part of routine processing). Obtaining and analyzing *new_data* may involve expensive

[4] IMS almost never "crashes" due to software failures – the system tries to handle errors gracefully. However, errors may cause the system to lose things – events, data for the analyst to review, etc. In cases where the lost data may be important, a fairly tedious manual corrective action will eventually be needed.

(in terms of resources) data retrieval and processing operations. For example, it might require that a complex data adaptive beamforming operation be performed; such computations may require hours of CPU time. Clearly, one would not want to perform this sort of operation on hub A when hub B has already performed one. It follows that when a *new_data* request is made, a service will be needed to determine if the computation has already been performed (or is underway), and if so, whether it would be cheaper to transfer the computational results or to transfer the raw data and repeat the analysis locally.

It is natural to think of such a version of IMS as generating and manipulating a large event–file or database. This file would identify both raw events and the location (and size, and computational cost) of the corresponding processed data file, or the location of any hub currently engagaed in such a computation. The problem can thus be reduced to one of locating resources in a WAN.

A number of difficult problems now arise. First, observe that the naming space is a dynamically changing one with several natural forms of hierarchy: physical hierarchy in space (i.e., the set of events known only within some local cluster), logical hierarchy (i.e., the set of raw-data objects associated with some *new_data* event), and global hierarchy (i.e., a set events currently under consideration as evidence that a nuclear test has been detected). Operations on the naming space will be search requests, read requests, and update requests. For simplicity of design, one would want this namespace to present a seamless global abstraction. At the same time, information should be maintained close to where it will be generated or manipulated, to avoid excess WAN communication.

Consistency or coherency of such a WAN naming structure will correspond to the property that any update eventually reaches all clusters with a copy of an event descriptor, that read operations preserve the abstraction of a single global namespace, and in particular, that updates appear to be serialized. To see this, consider a computation that reads a descriptor (say, a correlation descriptor). The computation should subsequently see "current" copies of any other event descriptors on which this descriptor depends; otherwise, it would appear that the namespace has somehow become corrupted. Such a relationship is *causal*, and we will have more to say about mechanisms for enforcing causal orderings shortly.

For brevity, we will not develop a complete solution to this problem here. We observe, however, that the core mechanisms needed here will be ways to form WAN groups and to multicast updates to the group members. Given such tools, the resource management service would be structured into a collection of information domains within which updates would be multicast to all members. To ensure that the namespace presents a causally consistent abstraction, we will need to know that any multicast sent to such a WAN group (eventually) reaches all its members, and that if an update is dependent upon some prior update, then all WAN group members see the two updates in the order they were issued. Notice also that once a WAN group is formed in this application, its membership remains fairly stable. Only the creation of new hubs or thew addition of new sensor clusters would require changes in this part of the system configuration. Both operations will obviously be infrequent. The physical scale of WAN systems suggests that this form of stability should be fairly common. On the other hand, *within* such a WAN mulitcast group, one can easily imagine needing to send messages to a subset of the total membership.

3.3 Resource scheduling

The above examples show how IMS uses WAN file transfer and WAN multicast. They also hint at the need to support WAN resource allocation and scheduling policies in an extended system.

Notice that the existing IMS permits an analysis program or researcher working in Washington to initiate data retrieval requests and computation in Norway. This is not a major issue if there is only one hub. However, with multiple analysis hubs, it would become important to partition computational cycles among the various hub systems contending for database access and signal processing facilities. Otherwise, it would be easy for an IMS component at one location to overload a cluster located halfway around the world, preventing it from accomplishing locally critical tasks such as data compression and event detection, or even denying local analysis systems a fair share of the computational resources.

We can abstract this problem as one of selling tickets for a periodic event. Only a process holding the appropriate tickets will be granted access to the processor pool on a given LAN. An "event" in this formulation might correspond to one specific hour of activity on the Norway cluster, and a ticket to a permission to perform five minutes computation during that hour. The ticket sales problem has substantially more structure than the basic file transfer and remote notification problems seen in our first example.

A solution to this problem should address two goals. The first arises from the need to design a *loosely coupled* scheduling service. It should be possible to sell tickets for a future event on a remote cluster even if communication with that cluster is presently impossible, if a connection fails during the interaction, or even if a partitioning or cluster failure occurs. A second goal is that the system should satisfy the maximum number of demands possible (presumably using an application–specific cost function) while also guaranteeing fairness (also an application-specific notion).

Let us ask what can be said about this problem without speculating on the application-specific aspects. Clearly, if the distribution of tickets is static and fixed, a cluster that receives a large number of demands may not be able to satisfy all of them, while some other cluster may fail to sell some of the tickets it holds. This will compromise the second goal, and suggests that the distribution algorithm will either need a central decision making mechanism or a way to dynamically repartition the collection of tickets. A centralized policy would violate our first goal. Thus, we need a dynamic distributed allocation policy. Such an approach might pre-allocate tickets to clusters, but include a mechanism for reallocating unsold tickets as the "event period" approaches. Ideally, we would want this mechanism to make progress even if a communication failure or partition occurs.

3.3.1 Structure of the application

Assume that we have N clusters and that a group of ticket vending processes are active in each cluster. We will partition the pool of tickets in N subsets and pre-allocate each to a specific cluster. Each vending group uses its partition to serve demands from its local workers. Next, we divide the selling period in subperiods. At the end of each subperiod, each server multicasts a state message to its peers. This message reflects recent sales as well as the anticipated needs of the sender. Finally, on the basis of the state messages it receives, each server computes a new partitioning of unsold tickets using some deterministic, well known algorithm.

3.3.2 Classes of ticket repartitioning algorithms

Repartitioning algorithms can be characterized by their sensitivity to the delivery order of state messages, and by the degree to which actions by servers in different partitions are synchronized. We distinguish three classes of such algorithms:

1. *Class 1* consists of algorithms that operate asynchronously and are insensitive to the order in which state messages are received from different servers. These are all fixed, well known, and deterministic repartitioning algorithms. For example, suppose that we have five servers. An algorithm in class 1 might assign 1/5 of each lot of unsold tickets carried by each state message to each server. Notice that even if different servers see state messages in different orders, the number of tickets available to a given server in a given round will be the same.

   Class 1 algorithms are simple and stateless: they require only that the system provide eventual delivery of each state message its destinations, and that the set of participants be fixed before execution starts. We refer to WAN multicasts satisfying this eventual delivery property as *fault-tolerant WAN multicasts.*

2. *Class 2* algorithms operate by having each server wait for all the round-k state messages before carrying out the repartitioning for round $k+1$. Such an algorithm has more flexibility than the class 1 algorithms because it operates with full knowledge of ticket sales, availability of unsold tickets, and anticipated demand. Again, the algorithm must be deterministic and well known, so that all servers can execute it in parallel. Class 2 algorithms are thus insensitive to the order in which messages are received but synchronous. Like their counterparts in class 1, these algorithms require that the system provide information about the set of participants and support for fault-tolerant multicasts.

3. *Class 3* algorithms are sensitive to the delivery order of state messages and asynchronous. For example, consider a system in which a server needing tickets broadcasts its need, and servers with a surplus broadcast the existence of the surplus. One might imagine a rule under which all servers, in parallel, reallocate tickets as each such message is received. Such a scheme has the advantage of making progress as rapidly as possible, as in the class 1 algorithms, but without requiring the rigid determinism of the class 1 algorithms.

   However, the order in which messages containing ticket requests are received may affect the way that tickets are repartitioned in this case. In general, servers implementing class 3 algorithms may need to see all state messages in the same order, or at least in a predictable order. We will refer to such multicasts as *ordered WAN multicasts.*

3.3.3 General remarks

1. *Class 1* algorithms will perform poorly if demands are not uniformly distributed within the WAN system as a whole. Typically, for these algorithms to maximize the number of requests satisfied, the selling period will need to be divided in small subperiods. Such division will increase the wide-area network traffic making the application components more *tightly coupled.*

2. *Class 2* algorithms might reduce availability at certain locations. Suppose that some server has no more tickets to sell. Even if it has already received a state message indicating that unsold tickets exist on some other server, and even if the repartitioning algorithm is such that

it will be allocated some of these at the repartition time, it has to wait until it receives all state messages before granting any further requests.

3. Because *class 3* algorithms allow servers to operate asynchronously, these are more likely to yield a *loosely coupled* solution. However, *class 3* algorithms need a multicast primitive with known delivery ordering properties, and this may be a more costly primitive than the one used in a class 1 asynchronous algorithm. We return to this issue below.

4. Communication failures will affect all these algorithms by delaying the delivering of state messages.

   - For *class 1* algorithms, delays impact ticket availability at certain locations. For example, suppose the two subsets of servers {A, B} and {C, D, E} are isolated from one another. Naturally, messages about unsold tickets released by each subset will not reach the other during the partition. Therefore any tickets released by A or B that the algorithm will assign to C, D or E will remain unused during the partition.
   - For *class 2* algorithms, the delay might completely inhibit ticket repartitioning for the duration of the partition.
   - Finally, for *class 3* algorithms, delays impact the availability of unsold tickets in certain partitions. Moreover, communication partitions might prevent the algorithm implementing atomic WAN multicast from making progress in certain partitions. For example, if WAN multicast is done using a multi-phase protocol, a partition during the first round could completely inhibit the delivery of WAN messages for the duration of the partition. This suggests that one-phase protocols are strongly preferable to multi-phase protocols in WAN settings.

3.4 Summary of WAN communication requirements

The examples discussed above seem representative of a reasonably large class of wide area applications. In this section, we summarize the essential WAN communication requirements that emerge.

An abstraction super-imposed upon the concept of group

WAN applications will typically need communication between a set of related groups located in different clusters. This wide area set of groups (*wSet*) constitutes a new WAN abstraction super-imposed upon the existing Isis LAN process group mechanisms. In such a set, each element is a group and there is at most one element on each cluster. It must be possible to transmit messages to individual members of this set of groups as well as to the set as a whole. Unlike groups in LAN settings, it seems reasonable to assume that *wSets* change infrequently after creation.

Fault-tolerant multicasts

Certain applications need a multicast protocol tolerant of failures. Such a protocol will eventually deliver messages to all its destinations even in presence of partitions, network crashes or connection failures. If a server issues a fault-tolerant multicast and then fails, and the system has "accepted" the message in a sense discussed below, this fault-tolerant multicast must be delivered sooner or later to all its destinations. Conversely, when a server recovers from a crash, it should be able to recover pending fault-tolerant multicasts destined to it.

Atomic and causal ordering

The name-server and class-3 scheduling problems point to application-level dependencies on the order in which related groups receive WAN multicasts. Our group has explored this issue in some depth in LAN settings, and we will not repeat this material here. To summarize, there are two forms of multicast delivery ordering of possible interest in applications with a group structure. One provides that all group members see the same messages in the same order. This has been called an atomic order in the literature. The second is a generalization of a fifo ordering, and consists of a multicast primitive that delivers messages in the order they were sent, which Lamport has termed the *happens before* or *potential causality* ordering.

That is, say that m_1 and m_2 are multicast messages and let $m_1 \prec m_2$ denote that m_1 was sent before m_2 (i.e. that there exists a path of messages and local actions linking the sending of m_1 to the sending of m_2). Lamport refers to $\prec$ as the "happens before" relation [7], because if $m_1 \prec m_2$, m_2 may somehow depend upon m_1. m_1 and m_2 are concurrent (were send "in parallel" by independent senders) if neither $m_1 \prec m_2$ nor $m_2 \prec m_1$. A multicast is said to be *causally ordered* if whenever $m_1 \prec m_2$, it delivers m_1 before m_2 at any destinations they have in common. A multicast is said to be *atomically ordered* if m_1 and m_2 are delivered in a fixed order at all common destinations, even if they were concurrent.

The basic practical difference between a causal and an atomic multicast is performance. Causal multicast can be implemented as a one-phase protocol that delivers most messages promptly upon reception. Atomic multicast is more ordered, and this forces such protocols to delay some messages in situations where a causal multicast would not. In fact, there are no one-phase multicast protocols for asynchronous systems. ISIS implements its atomic multicast protocol using two phases of causal multicasts; messages are delivered during the second phase. The resulting protocol is about one–half to one–third the speed of the causal one.

In [9], Frank Schmuck has demonstrated that most software designed to run over an atomic multicast protocol can be modified to run over a causal multicast. In a long-haul setting, this has an obvious benefit, since a protocol that runs in more phases will not only be slower, but will also have much higher risk of being delayed due to a partition. In our work, we will assume that most class 3 algorithms are built using a causal multicast; the remainder would run over a 2-phase atomic multicast that is itself built using a causal multicast.

Readers knowledgeable about ISIS will recognize that these needs are similar to the ones addressed by the ISIS system in LAN settings. However, three points distinguish our wide area system from systems like ISIS. The first concerns the type of asynchronous computation that arises in WAN settings and WAN applications. In ISIS, asynchronous computation is common, but it is normal to assume that asynchronous operations have low latency. In the WAN setting, latencies could be very large. The second is that most long haul applications will be loosely coupled. In a LAN, ISIS assumes that typical applications are object-oriented and hence consist of multiple, closely related process groups. In a WAN, it would be rare for a single application to make use of multiple wide-area groups. Consequently, messages exchanged exclusively between the servers associated with one WAN service are not likely to be related to those exchanged between servers of some other WAN service. Finally, WAN services will probably not change membership very often. In fact, most applications that we have considered are represented on every LAN cluster in the wide-area system.

4 Wide area sets and Long haul multicast protocols

The wide area system is structured in three layers (see figure 2). From the bottom to the top we have the transport layer, the interconnection layer, and the application manager layer. The transport layer implements a reliable end-to-end transport protocol. In our current system, this layer consists of a TCP channel. The interconnection layer implements *communication-failure free fifo channels* between pairs of clusters. The *application manager* layer manages the *wSets*, and the long haul multicast protocols. In this section, we focus on the *application manager layer*. Section 4.1 presents the *wSet* support. Section 4.2 formally defines the two long haul protocols emerging from our case studies.

4.1 Wide area sets

4.1.1 Spooling facility

Our case studies emphasize the importance of asynchronous communication in WAN settings. A basic characteristic of such communication is that processes may transmit messages without waiting until they have been delivered. This creates a buffering obligation if a long delay may occur before a message can actually be sent to its destination.

Accordingly, a reliable spooling facility is used as a core component of our system. Each spool is a reliable service restricted to within a single LAN, and built using the basic Isis toolkit. Spools provide a persistent buffering mechanism. When we say that a message is *spooled* to a logical address, we mean that the message is written to a stable log; a copy is also sent to the service associated with the address if it is running. A service that has failed will restart by initiating a spool replay operation, causing messages in the spool to be delivered in the order spooled to the service. When a message will no longer be needed, it can be removed from the spool. The spooling service is typically configured to activate automatically when certain services are not operational; the service empties and deactivates the spool after replay is completed.

During communication failures, messages that cannot be sent to a destination cluster are spooled in what we call an *interLAN* spool area. After communication is re-established, these messages are retrieved from the spool and sent to their destinations. The *interLAN* spools are located within the interconnection layer.

To deal with application-level asynchrony (i.e. an application that only runs periodically, or that is temporarily unavailable because of component failures), we also associate a *wide area application spool* (or *wSpool*) with each wide area application. A *wSpool* consists of a set of spools, one in each cluster where the application is represented. When a WAN message is deliverable to an application, but the local representative is not available, the message is logged into the corresponding *wSpool*. Once the local group recovers, it initiates spool replay and then shuts the spool down. The *wSpool* management software is part of the application manager layer.

During periods when an application is operational on all clusters and there are no WAN partitions, all *wSpools* will be inactive, and the *interLAN* spools will be updated asynchronously. In

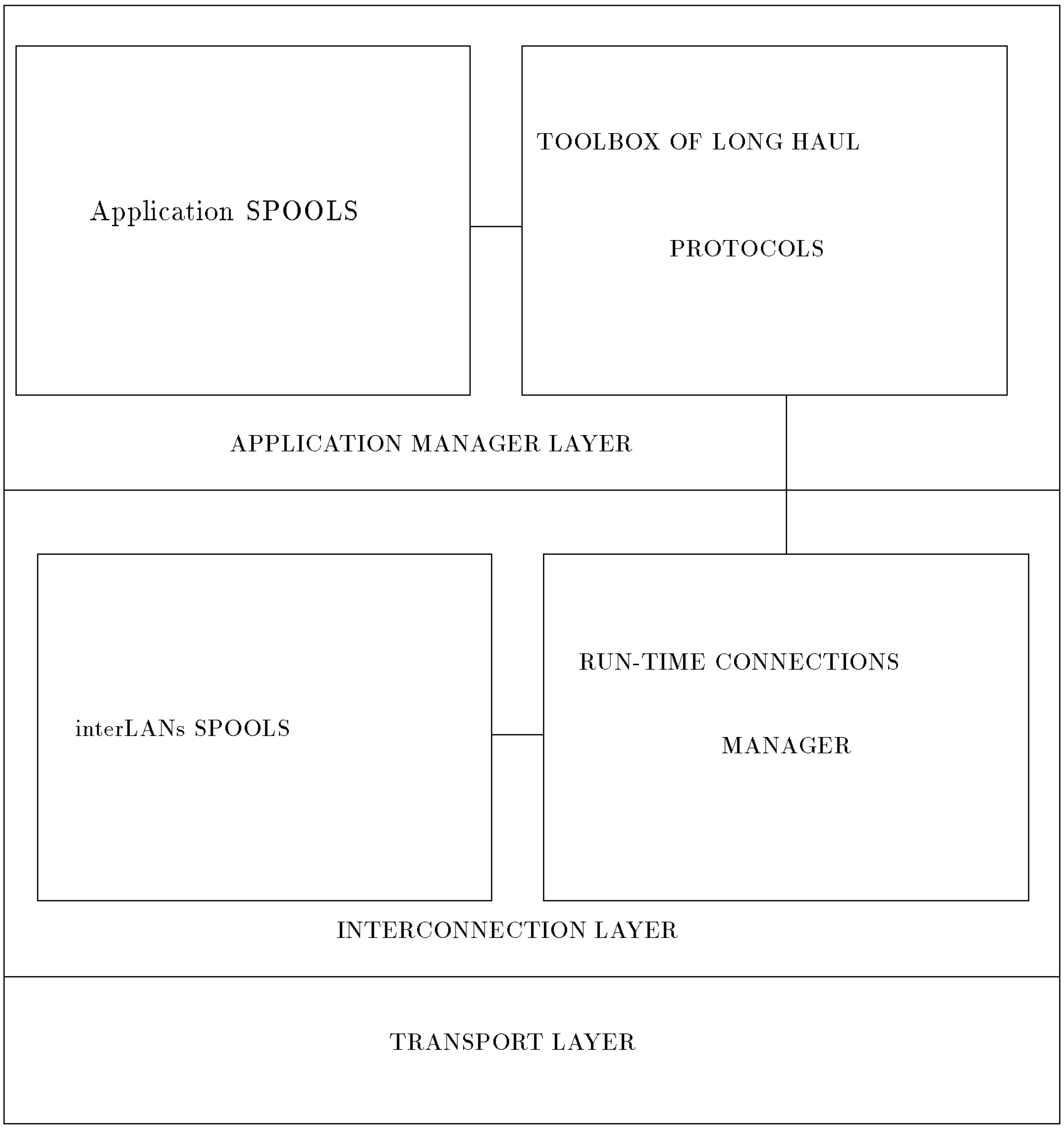

Figure 2: Structure of the long haul package

this (normal) mode of operation, the overhead associated with the spooling mechanisms is small.

4.1.2 Set join primitive

To join a *wSet*, an application invokes the primitive set_join(setName, gpname, clist), where setName is a symbolic name for the application; gpname is the local group name of the joining group; and clist is the list of clusters where components of this application reside.

The call creates and initializes WAN data structures associated with the *wSet* abstraction, if this has not already been done. Initialization includes creating the *wSpool* for the application, and triggers an exchange of messages between the caller's cluster and other clusters listed in the clist argument. The clist associated with a *wSet* is assumed static.

The call also registers the group named gpname as the local representative of the application named setName on the caller's cluster. This registration will trigger delivery of any pending fault-tolerant multicasts if the *wSet* was already active.

4.1.3 Conversations

A *wSet* identifies the full set of related groups making up a wide area application. However, as illustrated in the case studies, within a single application, there may be a need for communication between individual pairs of groups (point-to-point), for multicast to the full set of related groups (global multicast) or between subsets of the full set (restricted multicast). To permit all these kinds of addressing, our wide area system provides support for what we call *WAN conversations*. The mechanism is based on the notion of conversations used in the PSYNC system [8].

A *WAN conversation* is defined by its *participants* (a subset of the *wSet*) and a set of messages exchanged between these *participants*. A programmer creates a conversation and obtains a conversation identifier for it using the call:

$$\mathsf{ConvID} = \mathsf{getConvID}(\mathsf{setName},\ \mathsf{participants})$$

where setName is the *wSet* name, participants is the list of clusters participating to this *conversation*. The ConvID obtained in this manner is used as an argument to the WAN communication primitives shown below.

4.2 Long haul multicast protocols

This sections describes the two WAN multicast primitives supported by our system.

4.2.1 The Per Conversation Causal BroadCAST (pc_cbcast) protocol

Recall that $\prec$ represents the happens before relationship for the system. The pc_cbcast protocol guarantees that, for any pair of messages m and m' belonging to the same conversation, if $m \prec m'$, m will be delivered to each participant in the conversation before m'. During periods when a participant in a conversation is unreachable due to partition, pc_cbcast logs messages atomically in the interLAN spools associated with the channels connecting its sender's with its destination clusters. Similarly, on receiving a message at a remote cluster, if the destination process group is not operational, it is spooled for delayed delivery. We assume both types of spools are replicated for fault-tolerance and that the physical loss of all replicas of an interLAN spool is unlikely. Our software will tolerate failures and recovery of spooling processes but can lose messages if the spool files themselves are corrupted during a crash.

The details of the pc_cbcast implementation appear below. The protocol uses a timestamping scheme similar to the one used in the ISIS fast causal multicast protocol [4]. Under this approach, each message m carries a list of message-id's for messages m' where $m' \prec m$. If, when m arrives, such a message m' is still outstanding, m is delayed until m' has been received. Because our scheme assumes that all forms of LAN failure are transient and that interLAN spools are recovered after failure, m' will eventually be received and m' can then be delivered.

4.2.2 The Per Conversation Atomic BroadCAST (pc_abcast) protocol

The pc_abcast protocol provides that all multicast messages belonging to the same *conversation* will be delivered in the same order at common destinators. Our protocol implements pc_abcast using pc_cbcast:

For each *conversation*, we choose one of the participants to be the coordinator for pc_abcast messages belonging to this conversation. Our current scheme uses the participant with the smallest cluster identifier.

Participants other than the coordinator issue pc_abcast operations through the intermediary of the coordinator. This is done using a pc_cbcast message belonging to the conversation consisting only of the requester and the coordinator.

The coordinator now uses pc_cbcast to send the message to the specified set of participants on behalf of the real sender. Because all of these multicasts originate in a common sender, and pc_cbcast is FIFO, the delivery ordering will be the same at all destinations.

Our protocol does not change coordinator, even during failures. This decision simplifies the implementation, and since LAN subsystems are assumed to isolate the effect of failures, we see little benefit in changing coordinators. In fact, Skeen's work on partitioning suggests that this type of protocol must sometimes block during partitions, even if it has the freedom to change coordinators.

5 The ISIS wide area system

The ISIS long haul package is implemented upon the TCP protocol. In this section we focus on the implementation of the interconnection layer and the pc_cbcast in the ISIS environment. We will also discuss the interaction between this extension and the original ISIS toolkit. Section 5.1 discusses the implementation of the interconnection layer. Section 5.2 discusses the implementation of the pc_cbcast protocol. Finally, section 5.3 discusses the interaction between the ISIS toolkit and this new facility.

5.1 Interconnection of ISIS clusters

The ISIS wide area system is composed of a set of interconnected ISIS clusters. Each ISIS cluster has a unique identifier and name by which it is known to other clusters. The initial configuration of the wide area system is provided in a clusters file that lists, for each cluster, a set of its *access* *points*. Each *access point* is an internet system name or address and TCP port number on which long-haul connection requests from other clusters will be accepted.

Long-haul communication is done by a special WAN service consisting of a process group in each cluster. Between each pair of clusters, this service maintains a single *master connection* at all times. The connection is established between a single, randomly selected member of the cluster on one side and a randomly selected member of the cluster on the other side. The effect of this is to spread the responsibility for handling master connections around the group: if there are 3 group members in a system with 6 clusters, each member will normally handle 2 master connections. Should the process responsible for a particular master connection fail, one of the surviving group members opens a backup connection.

Recall from Sec. 2.1.3 that the interconnection layer must deal with both planned and unplanned communication outages. A control-function interface permits applications to enable and disable communication with one or more clusters. This mechanism is invoked when a controlled partition begins and subsequently re-invoked at the end of each period of partitioned activity. We assume that normal Isis tools can be used to implement such functionality as part of the application layer.

The interconnection layer must also recover from unplanned long haul connection failures, retransmitting messages that were lost due to the failure while also detecting and ignoring duplicate messages that may have arrived over different links. This requires that all members of the long-haul group be kept closely synchronized with the process handling the master connection.

To solve this, we atomically multicast all long-haul requests to the full membership of the long-haul process group. All members update their local states on receiving such a message. Events that should trigger an external action, such as transmitting a message to a remote cluster or forwarding a message to a local process group, are performed only by the process managing the corresponding master connection.

Note that all members of the long-haul group observe all events in the same order. The implication of this is that no intra-member communication is needed to keep the group members synchronized.

On the destination cluster, the member that receives the message multicasts it to all other members of the communication group. Hence, each member learns of the reception of any message within its cluster. The cluster also acknowledges the messages it receives, piggybacking this information on any normal messages sent in the reverse direction.

If a master connection fails, the process that will replace it re-opens the connection, retransmits any unacknowledged messages and then resumes normal service. In the receiving side, the group detects any duplicated message and discards it.

5.2 The pc_cbcast implementation

Our pc_cbcast implementation is similar to the one used to implement the Isis cbcast protocol, but re-engineered in light of the special characteristics of the WAN environment.

5.2.1 The main structures

An Isis application issues a pc_cbcast as follows. First, if the message has a local destination, a copy of this message is multicast to the local group representing the destination *wSet* using the normal Isis cbcast protocol, and spooled if that group is not currently active. Next, if the message has remote destinations, it is multicast to the the long-haul communication group. Before doing this, the sender allocates a descriptor for the multicast, containing the following information:

- A unique message identifier (messID).
- The identifier for the sending cluster (senderID).
- The identifier for the destination cluster(s) (destID).[5]
- An n-bit vector specifying destination clusters that have not yet received copies of this message (DestClusters).

Additionally, each message carries a list of descriptors for predecessors' which have not yet been sent, or have changed since the last time they were sent, to the current destination. These descriptors are sorted so that if m_1, ..., m_k are the messages described in the list, for every $i \in 1 \ldots k-1$, $m_i \prec m_{i+1}$.

The application manager layer of the long-haul service maintains three types of descriptor queues and three types of messages queues for each application (*wSet*).

1. The DESCBUF queue contains pc_cbcast descriptors sent or received by this participant.
2. For each cluster, a queue of waiting descriptors (PRECEDES). The descriptors in these queues will be sent with the next message destined to the associated partner. PRECEDES consists of pointers to the items in the DESCBUF.
3. For each cluster, a queue of descriptors that have been seen previously from that cluster (KNOWN). These queues also point to items in the DESCBUF.
4. A global queue of undeliverable messages (GDELAYED). These are messages for which delivery has been delayed while waiting for some predecessor that has not yet arrived.
5. For each participant, a queue of waiting messages (PDELAYED). This list contains pointers to items in the GDELAYED.
6. A queue of deliverable messages (DELIVERABLE). Once a message is deliverable, it is put in this queue and then later delivered to the application.

5.2.2 The sending and receiving procedures

The sending procedure for a message is as follows. First, a descriptor is allocated. The DestClusters field of the descriptor is set from the participants list in the conversation used. Then, the new descriptor is added to the DESCBUF and to all PRECEDES queues other than the one associated with the sender cluster. Finally, all the descriptors present in the PRECEDES list associated with the destination are piggybacked on the message. Once a message has been sent to a particular destination, these descriptors can be removed from the corresponding PRECEDES list, taking advantage of the failure-free communication channel assumption.

[5]This identifier names a specific cluster in case of point-to-point communication. In case of a multicast, this field will have a distinguished value, and the destinations will be specified in the DestClusters field.

When an application manager receives a message, it first analyzes the list of piggybacked descriptors, as follows. For each descriptor d (starting with the first one), this procedure verifies that the descriptor is *valid*, i.e. that there is no descriptor d' such that $d' \prec d$, d and d' have the same sender, and d' is known but d is not. This test is carried out by search of the sender KNOWN queue. If the descriptor is valid, the receiving procedure looks it up in the DESCBUF. If found, the stored DestClusters information is masked by and-ing it with information in the incoming descriptor. If the descriptor is not found in the DESCBUF, a new item is added to DESCBUF and to the KNOWN list ‘ associated with the original sender of the message to which the descriptor corresponds. A new arrival descriptor is then added to the PRECEDES queue for the destination.

A pc_cbcast message is *deliverable* if all its predecessors have already been delivered. If the arrival message is not deliverable, it is appended to the GDELAYED and PDELAYED queue for senders of its missing predecessors. If the message is deliverable, it is appended to the deliverable queue (DELIVERABLE).

5.2.3 Garbage collection of messages and descriptors

When an application manager receives a valid descriptor for the first time, it clears the corresponding bit in the DestClusters field if the described message is destined to a *participant* located in this cluster, and if the message itself arrived with this descriptor. After a destination has cleared its bit within the DestClusters field of a descriptor, it resends this descriptor to all other partners. One can easily establish that

1. If any participant sees the DestClusters vector associated with some message become zero, then the corresponding message has been received by all its destinations.

2. For any participant, the DestClusters vector field of any descriptor becomes zero in finite time, provided that there exists a minimum level of interaction between each pair of clusters. This is because each time a process resets a bit within the DestClusters field, it resends the descriptor to all other interested partners. Provided that at least one message is sent to any destination after this update, all other partners will see the update.

3. A new arrival descriptor d' is stale and may be discarded if there exists a known descriptor d such that d and d' have the same initial sender and the message described by d' "*precedes*" the one described by d, and d' is not in the DESCBUF queue. This is true for two reasons. First, each descriptor arrives at any destination with the descriptors of messages preceding the one it describes, unless these descriptors have previously been sent to this destination. Secondly, because clusters communicate through fifo channels, the descriptors are received in the order sent. The only case in which if $d' \prec d$ but d' is not in the DESCBUF is when d' has been garbage collected, in the manner described below.

The garbage collection procedure relies on these three properties. Once the DestClusters field of a descriptor associated to some message is zero, the message body itself is garbage collected. However, its descriptor is not garbage collected immediately. A descriptor is garbage collectible if its DestClusters bit vector is zero, and if there is no more links pointing to it from any KNOWN or PRECEDES list. Links from different PRECEDES lists are removed as soon as descriptors are sent to their destinations, as described in the previous section. A link is removed from a KNOWN list when the DestClusters bit vector of the descriptor it points to is zero, and if this link is the head of its list. By delaying the removal of links from KNOWN lists until they are at the head of their list, we prevent the addition of invalid descriptors. Notice that the head of a KNOWN identifies the oldest valid message from the corresponding remote cluster.

5.3 Interaction with the Isis toolkit

The figure 3 summarizes the interface between the long haul package and the Isis toolkit. A client process communicates with the long haul package through the following interface.

```
set_join(char *setName);
bitvec *ConvID = getConvID(char *setName, char **partipantList);
pc_cbcast(char *setName, char *dest, int entry, message *msg, bitvec *ConvID);
pc_abcast(char *setName, char *dest, int entry, message *msg, bitvec *ConvID);
```

The primitives set_join and getConvID have already been described. Notice that in our implementation, the Isis set_join primitive only has one argument. Our initial implementation assumes that each wide area application has components in all Isis clusters; it also assumes also that these components have the same Isis group name everywhere.

The arguments to pc_cbcast and pc_abcast are as follows. setName is the name of the wide area application, which is also the name of the local group representing this application on each cluster. For a point-to-point communication, dest is the name of the destination cluster. If dest is the string "all", this multicast is addressed to the participants in the conversation identified by the ConvID argument. The entry argument specifies the entry point of the application (or more precisely of the groups representing it) to which this message is to be delivered. Finally, msg is the message to be delivered.

To transmit pc_cbcast and pc_abcast requests to the wide area communication service, the Isis abcast protocol is used. This ensures that all members of the service receive these requests in the same order, and hence can assign message identifiers and compute pc_cbcast descriptors without first running a potentially complex protocol. Upon reception on the destination cluster, the wide area package uses the corresponding Isis multicast protocol to deliver the message to the members of the group named setName; (i.e. cbcast for pc_cbcast, and abcast for pc_abcast).

One can show that the end-to-end protocols (i.e. between the client processes and the set of all members) are as defined for the per-conversation causal (resp. atomic) multicast protocol.

The long-haul tool also includes a file transfer interface. To use it, a message is *tagged* with the name of a data file. As the tagged message is transferred over a communication link, the link-level software appends to it the byte stream associated with the data file. On the reception side, the process managing the master link copies this data either to a pre-specified, fixed destination, or to a dynamically selected spooling area. Functions are provided for determining the file name that was used. If desired, a callback is done on the sending side to signal successful completion of the transfer.

6 Performance

Our performance analysis focuses on latency of the long-haul facilities in the case where no communication failure occurs and the participating groups are all operational. We include RPC performance figures, although we should also note that RPC is not entirely meaningful in the long-haul environment because of frequent disruptions in the communication network. For example, when

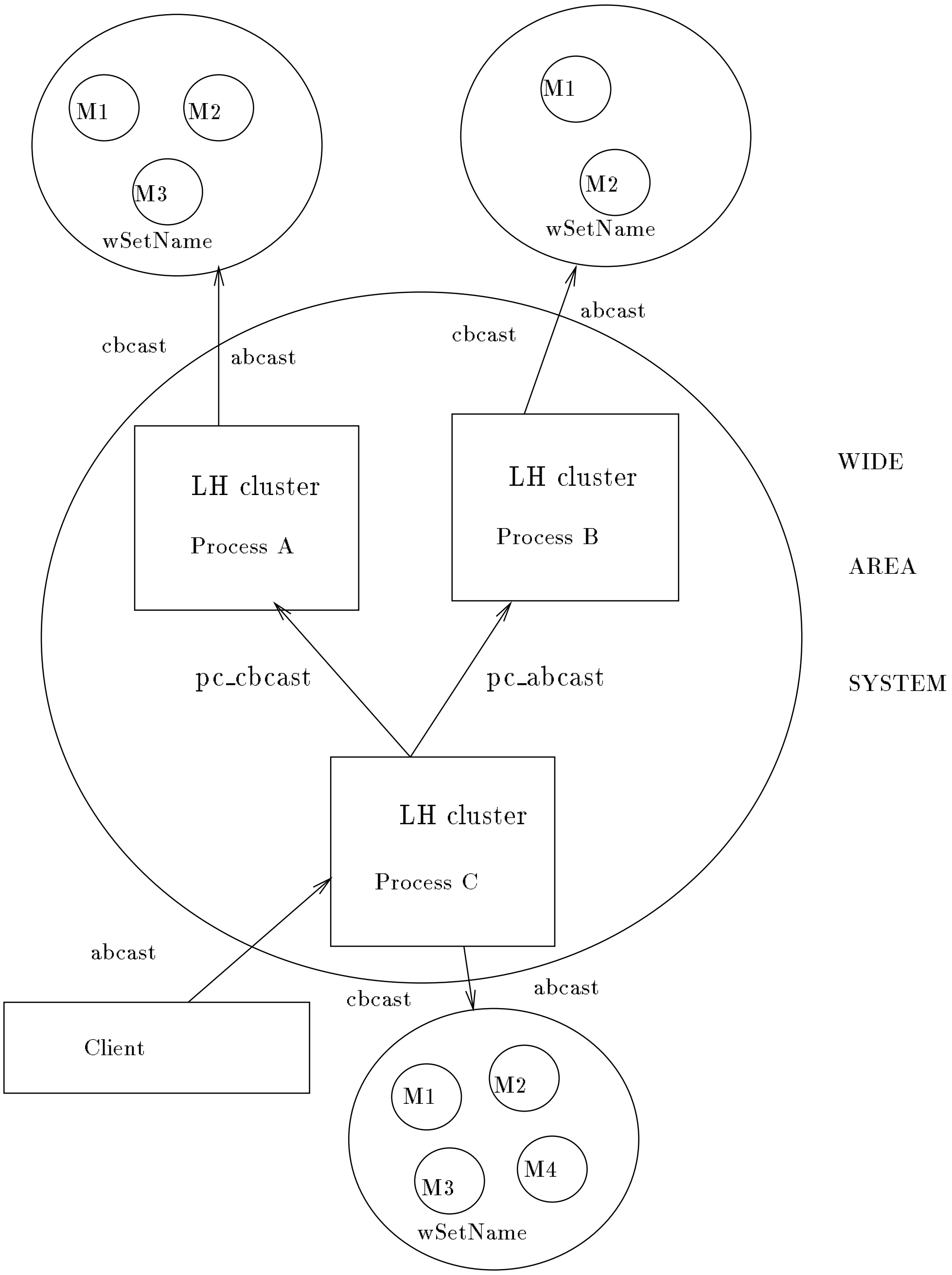

Figure 3: Interaction between the long haul package and the Isis toolkit

IMS was actually used between San Diego and Norway, it was difficult to maintain an open TCP connection for more than 1 to 10 minutes at a time. Application software based on an RPC-style of communication would thus experience frequent timeouts and failures.

We separately evaluated the local delay to initiate a long-haul operation (the time before a client message was logged in the interLAN spool), the delivery delay (the time to obtain a message already logged on reception in an application spool), and the delay associated with transferring logged messages from one cluster to another. We found that the local delay and the delivery delay depend primarily on the Isis multicast delay than on the specific facilities described in this paper. These figures are reported elsewhere [3,4], and have substantially improved in ISIS V2.1. For ISIS V1.3.1, which we used in these tests, the abcast protocol cost approximately 20 to 25ms; for ISIS V2.1, this figure has dropped to less than 12ms. Isis performance impacts primarily on the *WIDE_SYS* figures shown below.

We undertook a more detailed analysis of the inter-cluster transfer rates, measuring the latency imposed by both the long haul transport and the interconnection layers. We also measured the transfer delay of the pc_cbcast protocol. Finally, we measured the intrinsic transfer delay and latency of the pc_cbcast protocol when the logging mechanisms are bypassed; although such a scheme would not be tolerant of communication failures.

All the figures reported here were measured during periods of low system load on a pair of Sun Sparcstation 1's under SUNOS 4.0.3c. The long haul package was run using Isis V1.3.1.

The two remote processes communicate through a TCP connection that was established before we start the timing. In addition, the long haul message sent during the test was pre-allocated. We run the measurements for user data field size equal to 0, 4, 64, 256, 512, 1024.

Within the long haul protocols layer, our tests used the pc_cbcast protocol.

6.1 Long haul transfer delay (Table 1)

In this test, the client procedure sends 99 messages, then waits for a message from the receiver indicating that the the 99th message was received. This gives a good measure of the time needed for to transfer 100 messages to the remote destination. The client repeated this transmission pattern 10 times for each measurement we made.

Table 1 summarizes the results of this experiment. Line TCP gives to the transfer delay seen when two Isis applications communicate directly using a TCP connection. Line LH_CH gives to the transfer delay of the interconnection layer of our wide area package. Line PC_CBCAST provides figures for the pc_cbcast protocol when we bypass the logging mechanisms. Line WIDE_SYS corresponds to the overall wide area system transfer delay, including the hop from the end-user to the wide-area subsystem and the time for remote delivery to the destination program, including all internal spooling and multicast costs. All figures are given in milliseconds, and the standard deviation of each measure is given in parentheses.

6.2 Long haul latency (Table 2)

For this test, the receiver procedure replied to every message it receives, in an RPC style, and the client waited for each reply before sending the next message.

Table 2 synthesizes the results of our measurements. Line TCP gives the TCP latency we measured between two Isis applications that communicated using a normal TCP connection. Line LH_CH gives to the latency of the interconnection layer of our wide area package. Line PC_CBCAST

Sizes	0	4	64	256	512	1024
TCP	2.02(0.02)	2.06(0.02)	2.35(0.03)	2.42(0.02)	2.72(0.02)	3.44(0.23)
LH_CH	9.38(0.49)	12.95(3.24)	9.98(0.29)	12.95(3.24)	10.41(0.36)	12.01(0.40)
CBCAST	10.36(0.37)	13.42(1.38)	11.10(0.57)	10.85(0.26)	12.86(2.26)	14.25(2.14)
WIDE_SYS	94.66(4.84)	103.72(19.75)	103.73(19.16)	128.38(18.47)	171.21(61.08)	131.43(19.82)

Table 1: Long haul transfer delays, source to destination (ms)

Sizes	0	4	64	256	512	1024
TCP	7.38(0.24)	8.31(0.37)	7.94(0.93)	13.15(2.65)	15.03(3.63)	11.37(067)
LH_CH	23.29(1.53)	23.22(1.00)	23.97(0.99)	23.91(1.02)	47.79(3.99)	27.81(1.19)
CBCAST	26.69(1.40)	26.67(1.42)	26.36(0.86)	26.13(0.60)	28.17(1.24)	30.64(1.11)

Table 2: Long haul latency (milliseconds)

provides figures for the latency of the `pc_cbcast` protocol when we bypass the spooling mechanisms.

7 Conclusion

We have reported on a new wide-area communication facility for the Isis system. The system is oriented towards an unusually loosely coupled, asynchronous style of programming, but in which atomicity and ordering properties are nonetheless important determinants of application-level correctness. An implementation of the facility is included as part of the current Isis software release, and is being used in at least one major Isis application, namely the IMS system described in the paper.

8 Acknowledgements

We are grateful to Tom Bache and Mark Watson of Science Applications Internation Corporation, and to Pat Stephenson of Cornell University for their their many comments and suggestions regarding this material.

References

[1] Thomas C. Bache, et. al. The intelligent monitoring system. *Bulletin of the Seismological Society of America*, 80(6): 59–77, December 1990.

[2] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in an unreliable environment. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[3] Kenneth P. Birman et. al. *ISIS – A distributed programming environment (programmers manual).* Department of Computer Science, Cornell University, Rev. 2.1 (Aug. 1990).

[4] Kenneth P. Birman, Andre Schiper and Patrick Stephenson. Fast causal multicast. Dept. of Computer Science TR 90-1105 (*submitted for publication*), April 1990.

[5] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the V Kernel. *ACM Transactions on Computer Systems*, 3(2), May 1985.

[6] Mike Fisher, Nancy Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2): 374–382, April 1985.

[7] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *CACM*, 7(21), July 1978.

[8] Larry Peterson, Nick Bucholz and Richard Schlicting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[9] Frank Schmuck. The use of efficient broadcast protocols in asynchronous distributed systems. *Ph.D. thesis*, Dept. of Computer Science, Cornell University, 1988.

[10] Dale Skeen. Crash recovery in a distributed database management system. *Ph.D. thesis*, Dept. of EECS, University of California (Berkeley), 1982.

[11] Hubert Zimmermann, Marc Guillemont, Gérard Morisset, and Jean-Serge Banino. Chorus: a communication and processing architecture for distributed systems. Rapport de Recherche 328, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), September 1984.