EXPLOITING VIRTUAL SYNCHRONY IN DISTRIBUTED SYSTEMS

Kenneth P. Birman and Thomas A. Joseph

Department of Computer Science,
Cornell University, Ithaca, New York 14853.

Abstract: We describe applications of a *virtually synchronous* environment for distributed programming, which underlies a collection of distributed programming tools in the *ISIS*$_2$ system. A virtually synchronous environment allows processes to be structured into *process groups*, and makes events like broadcasts to the group as an entity, group membership changes, and even migration of an activity from one place to another appear to occur instantaneously -- in other words, synchronously. A major advantage to this approach is that many aspects of a distributed application can be treated independently without compromising correctness. Moreover, user code that is designed as if the system were synchronous can often be executed concurrently. We argue that this approach to building distributed and fault-tolerant software is more straightforward, more flexible, and more likely to yield correct solutions than alternative approaches.

1. A toolkit for distributed systems

Consider the design of a distributed system for factory automation, say for VLSI chip fabrication. Such a system would need to group control processes into *services* responsible for different aspects of the fabrication procedure. One service might accept batches of chips needing photographic emulsions, another oversee transport of chips from station to station, etc. Within a service, algorithms would be needed for scheduling work, replicating data, coordinating actions at physically separate locations, load balancing, and dynamically reconfiguring the system after a component goes off line or comes back on line.

The premise of the *ISIS* project and this paper is that the existing software development methodology is inadequate to address this class of applications. Here, we put forward a new approach that permits applications to be decomposed into orthogonal components that can be treated separately and in a surprisingly "non-distributed" fashion. Our research seeks to provide a *toolkit for distributed programming* to assist in solving those sub-problems that arise most commonly in distributed systems. Each tool consists of a set of subroutines callable from application software, in some cases augmented by a distributed program that maintains persistent state information. Users develop software by interconnecting non-distributed programs using the tools. *ISIS*$_2$ provides tools for initiating asynchronous actions, updating replicated data without blocking, obtaining mutual exclusion using fault-tolerant replicated semaphores, and many others. A distributed program that uses replicated data would consist of a set of conventional programs, each of which performs subroutine calls to the appropriate tool to access their shared state.

The essential issue in designing the toolkit is to ensure that the tools have orthogonal functionality, since it is this aspect that permits the programmer to break up an application into components that can be solved independently and extended gradually into a complete system. A second issue relates to concurrency. In order to make full use of the potential for concurrency available in a distributed system, processes must be able to make local decisions whenever possible, since a process that must interact with others before making a decision would be delayed until they respond. To address these issues we have

This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contracts MDA903-85-C-0124 and N00140-87-C-8904, and by the National Science Foundation under grant DCR-8412582. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specfic permission.

© 1987 ACM 089791-242-X/87/0011/0123 $1.50

developed a new computation model that we refer to as *virtual synchrony*. In a virtually synchronous environment, routines can be programmed and will behave as if distributed actions were performed instantaneously and in lock-step. The physical realization of such an environment can be much more concurrent, however. For example, the replicated update tool mentioned above operates asynchronously. That is, the caller that requested an update may continue computing without waiting for it to complete everywhere, but can be programmed as if updates occur instantaneously. No sequence of actions, even indirect ones, will cause a read that is performed after such an update to be satisfied using a prior value of the updated data item. The tools themselves are implemented using a more primitive communication mechanism that provides *virtually synchronous process groups* [Birman-a].

The notion of providing an idealized distributed programming environment is not a new one [Lamport-a] [Schneider-a]. Similarities exist between our work and that of [Cheriton] (who proposed a system structuring based on process groups), of [Chang] (who gives a protocol for atomic multicast communication) and of [Jefferson] [Strom] and [Peterson] (who develop mechanisms for supporting asynchronous executions that exhibit properties similar to virtual synchrony). Since virtual synchrony combines a notion of atomicity with an ordering restriction, it is also related to transactional serializability, although nothing analogous to the "transaction" exists in a virtually synchronous setting. What we have done in the *ISIS*$_2$ project is essentially to unify these concepts, weakening some aspects that proved to be overly limiting, and optimizing behavior in the common situations that arise when asynchronous computation is desired and failures can occur. The result is a system capable of satisfying even demanding practitioners that is at the same time formally rigorous.

This paper begins by exploring the concept of virtual synchrony and the ways that it is reflected in the interfaces provided by the *ISIS*$_2$ toolkit. We illustrate these mechanisms by examining the internal stepwise development of an *ISIS*$_2$ application and of a typical toolkit routine. We then examine performance issues.

2. Virtual synchrony

2.1. Assumptions

In this work, we assume that a distributed system consists of *processes* with disjoint address spaces communicating over a conventional LAN using message passing. Processes are assumed to execute on computing *sites*. Individual processes and entire sites can crash; the former type of crash is assumed detectable by some monitoring mechanism at the site of the process, while the latter can only be detected by another site by means of a timeout. It is assumed that failing processes send no incorrect messages. Our system tolerates message loss, but not partitioning failures (wherein links that interconnect groups of sites fail). Partitioning could cause parts of our system to hang until communication is restored.

2.2. Subproblems we wish to solve

We now enumerate some of the specific subproblems we wish to solve in this setting; each of these corresponds to a separate tool within *ISIS*$_2$.

Process groups and group communication. It is often desirable to structure a system into "groups" of (possibly non-identical) processes -- such a group might implement a high level abstraction like the emulsion depositing service, or a low level one, like a replicated data item. Ideally, such a mechanism should enable each process to belong to multiple process groups, provide flexible mechanisms for joining and leaving groups, and be inexpensive. Also needed is a facility for communicating with the members of a group while membership is changing.

Deciding how to respond to a request. When a group of processes receives a request, a strategy must be devised for executing it. The process group mechanism should enable a process to respond to a request using only local information (without running any agreement protocol among the group members), if it is practical to do so.

Concurrency. As much as possible, designers will need to exploit the concurrency available in a distributed system, for example by arranging for several processes to take actions at the same time, or to continue executing asynchronously after sending a message to inform other processes of some event (without waiting for that message to be delivered).

Synchronization. On the flip side of the coin, processes executing concurrently will need locking

and mutual exclusion mechanisms to avoid interference between concurrent activities. These mechanisms must also deal with the failure of a process holding a lock or semaphore, and with deadlock detection.

Replicated data. Many applications require efficient mechanisms for replicating data, at the level of individual data structures, and among cooperating but not necessarily identical processes.

Detecting and reacting to a failure. A mechanism is required for detecting failures and informing any interested parties of a failure. For example, when the members of a group are cooperating to respond to a request from a caller, operational members should be informed if a member fails, and the caller should be informed if all members fail. One would also like the assurance that a message from a failed process will never be delayed so long as to arrive after the failure has already been observed.

Dynamic reconfiguration. After a failure, recovery, or in response to certain types of requests or changes in system load, it may be desirable to adjust the system configuration. To be practical, this must impact as little as possible on new and ongoing activities.

Stable storage. If processes need to recover their state after a failure, a mechanism is needed for creating periodic checkpoints or logs that can be replayed on recovery.

Recovery. It should be possible to design software capable of recovering after failures. After a *total* failure, where all the processes that make up an application crash simultaneously, the need is to restart the whole application gracefully using its stable state. The second and more common problem is to recover from a *partial* failure, when a failed process is attempting to recover while the remainder of the system is still operational. Mechanisms are needed for reintegrating such a process into the system, and perhaps for transferring some part of the current system state to it.

Transactions. Applications that manage shared complex disk-based data structures or distributed ones will sometimes need ways to access and update them as a transaction. Even though the focus of $ISIS_2$ is on non-transactional software, such applications will need to be supported, and they should be able to make use of the remainder of $ISIS_2$ as well.

Protection. To the maximum extent that is practical, the $ISIS_2$ system must protect itself and its clients against actions by erroneous clients.

Consistency. Pervading the above discussion is an implicit notion of consistency. Despite the uncertainty of the system state introduced by concurrency and failures, the designer needs to know that there is some sense in which the operational processes in the system will satisfy a global correctness constraint. As far as possible, given locally correct algorithms one would like to know that a globally correct system will result from interconnecting them. In particular, concurrency should not introduce subtle correctness problems, even when processes are sometimes out of synch with other processes they need to interact with.

2.3. Existing methodologies

What makes problems hard to tackle in a conventional environment is the asynchronous propagation of information among processes. In the absence of shared memory, the only way a process can learn of the behavior of other processes is through messages it receives. Since message transmission times vary from process to process, and change with the load on the system, messages relating to a single event may arrive at different processes at different times, and in different orders relative to other messages. Further, the failure of a process can only be detected when a timeout occurs while waiting for a message from it, and hence cannot be distinguished from a transient communication failure or an overload. All this makes it difficult for a set of processes to maintain a consistent view of one another's status, or for them to coordinate their actions efficiently.

We believe that the current distributed programming methodologies are inadequate in light of these concerns. Most distributed systems are based on remote procedure calls (RPC) with timeout for failure detection [Birrell]: a mechanism that provides almost no support for any of the issues cited above. The current trend is to turn to nested transactions [Moss] or atomic actions for purposes of fault-tolerance, but these provide only a limited solution. Transactions facilitate the management of stable storage, but they offer no help in integrating a recovered state with the current state of the rest of the system. In large systems, transactional concurrency control can be overly restrictive: many of the behaviors listed above are inherently nonserializable. Further, long computations tend to lock shared data structures for extended periods, delaying other computations. We claim that transactions are the appropriate mechanism in situa-

tions that involve *short-lived access to shared data stored on a disk.* Alternatives are needed in other situations.

2.4. Virtually synchronous environments

One way out of the problems enumerated in Sec. 2.2 would be to base the system on atomic multicast protocols.[1] A multicast is atomic if all of its operational destinations receive the message unless the sender fails, in which case either all receive it or none does so. Moreover, all recipients see the same message delivery ordering. We need to extend this definition of atomicity to cover the case of a multicast whose destinations include process groups with memberships that may be changing. Such a multicast should logically be delivered to the group membership that applied when it was invoked, but this may not be the one that is current at the time of delivery. We will consequently require that the delivery of an atomic multicast is always completed before a group that forms part of its destinations is allowed to take on a new member. We point out that many existing atomic multicast protocols assume *static* sets of destinations that are known when the protocol is initiated [Chang] [Cristian] [Schneider-b].

We will use the term *synchronous* to a describe an environment in which multicasts are atomic and events such as message deliveries, process and site failures, recoveries, and other events described below, occur in the same order everywhere. In a synchronous environment, mechanisms solving all the problems cited above can be implemented without much difficulty. Processes can easily maintain a consistent view of one another, as each process is always in the same point in its computation as any other. Synchronization, when needed, is simple for the same reason. Process failures can be handled consistently because all operational processes learn of a failure simultaneously, in the same computational step. Unfortunately, this is prohibitively expensive. The problem is that it requires *all* message deliveries to be ordered relative to one another, regardless of whether the application needs this to maintain consistency. The protocols needed to achieve such a strong ordering are invariably costly, both in terms of the number of messages sent and in terms of the time spent waiting for them to terminate.

This leads us to the concept of *virtual synchrony.* The basic idea is to preserve the illusion that events are occurring instantaneously, but to use different communication primitives that enforce weaker delivery orderings in situations where the application or tool is insensitive to the delivery ordering. For example, one could imagine a multicast primitive that delivers messages in the order that the sending process sent them, but is completely unordered relative to multicasts from other origins. A process with private access to a replicated FIFO queue could use this primitive to update it, since updates would be processed in the same order at all copies. On the other hand, if more than one process can perform operations on the queue, this primitive would be inadequate, because updates from different processes might be processed out of order. The advantage of using this primitive in the former case is that it is likely to have a cheaper implementation than a full atomic multicast, and yet gives the degree of synchrony needed for that application.

A virtually synchronous execution is thus characterized by the following property:

> It will appear to any observer -- any process using the system -- that all processes observed the *same events in the same order.* This applies not just to message delivery events, but also to failures, recoveries, group membership changes, and other events described below. As we will see in the next section, this enables one to make a-priori assumptions about the actions other processes will take, and simplifies algorithmic design.

Recall that the actual sequence of events will sometimes differ from process to process in situations where the resulting actions are the same (or semantically equivalent) to those that would have been taken had the event sequences actually been identical. We will exploit this to increase concurrency.

2.5. Other virtually synchronous tools

The above discussion is so focused on atomic multicasting that one might conclude that this is all $ISIS_2$ provides. In fact, we view atomic multicasts as just one of a family of tools that all provide virtually synchronous behavior. For example, there is a tool that supports bulk transfers of information between processes in a way that looks instantaneous. Another

[1] This is a multicast to a set of processes, not a broadcast to all the machines connected to a local network with a hardware broadcast capability. Such hardware might, however, be exploited to optimize the implementation of the multicast protocol [Babaoglu].

tool makes updates to replicated data appear to be instantaneous. The actual implementations of the tools, moreover, are highly concurrent and asynchronous. The main point is that they can be used *as if* they were synchronous. Furthermore, the tools meet our goal of orthogonality. After developing an application using the replication tool, one can extend it using the state transfer tool: these two kinds of "instantaneous" events are guaranteed not to conflict. These and other tools are described in the next section.

3. Virtually synchronous tools

This section reviews some of the tools supported by *ISIS*$_2$, describing both the role of each tool and the sense in which its behavior is virtually synchronous. Our aim in chosing this set of tools was to enable one to develop applications using a small set of tools, and then to add functionality by invoking additional tools, making only minor changes to the existing code. We expect the tools to grow into an extensive collection covering most of the problems that arise commonly in distributed systems. We begin our discussion with the lowest level of the system, which provides communication primitives, and then work up to higher level tools, many of which use these primitives. Except in Section 3.11, we restrict ourselves to tools that are fully operational (as of August, 1987).

3.1. Atomic multicast primitives

The three primitives described below, *ABCAST*, *CBCAST* and *GBCAST* have been described in [Birman-a]. The implementation is faithful to the one in that paper and is not discussed here. Readers familiar with the primitives may wish to skip to Section 3.2.

ABCAST primitive. A commonly occurring situation involves a number of concurrently executing processes that communicate with a shared distributed resource, whose internal state is sensitive to the order in which requests arrive at different components of the resource. For example, concurrent operations on a shared replicated FIFO queue must be received and processed at all copies in the same order. This ordering requirement corresponds to the primitive we call *ABCAST*, which delivers messages atomically and in the same order everywhere. If all requests for queue operations are transmitted using this primitive, the enqueuing operations would look synchronous relative to other such operations on the same queue.

CBCAST primitive. The correctness of a replicated FIFO queue depends on preserving the order of all operations performed on it. Consider, instead, a service that maintains a set of replicated variables on behalf of several clients. Each client has exclusive access to its variables. Although the service is likely to receive requests concurrently from many clients, it is only necessary to preserve the order of requests originating from the *same* client. Clearly, a multicast primitive weaker than *ABCAST* could be used in this case. On the other hand, because of remote procedure calls, a computation could span multiple processes, and hence messages sent by the same client could originate from several different processes. Short of ordering all multicasts, is there a way of characterizing the ordering requirement applicable in this case?

Lamport observed that in a distributed system, the ordering of events is meaningful only when information could have flowed from one to the other through some chain of message transmissions, receptions and intervening local computations [Lamport-b]. It follows that we can define two multicast events to be *potentially causally related* if information about the first could have reached the point where the second was begun before it was initiated there. Notice that by this definition, two multicasts issued by a single computation are always potentially causally related. This leads us to the primitive called *CBCAST*, which guarantees that if any invocations of *CBCAST* are potentially causally related, the corresponding messages are delivered everywhere in the order of invocation. This is a conservative[2] approach to ensuring that any genuinely related operations will be seen in the correct order.

GBCAST primitive. We have arrived at a situation in which applications might be constructed using mixtures of two kinds of multicasts -- *ABCAST* and *CBCAST*. For example, one could use *ABCAST* to obtain a replicated lock on a distributed resource, and once mutual exclusion has been obtained, switch to *CBCAST* when accessing that resource. Some algorithms, however, will perform operations that look instantaneous with respect to both kinds of primitive. This is what the protocol we call *GBCAST* is designed to do. *GBCAST* is used by the system to manage

[2] *CBCAST* is conservative because, were we in a position to exploit still more semantic information, it might be possible to use a weaker primitive. See [Schmuck] for a more sophisticated treatment of this issue.

group addressing, and is available to users as well, for managing *configuration* data structures (see below).

3.2. Process groups and group RPC

Process groups. This collection of tools implements process groups, providing an interface that can be used to join a group, leave a group, and to monitor group membership changes. Each member sees the same sequence of membership changes, and all processes receiving a multicast addressed to the group see the same "current" membership at the time of reception. Moreover, the membership list is sorted in order of decreasing age, providing a natural ranking on the members, and one that is the same at all members. If a process group member combines these properties with knowledge of the algorithms that other members are using, actions taken by the different members can be coordinated using any deterministic rule, without a special exchange of messages. Notice that in a synchronous system these properties are immediate consequences because group membership changes occur instantly and when no messages are being sent to a process group. Thus, the behavior we describe is virtually synchronous.

Broadcasts and group RPC. This facility provides a remote-procedure call interface to the *CBCAST*, *ABCAST*, and *GBCAST* protocols. Each message can be transmitted to a list of destinations; if one of these destinations is a process group, a copy will be delivered to each of its members as described in the previous section. On receiving a message, a process group member can assume that all other current members received a copy too, and process the message accordingly (this does not imply that all recipients process the message; it is always possible for a recipient to crash before being able to act upon a message).

The caller indicates how many responses are desired; this will normally be 0, 1, or *ALL*, although any limit could be specified. If no responses are desired, the broadcast is performed asynchronously[3] and the client is permitted to continue executing. Otherwise, the client specifies an array into which responses can be stored, and a second array into which the addresses of the respondents can be stored. While collecting responses, the system waits until it has the number desired, or until all the remaining destinations have failed.

A *reply* mechanism is used to respond to a group RPC. The reply itself will be transmitted using a multicast protocol, hence copies can be sent to other processes if desired, and we will use this ability below. Superfluous and duplicate replies are discarded silently. It is also possible for a destination to send a *null reply*, indicating that it does not intend to send a normal reply to a message. The null reply mechanism is useful when a group includes extra processes that receive copies of messages to the group but simply log or ignore them, as a standby might do. In this case, the standbys can send null replies and the system will not wait for them even if a client requests replies from *ALL* group members. This makes it unnecessary for a client to know about the existence of the standbys.

3.3. Cooperating to execute requests

ISIS$_2$ provides tools that make it possible to employ any of the popular methods for responding to a request, as well as to create one's own method, depending on the needs of application.

Configuration tool. This tool allows a process group to maintain a configuration data structure, much like the one that lists membership for a process group. The data structure is stored directly in the process group members, hence there is minimal overhead associated with accessing it. As with a group membership change, it will appear that configuration changes occur when no multicasts to the group are pending, hence all recipients of a message will see the same group configuration when a message arrives. If all members use this data structure to decide how to divide up the work, they will make mutually consistent decisions.

Quorum and full replication. Some replicated processing methods, such as the full replication method used in CIRCUS [Cooper] or the quorum methods used in [Gifford] [Herlihy], have straightforward implementations in *ISIS*$_2$. In the former case, the caller waits for *ALL* responses and all recipients respond. If the caller knows the quorum size, Q, it simply waits for Q replies. If it does not know the quorum, it waits for *ALL* replies, and the Q oldest group members (or any other set of Q members that can be identified consistently) reply, giving the value

[3]When messages are being sent asynchronously, it is advisable to invoke the **flush** primitive described in [Birman-a] prior to interacting with the external world or updating stable storage. **Flush** blocks until all asynchronous broadcasts have been delivered, and is called automatically by the tools that manage logs and stable storage.

of Q as part of their reply. Other members send *null* replies. The caller will obtain fewer than Q replies only if some of the processes responsible for executing a request have failed.

Coordinator-cohort tool. The preferred replicated processing method in *ISIS*$_2$ is the coordinator-cohort scheme, whereby the action associated with a request is performed by one group member while others monitor its progress, taking over one by one as failures occur [Birman-b]. The tool is invoked by all processes receiving a request for a computation (normally, all members of a process group). The tool picks the coordinator to reside at the same site as the caller if possible (to minimize latency), and otherwise in a way that will balance load. When the coordinator terminates, a copy of its reply message is sent to the cohorts. Because the multicast used to send this reply is atomic, it reaches the cohorts if it reached the caller. Thus, if the coordinator is observed to fail before receiving the reply, the tool can deduce that the reply was not sent and select a cohort to take over. If a copy of the reply is received, the computation succeeded.

3.4. Concurrency

The primary tool for obtaining concurrency in *ISIS*$_2$ is the asynchronous multicast. One can multicast a request to a set of processes; all will receive the request concurrently and can execute it in parallel. For example, when *CBCAST* is used to asynchronously update replicated information, the caller can pretend that the message was delivered to its destinations at the moment the *CBCAST* was issued. The properties of *CBCAST* ensure that such a caller will not somehow interact with an "out of synch" destination. Thus, there is no need to implement timestamps at the application level, as in [Liskov], where this is done to resynchronize callers and services when asynchronous updates are being done.

3.5. Semaphores

ISIS$_2$ provides replicated semaphores, using a fair (FIFO) request queueing method. If desired, a semaphore will automatically be released when the holder fails.

3.6. Replicated data

This tool provides a simple way to replicate data, reducing access time in read-intensive settings and achieving low-overhead fault-tolerance. The processes that are managing the item supply routines that will update and, if meaningful, perform read-only access to the item. Arguments such as the item name, byte offset, etc. are passed to these routines without interpretation. The client, which may be one of the processes managing a copy of the item, sees an interface exported by the tool, which can be concealed beneath an RPC stub. In an optional *logging* mode, the tool records updates on stable storage, making it possible to reload data after recovery from a crash and to automatically transfer it to a process joining a process group (see Sec. 3.9). In this mode, a checkpointing routine can optionally be supplied; it must be capable of carving the replicated data into some number of chunks (of variable size), and is invoked repeatedly during transfers and to create a checkpoint if the log gets long.

The replication tool is completely general: replicated data could be memory resident, stored on a disk, or could even be computed on request. The tool interface handles the multicasting needed to ensure that the replicated data structure will remain in a consistent state. If the process managing a replicated data structure indicates that it requires a globally consistent request ordering, like the FIFO queue we mentioned earlier, *ABCAST* is used to transmit reads and updates. If the data structure can be updated asynchronously or the caller has obtained mutual exclusion, *CBCAST* is used instead.

3.7. Detecting and reacting to failures

ISIS$_2$ provides a site-monitoring facility that can trigger actions when a site or process fails or a site recovers. Site and process failures are clean events in *ISIS*$_2$: once a failure is signaled, all interested processes will observe it, and all see the same sequence of failures and recoveries. The failed entity will have to undergo recovery even if it was actually experiencing a transient communication problem that looked like a failure. The *ISIS*$_2$ failure detector adaptively adjusts the timeout interval to avoid treating an overloaded site as having failed.

3.8. Recovery and reconfiguration

Recovery manager. This tool will restart processes after they fail, or if a site recovers. The recovery manager runs an algorithm similar to the one in [Skeen] to distinguish the total failure of a process group from the partial failure of a member, and will advise the recovering process either to restart the

group (if it was one of the last to fail) or to wait for it to restart elsewhere and then rejoin. The recovery manager can be used with the replication tool to obtain a simple mechanism for restarting services that maintain replicated data.

State transfer. This tool provides a way to join a pre-existing group of processes, transferring state from the operational processes to the one that wants to join. The application must be able to encode its state into a series of variable sized blocks of data. The tool transfers successive blocks, using *ISIS*$_2$ messages for small transfers and TCP channels for large ones. The transfer is virtually synchronous with respect to incoming requests to the group. Up to the instant before the join occurs, the old set of members continue to receive requests and the new one does not. Then, the join takes place and the next request is received by the new member too, and only after it has received the state that was current at the time of the join. Process migration can thus be performed by starting a process that will join the group and then arranging for some other member to drop out of the group as soon as the transfer completes. Clients will see this as an atomic event. If a state transfer is interrupted by a failure, it is restarted automatically, either from the point of interruption or from the beginning. Most of the tools, such as the configuration tool, the replicated data tool, and the semaphore tool, automatically transfer their internal states when this facility is in use.

3.9. News service

This service allows processes to enroll in a system-wide news facility. Each subscriber receives a copy of any messages having a "subject" for which it has enrolled in the order they were posted. Although modeled after net-news, the news service is an active entity that informs processes immediately on learning of an event about which they have expressed interest.

3.10. Protection

A protection tool is provided that, if desired, will validate all incoming messages using the sender address. Messages that arrive from an unknown or untrusted client will be presented to a user-specified routine that must determine the appropriate action to take based on the sender and the message contents. This works because *ISIS*$_2$ ensures that a sender's address cannot be forged. Group membership changes are similarly validated before a process is allowed to join or to receive a state transfer. Provided that clients work only through the toolkit, *ISIS*$_2$ cannot be corrupted by the actions of an erroneous user program.

3.11. Additional tools

Several tools are now being designed and will be implemented later this year. We plan to add a real time facility to *ISIS*$_2$. The tool would provide for clock synchronization within site clusters, scheduling actions at predetermined global times, and reconciliation of sensor readings (the tool will act as a database, collecting timestamped sensor values and reporting the set of sensor values read during a given time interval). We have also designed a transactional facility, providing a simple subroutine interface implementing the nested transaction constructs **begin**, **commit**, and **abort** [Moss], which the user simply includes in his or her code. Transactional access to stable storage and 2-phase locks will be provided, using the algorithms (and much of the code!) reported in [Joseph] [Birman-b]. Finally, in [Birman-d] we describe a very high level tool that supports bulletin boards of the sort used in many artificial intelligence applications. Unlike the news service, the bulletin board facility is linked directly into its clients and does not exist as a separate entity; it is intended for high performance shared data management. Processes can read and post messages on one or more shared bulletin boards, and these operations are implemented using the multicast primitives.

4. Miscellaneous system-level facilities

The remaining sections of this paper focus on some examples. To understand them, it will be helpful to have a picture of the overall *ISIS*$_2$ architecture, illustrated in Figure 1. As the figure shows, the system is organized around a *protocols process* which implements the multicast primitives, handles process group addressing and does all inter-site communication. This process maintains process group membership views, using a cache for groups not resident at the site. *Client programs* are linked directly to whatever tools they employ. A set of *service processes* handle service-specific databases. Several services exist at each site: the remote execution service, the recovery manager, and the news service.

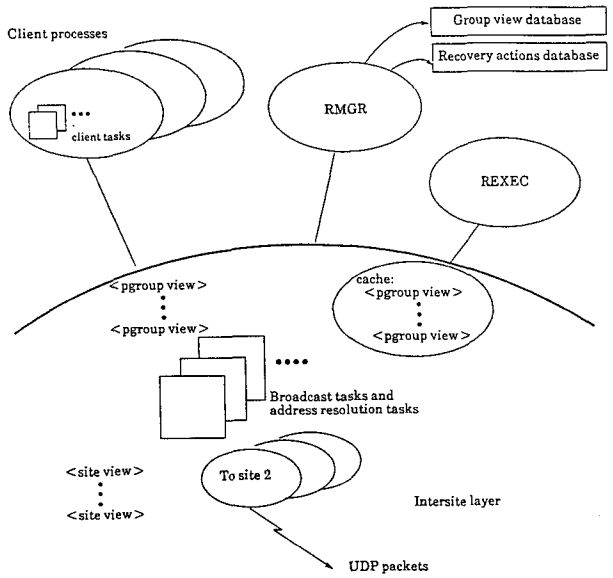

Figure 1: ISIS system architecture

4.1. Run time facilities

All processes in the system have access to the following run time support facilities.

Message subsystem. In *ISIS*$_2$, a message is represented as a symbol table containing multiple fields, each having a name, type, and variable length data. Fields can be inserted and deleted at will, and special system fields carry information such as the address of the sender of a message (this cannot be forged), the session-id number used to match a reply with a pending call, etc. A field can even contain another message.

Tasks. *ISIS*$_2$ implements a light-weight task facility permitting a single process to execute multiple concurrent tasks with no changes to the operating system. Tasks have stack areas of fixed but large size, and are implemented using a coroutine mechanism.

Addresses. *ISIS*$_2$ supports a highly encoded process addressing scheme that represents addresses using an 8-byte identifier. Group addresses can be used in any context where a process address is acceptable, and a way to map symbolic names to group addresses is provided.

Entries. Each process using *ISIS*$_2$ binds routines to any entry point on which it will receive messages. Entry points are known to callers through 1-byte identifiers. Some entry points are *generic* ones used by the toolkit, for example the entry used to join a process group, and the one used by the system to report a group membership change. When a message arrives, a new task is started up corresponding to the entry point in its destination address, and the message is passed to this task for processing.

Filters. Messages arriving in a client are passed through a series of *filters*. A filter is a software procedure that will be given an opportunity to examine each arriving message. For example, the protection facility uses a filter to validate incoming messages. The last filter is the one that creates new tasks.

4.2. Machine Independence and scaling.

ISIS$_2$ currently runs on 4.3BSD UNIX systems (it is operational on DEC, SUN, and GOULD versions of the system). We hope to port it to non-UNIX systems in the future. *ISIS*$_2$ currently implements a non-hierarchical protocol suite. Although these would scale smoothly up to groups of 32 or 64 sites, the extensions reported in [Birman-a] will be needed in much larger networks.

5. A toolkit application

One of our goals in developing the toolkit was to support the stepwise development of distributed application software. To see how the toolkit makes possible such an approach, we now present an example: a "twenty questions" program that was one of the first operational *ISIS*$_2$ applications. The program plays a guessing game in which a caller issues up to 20 questions about an unknown category of objects ("cars", "planes", etc) and then must guess the category based on the answers. Only questions that can be answered yes, no, or sometimes are permitted.

Twenty questions may seem to be a frivolous application, but in fact it is illustrative of a large class of serious ones. Our program works by partitioning a replicated database among several processes and supporting queries on it. It divides the responsibility for handling queries among the processes, which requires that each incoming request be handled consistently. The program supports dynamic updates, tolerates failures, and can dynamically reassign the workload decomposition. As noted in the introduction, an application like this one would be exceptionally difficult to develop in most settings. In *ISIS*$_2$, the first 5 steps described below were completed in one day, required only 450 lines of code (in *C*) for the twenty-questions service and 150 for the interactive front end. This includes all code, even comments, that constitute the two programs, but excludes the

toolkit routines the application employs. It was nearly bug-free from the outset. We now enumerate the stages in developing this program.

Step 1. Non-distributed version.

We started by designing a non-distributed twenty questions program with a static database, consisting of a back-end program that reads the database and a front-end program that interacts with users, the front-end does RPC's to pass queries to the back-end. The database is organized as a relation; the first 11 lines of the one we use for demonstrations are as follows:

object	color	size	price	make	model
car	red	small	5	Wonka	Toy
car	yellow	tiny	6	Matel	Toy
car	black	compact	4995	Hyundai	Excel
car	tan	wagon	6199	Nissan	Sentra
car	green	sedan	10659	Ford	Taurus
car	blue	compact	5799	Honda	Civic
car	white	wagon	15243	Ford	Taurus
car	blue	sport	18499	Nissan	300ZX
car	blue	sport	26775	Porche	944
car	white	sport	35000	Mercedes	300D

A query specifies an item, a value, and a relational operator, for example *price* >9000 or *color* =*red*. The answer to such a query would be *yes*, *no*, or *sometimes*. Obviously, a real database would have several kinds of objects, and the game would start by picking the object using a random number generator. All queries would be implicitly qualified by this (secret) number.

Implementing this program in the *ISIS*$_2$ system is straightforward. A main procedure initializes the program (by reading the database), declares the entry that will respond to queries, and then runs the lightweight task subsystem. As each query arrives, a lightweight task is created to respond to it.

Step 2. Distributed version.

A distributed twenty questions program would replicate the database among members of a process group that makes up the twenty questions "service." Say that there are *NMEMBERS* such processes. There are two options. We could divide the work *vertically*, with each process being responsible for one or more *columns* of the database, or we could do so *horizontally*, with each process being responsible for one or more *rows*. We decided to provide both options, and to extend the query interface to specify which option is to be used. A *vertical mode* query looks just like the ones described above. We adopted the rule that a query referencing column *C* of the database should be handled by member *C mod NMEMBERS*. A *horizontal mode* query is prefixed by a *, e.g. **price* >9000. All the members respond to such a query, with member *M* basing its response on the rows *R* in the database satisfying *R mod NMEMBERS* =*M*. For the above database, if *NMEMBERS* =5, the query **price* >9000 would return the following set of replies:

no	sometimes	sometimes	sometimes	yes

Notice that both kinds of query require a well known ordering on the members of the service.

This extension requires minor changes to the front end program, since it must know how many replies to wait for, viz. 1 in the vertical case, and ALL in the horizontal case (or *NMEMBERS*, if this is known). The extension to the back end program involves adding an argument to the program which, when the program is run, indicates if it should "join" the service or "create" it. The creator first reads the database and creates a process group with symbolic name "twenty". A joining member calls the toolkit routine *join_and_xfer(gid,credentials)* which requests permission to join the specified group (the gid is obtained by calling *pg_lookup("twenty")*). The current state of the group is then transferred to the process that is joining -- in our case, the contents of the database.

Each time a process joins the group or fails, the operational members will need to know about this. Hence, all members *monitor* the membership data structure. This is done by a call to a system procedure *pg_monitor(routine)*, where *routine* is the procedure to invoke each time such a change occurs. Because members are listed in order of decreasing age within this structure, and all see the same sequence of changes, and see those changes in the same order relative to arriving requests, a member's index in this list can be taken as its member number. By so doing, each incoming request can be handled in a consistent manner by all the members, provided that *NMEMBERS* processes are actually operational.

This solution assumes that *NMEMBERS* processes are operational. In a vertical mode query, if fewer than *NMEMBERS* processes belong to the group when it arrives, a caller, who will have requested one response, might get no responses and hang if the process responsible for sending the response fails. In our version, we corrected this problem by having non-respondents send *null replies*, thus informing the system that they will not send a true reply to the mes-

sage in question. Instead of hanging, the caller will now obtain an error code from the multicast it used to issue the query, and will have to reissue its request. We could also have had the respondent send copies of its replies to the other members of the service, using an approach not unlike the coordinator-cohort one described earlier. However, this approach would be more complex.

A different kind of incorrect behavior occurs if a a horizontal query is handled using the above algorithm when the number of processes drops below *NMEMBERS*. Here, the caller will not get the correct number of responses, and will thus only learn about some rows of the database. In our solution, the caller iterates until it receives the expected number of responses. A more complex alternative would be to use a coordinator-cohort scheme under which some representative of the service would compute and return the entire vector of responses.

Step 3. Automatic member restart.

An easy extension to the above solution is to have the oldest member of the service start new members up at an appropriate site until the number of operational ones reaches *NMEMBERS*. If the oldest member fails while doing this restart, a surviving member could take over and reissue the restart. Notice that this involves a potential race condition that could result in extra group members beyond the number intended. This can be corrected by having cohorts spy on the restart process, but we chose not to do so, for reasons described below.

Step 4. Hot standby processes.

The extra group member "problem" can be turned to our advantage. The idea is to have *NMEMBERS* + *NSTANDBY* processes (or more) operational group members whenever possible. Standbys would join the group, but send *null replies* to all incoming requests, thus a client will be oblivious to their existence. On the other hand, should a member fail, the standbys will recompute their ranking along with all the other members, and decide whether to function as a real member. This results in a very rapid transfer of responsibilities.

Step 5. Dynamically updating the database.

Having arrived at a workable distributed twenty questions program, we can now extend it to support dynamic updates to the database. One could make the rule that only existing members can issue updates, or that only specially designated clients can do so (this can be enforced using the *ISIS*$_2$ protection tool), or that any client can do updates.

Clearly, we need to arrange for updates that are virtually synchronous relative to queries, hence we must pick the appropriate protocol for sending queries and updates. One option is to implement both queries and updates using *ABCAST*. The alternative is to implement queries with *CBCAST* and updates with *GBCAST*, or *vice versa*. The choice should be based on the relative frequency of these operations. For example, if it can be predicted that most requests will be queries, one would use *CBCAST* to transmit queries, and *GBCAST* for updates. This is how our version works. Having made this decision, one might want to use the replicated data tool to maintain the twenty questions database, specifying the kind of multicast to use for updates and queries ("read" operations). The changes needed to make this conversion are minimal.

Step 6. Restarting from total failures.

Our solution is tolerant of partial failures, but not total ones. An easy way to extend it would be to activate the logging option in the replicated data tool, which will now maintain checkpoints and logs from which the database state could be recovered. One must also register the twenty questions service with the "recovery manager" at those sites where the service can be restarted after failure, and call the log-recovery routine during recovery, when the original version of the program would have read the database from disk.

Step 7. Dynamic load balancing.

If desired, it would be straightforward to use the configuration tool to change the rule for assigning numbers to members at run time. Such a change might be used to dynamically shuffle the members when a site becomes overloaded and unresponsive (an overloaded member could also just drop out!).

Summary.

Virtual synchrony was useful in several ways in the above solution. The most obvious benefit was the clean decomposition of this distributed program into aspects that could be solved relatively independently from one another. Virtual synchrony also permitted us to design the distributed algorithm using simple assumptions about how a *set* of processes would react to an event that all observe. For example, we did this when we based the response of member *M* on the

value of M: obviously, such an approach only works if each process knows its relative number and the numbering is the same when each sees a given request. We were able to write a fault-tolerant distributed program in one day. When run on 4 SUN 3/50 workstations using a 10-Mbit ethernet and with members at all sites, it supports an aggregate of 30 queries or 5 replicated updates per second. We know of no alternative distributed programming methodology in which this would have been possible.

The solution also illustrates some of the limits to the methodology in its present realization. For example, if a process takes an external action after receiving a message, it is hard to deduce the status of the action if a failure occurs before the action completes. Eventually, we hope to identify paradigms for problems like this, and to package solutions as tools. Moreover, correct behavior of the twenty-questions service when dynamic updates are being done requires that the appropriate broadcast primitive be used by clients when transmitting update and query requests. A programming error in one of many clients could violate such a rule, affecting other clients. A "type checking" mechanism seems to be needed for verifying the compliance of clients with the requirements of services they exploit.

6. Inside the coordinator-cohort tool

This section focuses on the internal structure of the coordinator-cohort tool. It is a relatively simple tool, and we present it primarily to demystify the internals of the toolkit. A seemingly more complex tool, the state transfer facility, is basically just an encapsulation of this method into a special interface.

As described in Section 3, this tool enables a group of processes to use the coordinator-cohort strategy to respond to a message sent to the group by a caller. This approach is meaningful only when more than one member of the group is capable of performing the action requested by the caller, so that at least one cohort can take over should the coordinator fail. The caller simply does a group RPC, and waits for one reply. When the group members receive a message, they each use the same deterministic algorithm to determine a subset of the group members, *plist*, that will actually participate in this coordinator-cohort computation. This list depends on the action to be taken, since some members may be incapable of performing some requests (say, if they do not have access to necessary data). The members in *plist* then each call the toolkit routine

coord_cohort(msg, gid, plist, action, got_reply),

where *msg* is the incoming message, *gid* is the group-id for the group, *action* is the routine that processes the request, and *got_reply* is a routine that, in a cohort, will be called when the coordinator completes its action and replies to the caller. Non-participants issue null replies to the request.

The toolkit routine itself behaves as follows. When called, it examines *msg* to determine the site-id of the caller. It then calls *pg_lookup (gid)* to find the current membership of the group, and scans *plist* to find an operational process that resides at that site. If there is one, it is assumed to be the coordinator for this computation (if there is more than one such process, the first is chosen). If there is no process at that site, the caller's site-id is used as a "random" index into *plist*, and the first operational process, in a circular scan, is chosen. Notice that because all the participants use the same plist and see the same group membership, all will agree on the same value for the coordinator, without any additional communication among the group members. The other processes in *plist* are the cohorts, and the remaining members of the group are non-participants.

If a member determines that it is the coordinator, it then calls the routine *action*. When it returns, it multicasts the result not just to the caller, but also to the generic entry point *GENERIC_CC_REPLY* in each of the cohorts. The computation then terminates in the coordinator.

The cohorts, meanwhile, call the routine *pg_monitor (gid)* to monitor the status of the group. Should the coordinator fail before sending a reply, all cohorts learn of this and, again without interacting, use the same algorithm as above to pick a new coordinator and monitor its progress. If the coordinator succeeds in sending a reply to the caller, the *GENERIC_CC_REPLY* entry in each of the cohorts will be called. It first deactivates the monitor, then calls *got_reply*, passing a pointer to the result and its length as arguments. This terminates the cohort algorithm.

What about the case where all recipients fail before the computation terminates? Here, the caller will receive an error code, since the group RPC will detect that no possible respondents are still operational. Because non-participants send null replies, this works even when a subset of the group members

TABLE I -- MULTICAST OVERHEAD FOR SELECTED TOOLS

Tool	Description	Multicasts required
Group RPC		
nreps=mcast(dests,msg,nwant,answers,who)	Multicast, collect *nwant* replies	*See Figure 2.*
reply(msg,answ,alen)	Normal or null reply to *msg.*	1 async *CBCAST* (1 dest)
reply_cc(msg,cc_dests,answ,alen)	Reply, with copies.	1 async *CBCAST*
Process groups		
gid=pg_create("symbolic name")	Create process group	1 local RPC
gid=pg_lookup("symbolic name")	Lookup group address	1 local RPC [+ 1 CBCAST, 1 reply]
pg_addmember(who,gid)	Add member (done by member)	1 *GBCAST*
pg_leave(gid)	Leave group	1 *GBCAST*
pg_join(gid,credentials)	Request to be added	1 *CBCAST*, 1 pg_addmemb, 1 reply
pg_kill(gid,signal_no)	Send UNIX signal	1 *ABCAST*
pg_monitor(gid,mroutine)	*mroutine* monitors membership	1 local RPC per change
pg_msg_verify(vroutine)	*vroutine* validates messages	*No cost*
pg_join_verify(vroutine)	*vroutine* validates joins	*No cost*
State transfer		
join_and_xfer(gid,credentials,sroutine)	Join, *sroutine* accepts state	1 pg_join + 1 TCP transfer
Coordinator-cohort		
coord_cohort(msg,gid,plist,action,got_res)	See section 6.	1 *CBCAST* to invoke, 1 to reply
Replicated data		
update(gid,args)	Update replicated data	1 async *CBCAST* or 1 *ABCAST*
read(gid,args)	Read-only access by manager	*No cost*
read(gid,args)	Read-only access by other clients	*CBCAST* + 1 reply
Synchronization		
P(gid,sname,free_on_failure)	Obtain mutual exclusion	1 *ABCAST*, all replies
V(gid,sname)	Release mutual exclusion	1 async *CBCAST*
Configuration		
conf_update(item,value,len)	Update configution	1 *GBCAST*
conf_read(item,&value,&len)	Read configuration	*No cost*
News		
subscribe("subject",read_routine)	Register with service	1 local RPC per posting
post_news("subject",msg)	Post a news message	1 async *CBCAST* or *ABCAST*

run the algorithm. Finally, we note that the tool can be invoked reentrantly, provided that appropriate care is taken in the action routine if the computation will require mutual exclusion on any resources.

The cost of the approach is low. Instead of an RPC to the single destination that will respond, the caller used a broadcast. However, the caller will often have received its reply and resumed computation before the original RPC even reaches the remote cohorts, since local communication is faster and the tool is biased towards picking a local coordinator. Thus, any overhead associated with the tool is primarily a background one.

7. Performance

Table I summarizes communication overhead, in multicasts, of the major toolkit routines cited in Section 3. Figure 2 shows the throughput in bytes per second for asynchronous *CBCAST's* (where the sender continues execution without requesting a reply), and the latency *seen by the sender* for *CBCAST*, *ABCAST* and *GBCAST* invocations in which one reply is needed and comes from a local process. This latency measures the delay between when the sender invokes the primitive and when the desired reply is received. Except for *CBCAST*, the primitives give similar behavior when all destinations reply. Asynchronous multicasts and multicasts with a local destination resulted in much more efficient CPU utilization: loads of 95% to 98% were observed on the sending site in these tests, compared with 30% to 35% when running a protocol like *ABCAST* that must wait for messages from remote sites. The remote sites, if otherwise idle, typically showed loads of 20% or less. The sharp rise in latency between message sizes of 1kbytes and 10kbytes occurs because large inter-site messages are fragmented into 4kbyte packets.

Figure 3 focuses on the actual costs associated with sending an *ABCAST* in the system. The figure reveals just how expensive message passing can be, in comparison with all other aspects of a distributed protocol. The link delays shown are for a *single traversal of the link*: 10ms to traverse a link within a site, and 16ms to send an intersite packet. Thus the latency before an *ABCAST* delivery occurs at a remote destination is 70ms -- 3 inter-site messages are sent. *CBCAST* sends 1 inter-site message, and *GBCAST* sends 3 or 5, depending on how it is used.

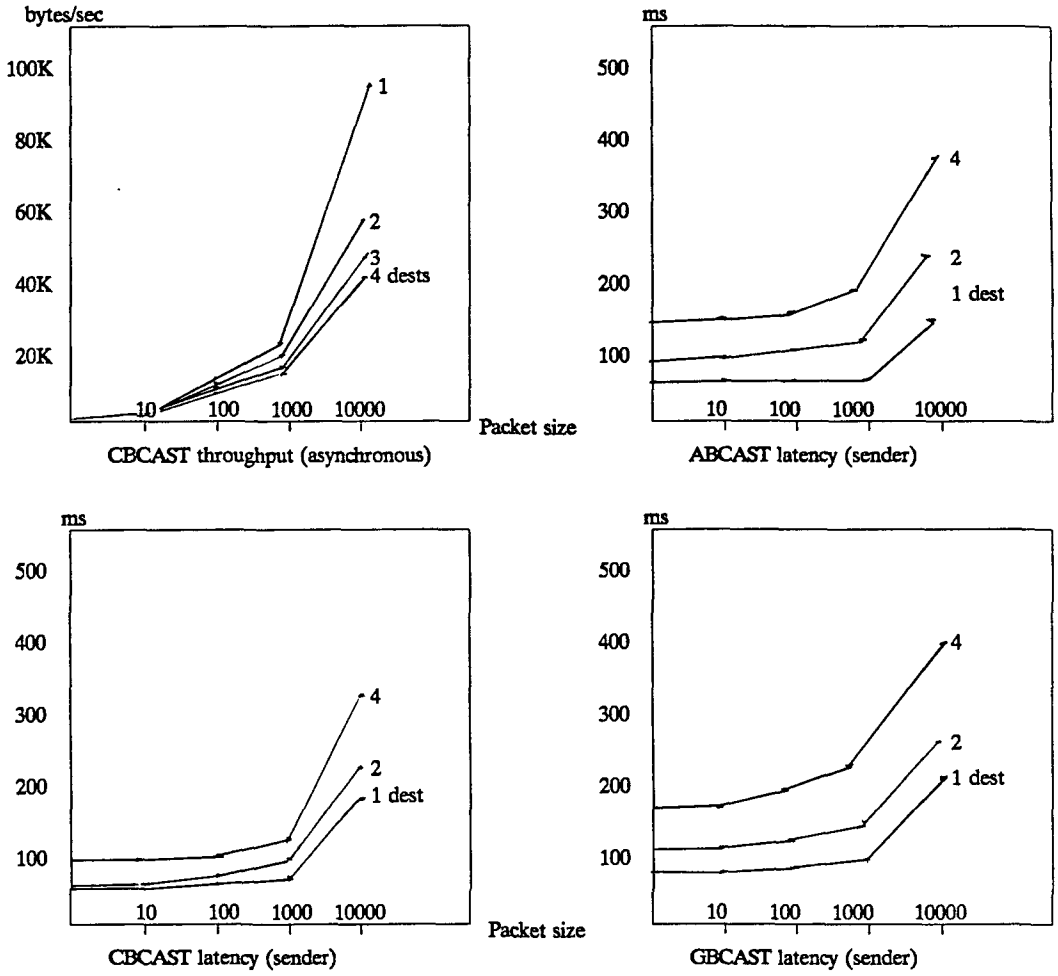

Figure 2: Throughput for broadcast primitives

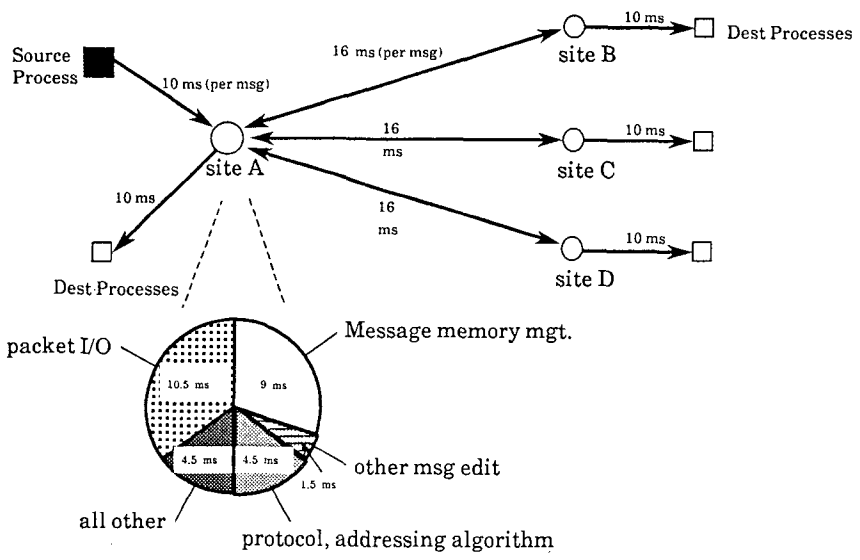

Figure 3: Breakdown of ABCAST execution time

In the future, we plan a much more detailed study of performance, including a study of how the protocols will perform on a system subjected to a uniform load from multiple sites, and how the system performance changes with scale. The initial version of the system has not been operational long enough to permit careful tuning, hence the figures reported above should be understood to be preliminary ones, and are likely to be reduced by optimizations.

8. Status

ISIS$_2$ has now been operational for six months, and is increasingly robust. Working in collaboration with other academic researchers at Cornell and with industrial research and development teams, we are now beginning to develop *ISIS*$_2$ based application software. Nonetheless, many questions remain open, and substantial changes and extensions to the system will be needed before we consider it complete. For example, although the present system is clearly capable of addressing many aspects of the factory automation example (Sec. 1), it remains to be shown that a very large system could really be built using our approach. A pragmatic problem that this raises is that *ISIS*$_2$ will have to coexist with many existing systems, such as the Manufacturing Automation Protocol (MAP), with a variety of databases, and may have to be ported to different kinds of hardware.

At a conceptual level, we are just learning how to infer the choice of protocol from context [Schmuck]. We have largely overlooked real time issues, and extremely demanding real time scheduling constraints are probably incompatible with the *ISIS*$_2$ system. Likewise, the most appropriate way to deal with network partitioning remains a pressing problem.

Despite these limitations, we are convinced that the virtually synchronous approach represents a conceptual breakthrough. Having tried to build robust distributed software using other methodologies and failed, we have now succeeded using this approach. As this technology becomes widely available and the remaining limitations are overcome, it could fundamentally change the way we formulate and solve distributed computing problems.

9. Acknowledgements

Yu-Jen Hsiao undertook the performance studies reported above. In addition to the theoretical work cited earlier, Frank Schmuck implemented the recovery manager currently used in *ISIS*$_2$ and has become increasingly involved in all aspects of system design. Keith Marzullo, Sam Toueg, and John Warne all made insightful suggestions about the virtual synchrony approach, for which we are grateful. Finally, we thank the SOSP program committee. In particular, Ozalp Babaoglu and Alfred Spector have provided invaluable guidance and assistance throughout the revision process, for which we are deeply indebted.

10. References

[Babaoglu] Babaoglu, O., and Drummond, R. Streets of Byzantium: Network architectures for fast reliable broadcasts. *IEEE TSE* **SE-11**, 6 (June 1985), 546-554.

[Birrell] Birrell, A., Nelson, B. Implementing remote procedure calls. *ACM Transactions on Computer Systems* 2, 1 (Feb. 1984), 39-59.

[Birman-a] Birman, K. and Joseph, T. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems* 5, 1 (Feb. 1987).

[Birman-b] Birman, K. Replication and fault-tolerance in the ISIS system. *Proc. 10th ACM SIGOPS Symposium on Operating Systems Principles.* Orcas Island, Washington, Dec. 1985, 79-86.

[Birman-c] Birman, K., Joseph, T. and Schmuck, F. ISIS System Documentation, Release I. Available as TR-87-849, Department of Computer Science, Cornell University, July 1987.

[Birman-d] Birman, K. and Joseph, T. Programming with shared bulletin boards in asynchronous distributed systems. Dept. of Computer Science TR-86-772, Cornell University (August 1986; Revised December 1986).

[Chang] Chang, J, Maxemchuk, N. Reliable broadcast protocols. *ACM Transactions on Computing Systems* 2, 3 (Aug. 1984), 251-273.

[Cheriton] Cheriton, D. and Zwaenepoel, W. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems* 3, 2 (May. 1985), 77-107.

[Cooper] Cooper, E. Replicated distributed programs. *Proc. 10th ACM SIGOPS Symposium on Operating Systems Principles.* Orcas Island, Washington, Dec. 1985, 63-78.

[Cristian] Cristian, F., Aghili, H., Strong, R., Dolev, D. Atomic broadcast: From simple message diffusion to Byzantine agreement. IBM Technical Report RJ 4540 (48668) 12/10/84.

[Gifford] Gifford, D. Weighted voting for replicated data. *Proc. 7th ACM SIGOPS Symposium on Operating Systems Principles.* December 1979.

[Herlihy] Herlihy, M. Replication methods for abstract data types. *Ph.D. thesis*, Dept. of Computer Science, MIT (LCS 84-319), May 1984.

[Jefferson] Jefferson, D. Virtual time. USC Technical report TR-83-213, University of Southern California, Los Angeles, May 1983.

[Joseph] Joseph, T. and Birman, K. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Transactions on Computing Systems* 4, 1 (Feb. 1986), 54-70.

[Lamport-a] Lamport, L. Using time instead of timeout for fault-tolerance in distributed systems. *ACM TOPLAS* **6**, 2 (April 1984), 254-280.

[Lamport-b] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *CACM* **21**, 7, July 1978, 558-565.

[Liskov] Liskov, B., Ladin, R. High Available Distributed Servers and Fault Tolerant Garbage Collection. *Proc 5th ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, Aug. 1986, 40-51.

[Moss] Moss, E. Nested transactions: An approach to reliable, distributed computing. Ph.D. thesis, MIT Dept of EECS, TR 260, April 1981.

[Peterson] Peterson, L. Preserving context information in an IPC abstraction. *Proc. 6th Symposium on Reliability in Distributed Software and Database Systems*, March 1987, 22-31.

[Schneider-a] Schneider, F. Synchronization in distributed programs. *ACM TOPLAS* **4**, 2 (April 1982), 179-195.

[Schneider-b] Schneider, F., Gries, D., Schlicting, R. Reliable broadcast protocols. *Science of Computer Programming* **3**, 2 (March 1984).

[Schmuck] Schmuck, F. Picking the cheapest broadcast protocols in a distributed program. Ph.D. thesis, Cornell Univ. Dept. of Computer Science, (expected) Dec. 1987.

[Skeen] Skeen, D. Determining the last process to fail. *ACM Transactions on Computing Systems* **3**, 1, Feb. 1985. 15-30.

[Strom] Strom, R. and Yemini, S. Optimistic recovery in distributed systems. *ACM Transactions on Computing Systems* **3**, 3 (April 1985), 204-226.

ıaraa.com