AVAILABILITY AND CONSISTENCY IN A PARTITIONABLE LOW BANDWIDTH NETWORK

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Takako Matoba Hickey

August 1998

AVAILABILITY AND CONSISTENCY IN A PARTITIONABLE LOW BANDWIDTH NETWORK

Takako Matoba Hickey, Ph.D.

Cornell University 1998

The Internet is a partitionable low bandwidth network. Two computers connected by the Internet could temporarily become unable to communicate for various reasons, from a failure at either end, to overloading of an intermediate node connecting the two, to an out-of-date routing table. The bandwidth of the Internet is relatively low: its data transfer speed is inadequate for applications that need to access very large data objects. Since its introduction the Internet kept a much faster rate of increase for the data available than for the data transfer speed. This trend is not likely to change any time soon.

This thesis presents techniques to deal with availability and consistency in partitionable low bandwidth networks, and it presents a service suitable for use in such networks.

The first part of the thesis presents replication techniques suitable in a partitionable network. A partitionable network puts goals of availability and consistency at odds. Making objects available in a partitionable network requires replication; yet modifications of replicated objects during a partition can introduce inconsistencies. We show various replication techniques that provide continuous availability while managing inconsistencies among replicas.

The second part of the thesis describes the PEX system, an example of a service that instantiates these abstractions. PEX is an execution service that facilitates processing of remote data. It allows computations to be performed near data, so that data need not be copied over the network. It provides a session interface which facilitates the management of related computations operating on scattered data. Internally, PEX replicates various information, including sessions, using some of the techniques introduced in the first section of the thesis. This permits the system to keep computations running in the face of partitions.

The third part of the thesis gives two example applications that use PEX (a distributed shell and a parallel make) and reports the performance of the PEX system.

BIOGRAPHICAL SKETCH

Takako Matoba Hickey spent her childhood in Tokyo, London, New York, and Los Angeles. She received her B.S. in Engineering and Applied Science from Caltech in June 1986, and her M.S. in Computer Science from Stanford in October 1987. She worked for Bellcore from April 1986 to May 1995.

ACKNOWLEDGEMENTS

Thanks to all those who gave me strength during my journey.

Special thanks

to Robbert van Renesse for his insights, kindness and patience,

to Fred B. Schneider for his discipline,

to Ken Birman for his support,

to Mark Hayden for his energy,

to Gary Adams for his compassion, and

to Jason Hickey for his love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

This thesis addresses the problems of availability and consistency in a partitionable low bandwidth network.

A prime example of a partitionable low bandwidth network is the Internet. Two computers connected by the Internet could temporarily become unable to communicate for various reasons, from a failure at either end, to overloading of an intermediate node connecting the two, to an out-of-date routing table.

Satisfying both availability and consistency goals in presence of partitions is difficult. Making a service available in spite of partitions requires replication. A service can be rendered as long as some replica is available. However, if a replica is modified while the network is partitioned, different replicas can become inconsistent.

Different applications cope with the partition problem in different ways. For some applications, availability is absolutely crucial. Such applications opt to sacrifice consistency in favor of availability. An example of such an application is an air traffic control system, consisting of processors each directing traffic at a specific area. A control tower must continue making decisions even if it cannot communicate with other processors. Waiting for a partition to heal (and not doing anything) could cost lives of people aboard airplanes.

Other applications may favor consistency over availability. For example, consider a scenario where a laptop user wishes to edit a file. The user may copy data over the network, disconnect the laptop from the network, and edit the copied data on the laptop. Once done with editing, the user can reconnect the laptop to the network and transfer back the file. This works well if the file being edited is exclusively owned by the user. Nobody besides the owner should modify the file and the copy of the file in the partition the owner is not connected is not accessible by him or her. So the system can prevent inconsistencies by allowing only the copy of the file accessible by the owner writable at a time.

Such copying data is not always practical nor possible in the Internet. This is because the Internet is relatively low bandwidth—its speed is such that data cannot be made available to all computations that need it. The amount of data stored in the Internet is growing at a much faster rate than the speed of the Internet. In 1981 (and up to quite recently) the typical technology for a local area network was a 10 Mbit Ethernet. Today, the typical technology for a LAN is a 100 Mbit Ethernet, an order of magnitude increase in the speed. The number of hosts in the Internet has grown from 213 in August 1981, 5089 in November 1986, one million in January 1993, and almost 30 million today [Lot92, Wiz]. This is a five orders-of-magnitude increase. Not only has the number of access points—or users—has grown, but the amount of data stored and access to these data by users have also increased. Users, companies, and institutes have all made available a wealth of data, ranging from home pages to digital libraries [PCGMW98, Moe98, GMS+98, WNMMC98] to observational data [FPSS96, GMPS96, BKK+96, FHS96]. Individual users access the Internet today far more frequently than they did a decade ago.

This trend is not likely to change any time soon.

1.1 Thesis Statement, Goals, Contributions

The gap between the growth in the demand for data and the speed of the Internet will likely continue to widen. The current practice of copying data to do local processing cannot be sustained. The goal of this thesis is to identify techniques useful in providing services in a partitionable low bandwidth network.

In attempting to achieve this goal, this thesis makes two contributions:

- A survey and analysis of replication techniques suitable for a partitionable network.

- Demonstration of a service for a partitionable low bandwidth network.

The first part of the thesis presents replication techniques suitable in a partitionable network. A partitionable networks puts the goals of availability and consistency at odds. Making objects available in a partitionable environment requires replication. Yet modifications of replicated objects during a partition can introduce inconsistencies. We survey replication techniques that provide continuous availability while managing inconsistencies among replicas.

The second part of the thesis describes the PEX system. PEX is an execution service that facilitates processing remote data in partitionable low bandwidth networks. It is motivated by massive amount of *observational data* that are generated automatically from remote sensors every hour. Dozens of particle accelerators around the world individually collect terabytes of data each year from experiments [ROR95]. Satellites send terabytes worth of observation to earth each day [NAS]. Much of this data is stored near where it was generated or received. The state-of-the-art in networking is not such that the data can be made available anywhere, and in any case copyright laws may prevent shipping the data. Due to the large amounts of data, replication is not usually possible, and caches cannot be made large enough to provide significant hit ratios. PEX has been designed to address these issues.

Computation on these observational data involves finding some pattern by scanning the entire data set. This type of computation can be easily parallelized, resulting in significant speed-up. The individual computations require little or no interaction. A cluster of computers on a conventional network is inexpensive and well-suited for this type of processing.

The typical `rsh`-style of remote computation, common in Unix processing, does not provide convenient access to the remote resources. The problem is that some computations require hundreds of processes. Creation of the processes is relatively simple, but coordinating hundreds is difficult.

There is a need for an execution service that organizes related processes into sessions. A user should be able to add processes to a session, list running (or recently terminated) processes in a session, and with a single command be able to terminate a session. In addition, it might be useful for a session to be shared among several users. These users would be cooperating on the same scientific computation, but would not necessarily be co-located.

Sessions must be able to tolerate failures of processors and network partitions. A single session may exist for days or weeks. If a process crash eradicates the session, or if a network failure renders the session inaccessible, hundreds of processes may run unmanaged and would most likely be inaccessible, consuming resources but not performing useful work.

An execution service provides more than just simple process placement or even sophisticated load balancing. It should:

- support process placement based on application-defined placement functions (such as data location);
- support long-running, parallel computations;
- be tolerant of processor failures and network partitions;
- support heterogeneity of CPU types and operating systems; and
- support cooperation between multiple, remote users.

PEX is such a service. It allows computations to be set up near data, so data need not be copied over the network. It provides a session interface that facilitates users in managing related computations over scattered data. Internally PEX replicates sessions, so that it can ensure that computations keep on running and that clients be able to continue accessing them in face of partitions.

1.2 Related Work: Execution Services

This work is related to two areas of research: partitionable operations and execution services. This section reviews prior work on execution services. Section 2.2 and Section 2.3 review partitionable operations.

Many execution services have been proposed in the past. However, most of them assumed a homogeneous pool of processors. None of them allowed placement of processes based on data availability. None of them provided fault-tolerance for the services themselves. [1]

The first wave of interest in sharing of workstations connected by network occurred in the early 1980s when the cost of workstations became cheap enough that it was viable to allocate a workstation to each user. Since no user needed his or her machine all the time, this created an opportunity to utilize otherwise wasted CPU cycles. Early research focused on identifying basic distributed operating system features that would be useful in sharing. The efforts include Demos/MP [PM83], V [TLC85], and Butler [Nic87]. Later projects focused more on robustness and scalability of systems themselves, and include DAWGS [CM92], Condor [LLM88, ELvD+96], Batrun [TKDA96], and DBC [CSL96].

All these systems had the primary goal of transparency—a user who submits a job to the system should not notice the difference whether the job runs on the local machine or a remote machine, and the workstation's owner should not be affected by jobs owned by other users if he or she wishes to use that machine.

[1] DAWGS[CM92] replicated its processor database but did not guarantee consistency of this database.

Features identified as useful included the detection of an owner's activity [Nic87], preemption and migration [PM83, Che88], and forwarding of messages after process migration [PM83].

These early systems assumed all workstations were equally capable of running any job. None of the systems provided support for coordinating processes working on the same task—each task is managed separately. The V system selects a machine through a random bidding protocol, while the Butler system selects machines by consulting a central database.[2] The central database aids in machine selection, but it creates a central point of failure.

DAWGS is similar to Butler in that the service maintains a database. However, to make the service highly available, DAWGS allows each machine to maintain its own database. Whenever the status of machine changes between idle and busy, the new state is broadcast. Each machine also independently determines whether other machines are functioning by sending/receiving periodic "heartbeat" messages.

DAWGS does not maintain consistency of the individual machine databases. This makes the automatic restart of failed processes (which DAWGS does not currently provide) unsafe, because disagreement about which machines are alive could cause multiple copies of processes to be restarted. If this repeats multiple times, the system can get clogged up. Another problem with DAWGS or with this architecture is that full replication does not scale.

The Condor and Batrun systems do support scalability, migration, and the de-

[2]The update to the database may be delayed, however, and does not eliminate the need for negotiation with the remote machine.

tection of an owner's activity. For scalability, both systems have a two-level hierarchical structure. Condor assumes that different clusters are managed by different administrative domains, and that accesses to resources in different clusters must be submitted separately to the manager of each cluster.

Both Condor and Batrun have a similar structure for the base level cluster. A centralized scheduler does all allocations within a cluster. The scheduler also maintains a central database about machines in that cluster. However, this creates a single point of failure within a cluster. If the scheduler become unavailable, all the workstations within that cluster become inaccessible. Like earlier systems, Condor and Batrun assume a homogeneous environment (Condor does allow different environments across different clusters).

The DBC system, which is written over Condor, does support asymmetry of data access. DBC uses Condor to select a machine for a job. If the machine does not have the necessary data, DBC copies the data to the machine before starting the job. This works well if the amount of data is small, but not for large data sets, or when data is protected by licenses or copyright.

1.3 Outline

The rest of this dissertation is organized as follows. Chapter 2 presents a classification of partitionable systems. Chapter 3 describes the execution service PEX. Chapter 4 presents performance measurements for PEX. Chapter 5 concludes the dissertation.

Chapter 2

Partitionable Systems

2.1 The System Model

2.1.1 Partitions in Asynchronous Distributed Systems

An asynchronous distributed system consists of a set of *processes* that communicate by sending and receiving *messages*. A message may be concurrently sent to more than one process. A message is considered to be *delivered* at a destination process when the destination process initiates actions based on the message.

A process may fail by *crashing, omitting to send a message, or omitting to receive a message*. Failures may leave a message undelivered at some destinations. A system is *physically partitioned* when no message can be sent from a process and delivered to another process for a significant period of time.

In general, connectivity in a distributed system is neither bidirectional nor transitive. It is possible for a message to be deliverable from process A to process B, but not from process B to process A. It is possible a message to be communicated between processes A and B, and between processes A and C, but not between processes B and C. Thus, having all processes agree on which others they can communicate with can be complicated. Yet having such an agreement is useful. We will come back to this issue in Section 2.3.

2.1.2 Partitionable Services

A *service* continually performs actions based on *request* messages and produces *response* messages. In an asynchronous distributed system, a service typically consists of a set of processes called *servers*, and requesters are processes called *clients*.

There are a number of reasons why the designer of a service may choose to use multiple servers rather than a single server.

Throughput: If different servers are capable of handling the same requests, increasing the number of servers proportionally increases the number of requests the service can handle per unit time, subject to contentions at shared resources.

Availability: If different servers are capable of handling the same sets of requests, then the service can continue operating even when individual servers fail, as long as at least one server is up and running.

Quality of Service: A service may render different quality of service based on what clients are willing to pay. Different servers may perform the same service for a different price, so that the service may be performed faster if clients are willing to pay more.

Specialization: A complicated service may be better organized and understood as a whole if servers specialize in different types of tasks.

Clients may or may not know that a service consists of multiple servers. Some services transparently distribute requests from clients to servers. Other services require that clients pick the sets of servers to which their requests are sent.

A service is *partitionable* if it can continue operating correctly despite partitions.

2.1.3 Replicated Objects

A service may maintain a set of *replicated objects*. Client requests may involve a creation, an update, or a deletion of these replicated objects. A replicated object consists of a set of *replicas* maintained by different servers comprising the service, although not every server necessarily has a replica of every replicated object. Every object is referred to by a service-wide unique name. An access to a replicated object returns one or more replicas of the object.

The main reasons for replicating objects are fault-tolerance and availability. A failure of a server may make a replica inaccessible. A replicated object will survive such a failure as long as not all servers fail at the same time.

Availability of replicated objects does not come without problems, however. If a replica of an object is temporarily inaccessible due to a partition and if a client request involves an update to the object, replicas may become inconsistent.

2.1.4 Consistency Maintenance Protocols

A goal of a partitionable service is to maintain *consistency* among replicas of the same replicated object despite network partitions. There are two ways to deal with this: pessimistic and optimistic.

Pessimistic protocols prevent operations that might cause inconsistencies from being carried out in the first place. These include token-based protocols, where only the token holder can access replicas [MW82], class conflict analysis, where only important operations can access replicas during partition [SW84, Wri83], and weighted voting [Gif79].

The weighted voting protocol forces a certain number of replicas be accessed for read and write operations (read and write quorums) so that conflicting operations can be detected and aborted. It is based on the following intuition. If two times the write quorum (the number of replicas to be read when writing an object) exceeds the total number of replicas, at least one replica will be in common between two write operations of an object and conflicts between two write operations can be prevented. If the sum of write and read quorums exceeds the total number of replicas, at least one replica will be in common between any write and read operations, and conflicts between read and write operations can be prevented.

Optimistic protocols allow operations that might cause inconsistencies to be carried out, but the effect of such updates do not become visible to users until some later time when the system ensures that updates can be applied to all replicas. For example, consider the case where a system was partitioned in such a way that only one of two replicas of an object was accessible from each partition. Suppose the replica in one of the partitions is modified. Then the system can prevent the inconsistency between the two replicas from being visible to users after the partition is healed by applying the update performed to one replica during partition to the other replica. Simply applying missing updates might not always ensure consistency, however. This is the case, for example, if replicas in different partitions were allowed to be updated differently and these updates do not commute.

There are many variations of pessimistic and optimistic protocols, and they can be quite complex. For representative protocols for database systems refer to a survey paper by Davidson, Garcia-Molina, and Skeen [DGMS85], and to a protocol by El Abbadi and Toueg [ET89].

Consistency-maintenance protocols face a complication when the service partitions while they are in progress. A properly designed protocol should be able to deal with this. We return to this issue in Section 2.3.

2.2 Partitionable Services

While many consistency-maintenance protocols for partitionable systems have been proposed, not many have been used in actual systems outside of the database area. In this section, we discuss consistency-maintenance protocols that have been used outside of database systems.

Most protocols developed for general systems have a much looser notion of consistency than those developed for database systems, and are optimistic. Optimistic protocols are good for systems if conflicts are rare, because they do not penalize most operations for expensive checking. This is a reasonable assumption to make, for example, for file systems where most files are owned by individual users.

2.2.1 Consistency Criteria

Protocols for database systems typically use one-copy serializability as their consistency criterion. Database systems group related operations into a transaction, consisting of a series of read and write operations. Database systems exhibit *serializability*, if the effect on the database of processing concurrent transactions is the same as some serial ordering of these transactions. In other words, this serializability isolates one transaction from another and makes each transaction look like it owns the database. A replicated database satisfies *one-copy serializability*, if the effect of running transactions on a replicated database is the same effect as some serial ordering of these transactions on non-replicated version of the database [BG83].

One-copy serializability is expensive to ensure. This section reviews alternative consistency criteria used outside of database systems.

One-Copy Serializability of Write Operations

One-copy serializability of write operations is the strongest consistency criterion for general systems. One-copy serializability of write operations concerns *write-write* conflicts but not *read-write* conflicts. Thus the criterion does not work well for transactions accessing multiple data items. However, this is not such a big loss for many general systems such as file systems where data items are more or less independent of one another, and there is a big advantage to being able to use this criterion. As described in Section 2.2.2, simpler algorithms can be developed for this than for one-copy serializability.

Mutual Consistency

Mutual consistency declares two replicas consistent if their contents are identical. This is an even simpler than one-copy serializability of write operations, yet still quite useful. For example, applying two commutative (that is, the order in which the updates are performed is not important) operations in different orders at different replicas generates a conflict in one-copy serializability of write operations, but not with respect to the mutual consistency criteria. As described in Section 2.2.2 many conflict resolution algorithms can be designed using this criteria.

Best Effort

Some applications do not require rigorous consistency. All they require might be enough context to recover information they need in case of inconsistencies. For example, in electronic mail systems, users typically do not care whether they receive mail addressed to a common set of people in the same order as others do or not. What matters more is that they eventually get all mail addressed to them [SBN82].

2.2.2 Conflict Detection Algorithms

Of the consistency criteria discussed above for general systems, the best effort criterion does not require conflict detection. Mutual consistency can be checked by direct comparison of replicas. In fact, some algorithm that exhibit consistency, such as state reconstruction from scratch (See Section 2.2.3), do not even require conflict detection. So the only remaining case is one-copy serializability of write operations, and here there is a good detection algorithm based on *version vector*.

Version Vectors

The version vector algorithm [JPR+83] ensures one-copy serializability of write operations. It was designed for LOCUS [WPE+83], which was the first general system to address the problem of availability and consistency in a partitionable environment. The algorithm has been adopted by many other systems including Coda [SKK+90], Bayou [PST+97], and Rover [JdT+95].

Version vectors compactly record the write history of a replica. It is easy to exchange version vectors over the network, and the conflict detection algorithm using version vectors is simple. The basic algorithm in [JPR+83] only allowed a fixed number of replicas. Ficus [Guy91], a follow-up project of LOCUS, extended the algorithm so that the number of replicas could change.

A version vector for an object is a mapping from replica to natural numbers. Each entry in a vector is a monotonically increasing number that records the number of updates initiated at that replica that are known to the owner of the version vector. One version vector dominates another if the entries of the first are entry by entry larger than those of the the second. A replica with a dominating vector knows all updates known by the other replica. Two divergent replicas can be reconciled by the submissive replica adopting the state and the version vector of the dominant replica. The reconciliation itself is an update operation, so the version vector entry for the replica that initiated the reconciliation should be incremented.

If neither vector dominates, then there is a write-write conflict. Each replica has made an update not known to the other replica. Reconciling conflicting replicas may or may not be possible. Most systems do automatic reconciliation for certain objects (e.g., directories and mail boxes), but leave it up to the user to reconcile others (e.g., text files).

In [JPR+83], an entry in a version vector was a pair of host and update number at the location, and not just the update number as described earlier. This allows the number of replicas to increase, but its implementation is relatively expensive. Comparing version vectors requires comparing corresponding elements, which is more expensive for sets than for integer indexed arrays. Perhaps, for this reason, the location information was never much utilized.

Ficus allows changing the number of replicas through a use of an *ignore mask*. Each bit in a ignore mask corresponds to a specific replica. Its bit location corresponds to its index in the version vector. When a replica is retired, its corresponding bit in the ignore mask is turned on. Retiring a replica is useful, for example, when the replica is too corrupted to bring it up to date. Once a replica is retired, it can never be resurrected. Like a version vector, each replica has its own ignore mask, which encodes its knowledge of the replica retirement history. In comparing version vectors, entries that have a corresponding ignore mask bit set are ignored. In reconciling, a new mask is formed by taking the union of two ignore masks.

The ignore mask solves an additional problem: the create/delete ambiguity (first noted by [FM82]). Consider an existence of a replica of an object at one site and no existence at another site that should have a replica. Without additional information, we cannot distinguish whether this is a creation of a replica at the first site, or deletion of previously existing replica at the second site. With the extended version vector, creation is represented with a ignore mask bits all off for the first site, and deletion is represented with a ignore mask bits with all set for the second site.

The Ficus extension to the version vector cannot be used in all cases. For example, consider the case where there are two replicas of an object. Asynchronous distributed systems cannot distinguish crash failures from transient failures. If two sites partition away, both may erroneously decide that the other is dead and retire the other replica. Upon merging the two sites, the union of ignore masks are formed, and all replicas would be lost. Thus the Ficus extension does not tolerate crash failures, and entries for dead replicas may have to remain forever available. Ficus gets around this problem by assuming there are no permanent crash failures.

2.2.3 Conflict Resolution Algorithms

Update logging is a conflict resolution algorithm that ensures one-copy serializability of write operations. All other algorithms presented in this section ensure the mutual consistency criteria. We do not claim that the techniques presented in this section cover all possible forms of recovery from partitions. For every application that is to work in a partitionable environment, the best way to merge state has to be carefully considered.

Update Logging

Update logging is simply the maintenance of a log of update operations done to replicas. Starting from the same consistent state, two replicas can arrive at another consistent state by replaying the same update log.

If updates are commutative, two update logs can be combined. This would allow two replicas that became divergent to become consistent by replaying missing operations from each other's log. For example, an integer object that only supports additions and subtractions would fall in this category, or a bank account that allows clients to build up a debt. When two partitions merge, they simply execute each other's logs to become consistent again.

When using a partitionable group communication system (see Section 2.3) members in the same partition shares the same log. When two partition merge they form a new partition with their logs combined. Therefore, logs can grow without bound. One way to cope with this problem is to take a snapshot of a replica state into a stable storage once in an while and truncate the log. A new replica state can then be created by starting with the saved state and replaying a new log started after the last truncation.

A variant of this approach is to use designated primary partitions. The primary partition does not maintain a log at all; it just applies updates to its state. When two non-primary partitions merge, they concatenate their logs. When a non-primary partition merges with the primary partition (to form a new primary partition), missing updates in the log are applied to the primary partition's state, and the result is transferred to the non-primary partition. The log is then discarded.[1]

Union

Often, the state of an application is either a list or is a set of small objects or entries. A good example is a directory system. The state may first be subdivided into individual directory objects, and then further divided into entries in a directory.

When two directories diverge, they can usually be merged fairly by taking their union. There are two problems that with this, though. First, if an entry was deleted from one of the directories, it will reappear. Second, the two directories may have entries for different objects under the same name.

The first problem is simply an inconvenience to the user, but preferable to a file disappearing (which may happen if state transfer were used). It can be a problem if the object is a so-called "lock file," because this appearance of a lock could hang up a system.

The second problem is harder to deal with. One solution would be to adopt the entry that is more recent, requiring all updates to be timestamped. Another would be to create two entries with slightly different names. For example, the two files, both called "x", could be named "x.a" and "x.b" respectively, and a mail message could be sent to the owner of the file explaining what was done. Typically, this solution will work fine, except if the file in question is a system file without which the system stops working (for example, the password file). A better solution in this

[1] It may be important that the application of the log to the state, and the discarding of the log, happen atomically, so that the log is not applied to the state twice.

case would be to leave one of the file versions under the same name.

In PEX (See Chapter 3), a similar solution is used for its session objects. A session describes a list of related processes. When merging partitions, PEX takes the union of versions of the same session. If there are conflicts, PEX can, in some cases, detect that one version is strictly newer than the other and adopt that one. Otherwise, PEX checks with the processors that run the processes in question, and tries to figure out the best solution. Although highly unlikely, it may not be the correct one. For example, PEX may not be able to tell from the session or from checking the processor whether a user has tried to kill a process. If PEX leaves a process that a user has tried to kill alive, the user can detect this when it lists the session and can attempt to kill the process again.

Diff Listings

In version control systems, it is possible to find the most recent version X of a file that two later versions, Y and Z, have in common. Then, it is possible to construct diff listings for Y and Z that can then be compared. Each diff listing describes how the corresponding version can be constructed from X. By merging the two diff listings, we can construct a version that contains both updates.

This may work in many cases. For example, if one user changed something in the beginning of a file, and some other user something in the end, then the file may still be internally consistent if both changes are applied. It may not work in other cases. For example, if the file is a program the first user removed a declaration of a variable that was previously unused, and the second user started using the variable, the problem of an undeclared variable would arise.

Systems supporting concurrency control try to prevent this. For example, a user may lock a file before trying to change it. This way two users cannot update the same file. Optimistic concurrency control tries to detect potential clashes by recording which parts of an object a user reads, and which part a user writes. When one user A reads a part that the other user B wrote subsequently, then A is not allowed, or just warned, when it tries to write something. This approach still may fail for the undeclared variable example above. The second user may not actually read the first part of the file, if he or she remembered that the declaration was there.

The effectiveness of these solutions depends on the granularity of the regions that are locked or recorded. In the program example above, locking the first part of the file would not solve the problem. Yet making the granularity too large may reduce performance by limiting the number of users that can work on a region concurrently and increases the probability that regions remain locked when the system partitions, thus reducing availability.

Examples of systems that use diff listings for conflict resolution are version control systems RCS [Tic85] and CVS [cvs].

State Reconstruction from Scratch

For some applications, it is possible to discard the state of the merging partitions altogether, and construct a new one from scratch. PEX, for example, maintains a replicated data base of available processors and their load. In the case of a partition, replicas of the data base may diverge because they may not be able to access the same set of processors. When merging, PEX simply takes the union of all known processors, and pings them all to see if they are available, and what their load is. In general, state reconstruction from scratch is suitable to replicated caches. When merging, the cache can simply be emptied.

2.3 Group Communication Systems

Most partitionable systems discussed so far regard partitions as if they were something that could be easily detected. However, detecting partitions is a difficult problem. Protocols to make such decisions are complicated, error prone, and subject to continual improvement.

One way to detect partitions is to rely on a partitionable group communication system [Bir96]. Group communication systems are designed to facilitate building distributed applications. They identify properties useful in building distributed applications and protocols to provide them, so that application designers can concentrate on the functionalities of applications themselves rather than dealing with peculiarities that arises because they have to run in a distributed environment. This section reviews representative group communication systems.

2.3.1 Basic Services

Although group communications systems differ in specifics of which suite of protocols they offer and how they offer them, the protocols tend to cover three fundamental areas: failure detection, group membership, and message ordering.

Failure Detection

A failure detection protocol decides which processors are suspected of being failed. In the simplest form, a failure detector pings each group member periodically and waits for a reply from each. Members that do not send a reply in some time period are suspected of having failed. A member suspected of failing may not necessarily be faulty. Such member might be temporarily congested, the network might temporarily be heavily loaded, or a message simply might have gotten lost. To avoid making a mistake, more advanced failure detectors wait to raise a suspicion until a member has failed to reply to multiple consecutive pings.

Group Membership

A membership protocol decides on a logical group—a set of members that need to and think that they are able to communicate with one another. To join a group, a member sends a “join request” to an appropriate place, be it a separate membership service, to a coordinator of a group, or all group members. The membership protocol uses a failure detection protocol to monitor existing group members. Periodically, the membership protocol gathers information about which new members should join a group and which old members should be dropped. It then sends out the new group information to all members of the new group.

A *view* of a group consists a set of members that some member observes.

Message Ordering

Having an agreement about order of messages are useful in coordinating distributed parts of an application. Group communication provides four fundamental types of message ordering: unordered, FIFO, total, and causal ordering. Total order guarantees that messages are delivered to every member in the same order. Causal order guarantees that messages are delivered to every member in the way that preserves causal relationship [Lam78].

Virtual synchrony guarantees that if two members are in the same consecutive views they see the same set of messages in the required order.

2.3.2 Partitions

Primary Partition Systems

The issue of partition has been receiving increasing attention in the area of group communication. Isis [BJ87] supported group communication only within the majority partition. Isis killed members in minority partition so that no minority partitions kept on running. Experience with Isis subsequently revealed that this approach is unsuitable for applications that can continue operating without and do not require access the primary partition.

Partitionable Systems

Next generation group communication systems, including Horus [vRBM96], Transis [ADKM92], Phoenix [MFSW95] Relacs [BDMS97], and NavTech [VRC$^+$95] allowed minority partitions to continue exist and communicate. In a partitionable group communication system, a group may be divided into multiple partitions. A partitionable group membership service provides consistent information to the members about the partition that they are in, as well as providing ordered communication within each partition.

As a result of membership change, partitions may merge back. When this happens states of previous partitions must be merged back. Most systems use a fixed policy of taking the majority partition view when merging partitions. This raises the problem that states created in minority partitions are not preserved.

Message Order Redefinition

Totem [MMSA+96, MAMSA94] proposed what they call *extended virtual synchrony*. It redefines the meaning of causal and total order, so that it makes sense across partitions.

Merge Tool for Applications

The latest trend in the group communication systems is the awareness that no single merging algorithm can satisfy all applications, and that, ultimately, merging must be done at the application level. Examples of systems that allow application to involve in merging include, besides PEX, merging-help tool NavCoop [CAV96], and applications designed for Relacs [BDMS97].

NavCoop is a merging-help tool build over the partitionable system NavTech. It defines a notion of sub-group and associates an importance with each sub-group, according to the identity and weight of its members. Application designers can specify split/merge functions based on the relative importance of sub-groups.

Partition-Aware Applications

To our knowledge PEX is the only application actually implemented that handles non-trivial merging of replicated objects. However, Relacs presents a paper design of a partition-aware application. Members of this application start with a set of tasks to be done. They also maintain a set of completed tasks. Each member continually picks a task from the set, generates a result, and multicasts it to other members. When the group partitions, members may not learn about tasks completed by members in other partitions. This leads to redundant work. In the worst case, it could take n times the resources to complete a set of tasks using n processors compared to using 1 processor with no speed up. (Consider the case where the system, consisting of processors of equal speed, partitions to individual processors briefly every time processors are about to start a new task from a set.) However, executing a task more than once does not lead to an incorrect results as long as the task does not have a side effect. When two partitions merge, members compute the completed task set by taking the superset of completed task sets from each partition.

Chapter 3

PEX: An Execution Service for A Partitionable Low Bandwidth Network

This chapter describes an execution service for a partitionable low-bandwidth network, called PEX. The contributions of this chapter are:

- the design of a remote-execution service that provides grouping of multiple related processes, process placement based on various criteria such as data location, support for heterogeneous environments, and fault-tolerance.
- support for process migration in heterogeneous environments.

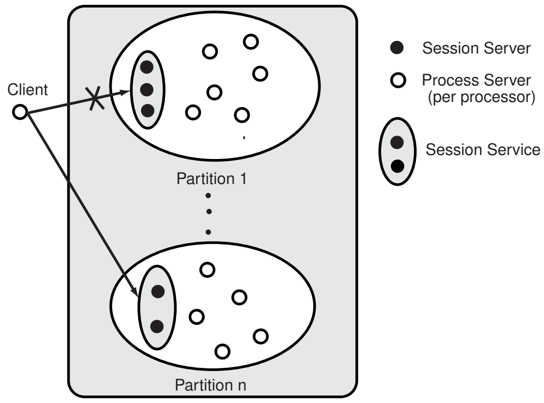

Figure 3.1: PEX architecture for a single institute.

3.1 The PEX Architecture

A PEX system consists of *institutes*. Resources managed by a single institute can be accessed with a single request and employ a group communications system. It is convenient to think of an institute as corresponding to an administrative domain (e.g., the Cornell computer science department). How many processors an institute can effectively manage ultimately depends on the performance of the underling group communication system.

If the client needs to access more than one institute, it has to break up its tasks into multiple sessions, one for each institute. Currently, we assume that the client knows which institute stores the data set that the computation needs; eventually, we hope to use or develop a fault-tolerant directory service for this purpose.

Figure 3.1 shows the PEX architecture for a single institute. An institute manages a set of processors. Each processor runs a *process server*. A process server handles execution and termination of processes on that processor. A small number of processors (typically, two or three) in an institute each run a *session server*.

PEX uses two groups: the *session service group* and the *execution service group*. The session service group consists of session servers. It serves as the access point for clients that want to use processors managed by the institute and maintains information about processors in the institute, on-going and recently terminated sessions, and requests issued by clients. This information is replicated, so that failure of processor running a session server does not result in loss of that information.

The execution service group consists of all sessions servers and process servers. Its purpose is to keep track of which processors are up and running in the institute and to forward client requests and responses between the session service group and processors involved.

An alternative design of the execution service group might be to have only one type of server that combines the functionality of session server and that of process server running on each processor. In this design, clients send requests to any of the servers. This approach may appear simpler and advantageous; if the server that received a request were able to process the request locally on the processor it runs, it could save the cost of a message send. However, more careful consideration reveals that this approach has disadvantages when the service must support more than simple random placement for a single command. Informed placement of commands in the institute requires an accurate view of the institute, including which commands are being served and on which processor. Duplicate execution of commands is expensive when commands are long-running. The problem would be compounded if these processes were to be restarted automatically by the service. Obtaining an accurate view of the institute requires communication with other servers, erasing the savings in message sends. Also, a target application of our service involves many processes executed at specific processors (ones with data). In this case, placement of processes on a remote processor is unavoidable.

Most importantly, however, the fatal disadvantage of the simple design is the cost of maintaining consistency of replicated information, which increases with the degree of replication. Among other problems, a higher degree of replication leads to greater message loss. Thus there is an incentive to keeping the degree of replication small, as in the two-service approach.

3.2 Replicated Objects

The session service replicates information so that the failure of a session server itself does not result in a loss for long-running computations. The replicated information is the processor database, on-going and recently completed sessions, and on-going and recently-completed client requests.

3.2.1 Processor Database

The *processor database* maintains information about the processors in the local processor group. For each processor, it maintains attributes like CPU type and speed, operating system, amount of memory, local data sets, group membership, load, and owner. Based on this information and information provided along with each process creation request (such as which data set it needs to access), the session service can do informed process placement.

3.2.2 Session Objects

A *session* provides a means for clients to keep track of related processes. Clients can monitor or manipulate processes individually or as a group according the session they belong to. Sessions are identified by an identifier unique across all partitions. This identifier is constructed from an RPC identifier attached to the session creation request, which includes the address of the client and the time that the request was issued. Sessions can be accessed by multiple clients. PEX arbitrates requests to a shared session, so that multiple clients can safely carry on tasks on the same session concurrently.

The session service maintains the following information for each process.

session identifier: the identifier for the session the process belongs to which.

process identifier: the identifier for the process.

process server: the address of the process server that runs at the processor where the process is running (or has most recently run).

process policies: the policies of the process. It specifies policies like whether the process should be automatically restarted on failure and whether it is migratable.

processor specification: the specification for processors. This is used to restart or to migrate the process, and is among the arguments to the client's process creation request.

process properties: the properties of process. The execution service allows a processor to restrict the kinds of processes it runs. A processor, for example, might only run processes related to a specific project or might give priority to the owner of the processor. A process can run on processors only if it can satisfy that processors's properties.

command specification: the command to execute. This may be qualified based on properties that processor has.

command flag, migration port: these entries are used for migrating the process.

process status: the status of the process: not started yet, running, transitioning (being killed, restarted, or migrated), done (completed, killed, or failed).

migration history: the history of where this process has moved, encoded as a list of addresses of the process servers of processors where the process ran. This information is used to filter out messages from old locations after the process has moved. It can also used when selecting a processor, if the process is migratable (e.g., to avoid the ping-pong effect.)

process start time: the time at which the process has started.

process end time: the time the process has completed its execution. This is used to garbage-collect the process object.

3.2.3 Client-Request Objects

The *client-request objects* describe on-going and recently completed client requests. They are replicated, both to avoid the loss of a request when a session server dies and, more importantly, to filter duplicate requests. The session service maintains the following information for each request.

session server: the address of the session server that has received the client request. This and the next item are used when sending a reply back to the client. If a client resends the request, it is assumed that the client considers the old server dead; the address of the new server replaces the old one.

client-request identifier: the RPC identifier of the client request.

client message identifier: the message identifier of the client request. This is location-independent information extracted from the request identifier, and it is used to filter retransmissions of the request.

client request: a copy of the client's request.

request stage: the stage in request processing: pre-processed (before any sub-request were forwarded), processed (at least one sub-request has been submitted to the underlying communication system for forwarding), done (request has been successfully completed and a reply has been sent to the client), or failed.

sub-requests: sub-requests generated from the client's requests and forwarded to process servers. These are saved in case they need to be resent. Resending sub-requests may happen when the client retransmits the request. Rather than resending all sub-requests, the session service only resends those that have not completed.

base index: sub-requests are identified by the index number unique within the request. The base index refers to the index of the lowest indexed sub-request. This number is set to a higher number every time the request is re-processed.

index ticket: this is the index number that should be used for the next newly generated sub-request. It is incremented for every new sub-request.

partial results: this is a place holder for results received from sub-requests, such as information about a newly created process.

The session service keeps the following information about each sub-request:

index: the index of the sub-request.

destination: the address of the process server where the sub-request was forwarded.

sub-request: the actual sub-request.

stage: the stage of the sub-request: pre-processed, processed (the sub-request has been submitted to the underlying communication system for forwarding), done (a reply has been received from the process server), or failed.

Table 3.1: The client interface.

Request	Arguments	Response
SessionCreate		session id
ProcessCreate	session id, processor specs, process properties, command list, failure recovery instruction	process id list, failed commands
SessionWait	session id	status
ProcessWait	session id, process id	status, small output
SessionKill	session id	status
ProcessKill	session id, process id	status
SessionGetState	session id	session state
ProcessGetState	session id, process id	process state
SessionDestroy	session id	status
DBGet	key	processor database entry
DBDelete	key	status
DBChange	key attribute-value list	status
Shutdown		status
UpStatus		information on processors

3.3 Client Interface

Clients communicate with PEX by sending RPC requests. PEX handles each request exactly once and sends back a reply. This section describes the client interface. There are two categories of requests: session management and administration. Typical PEX applications probably only need to access the session management interface. The administration interface is provided for system administrative applications to manage PEX. All types of requests are listed in Table 3.1.

3.3.1 Session Management

This section describes the RPC interface that clients use to manipulate sessions.

Before requesting that the session service start processes, a client must create a session using `SessionCreate`. The session service returns a unique session identifier. Once a session is created, clients can add processes to it using `ProcessCreate`. Arguments to a `ProcessCreate` request are a session identifier, processor specifications, process properties, a set of commands (one for each architecture), plus a *failure recovery instruction*. The reply to `ProcessCreate` includes process identifiers for each of the started processes and a list of commands that failed to start. PEX ensures that, at the time of creation, each process identifier refers to a unique object across all partitions. This is accomplished by adopting the already unique RPC identifier attached to the requests (see Section 3.4).

The processor specifications argument is a disjunctive list of conjunctive criteria used to select processors (e.g., [OS type = SunOS & load $<$ 1.0] $\vee$ [OS type = WinNT & data sets include EOS]). The session service selects a set of processors by testing each conjunctive criteria against the processor database. The process properties argument specifies characteristics of processes to be started. Some processors may restrict types of processes that they run (e.g., only those belonging to a particular user).

Each entry in the command set is a list of processor characteristics followed by a normal shell-style command. This allows execution of different versions of the same command based on processor architecture or operating system. The failure recovery instruction argument specifies the action that the session service should take in certain situations (such as whether to restart a process after a processor failure and recovery).

Clients can wait for processes using `SessionWait` or `ProcessWait`. `SessionWait` returns a reply when the last process in a session completes. `ProcessWait` waits for a particular process to complete. Using `SessionKill` or `ProcessKill`, processes can be terminated explicitly. The state of on-going sessions and processes can be examined using `SessionGetState` and `ProcessGetState`. Clients should destroy a session once it is no longer necessary using `SessionDestroy`.

3.3.2 Administration

The second category of interface that PEX provides is administrative; this interface should be restricted to system administrative applications. The interface provides messages to access the processor database and messages to communicate directly with servers.

PEX keeps various information about each processor in its processor database. Some of this information (e.g., the address of the process server and whether it is running) is obtained and updated when a process server is started on the processor. However not all information about a processor can be obtained systematically, by executing a command on the processor. Different types of processors provide different information; and even if different types of processors happen to provide the same information, the way this information is obtained may differ. Furthermore, some information kept at the processor database may not be known to the processor (e.g., the processor owner). Such special information about the processor can be added to the database using processor database interface request `DBChange`.

DBChange can be used both to change an entry or to create a new entry. It takes two arguments: key and attribute-value pairs. When it receives a `DBChange` request, the session service first looks for an entry in the processor database with the matching key. If an entry is found, then it is modified. Otherwise, a new entry is added for the given key with the given attribute-value pairs.

PEX also provides `DBGet` for accessing an entry from the processor database and `DBDelete` for deleting an entry from the processor database.

The other type of administrative messages that PEX provides are direct communication with servers. `UpStatus` inquires where servers are running and how many processes are running on each of the processors in the institute. `Shutdown` shuts down the entire execution service.

3.4 Handling Client Requests

Carrying out a client request exactly-once is desirable, not only because it avoids wasting resources, but because it simplifies maintaining consistency and coordinating multiple clients sharing the same session. However, achieving exactly-once execution is complicated when replicated objects are involved, since retransmissions of RPC requests may be dispatched to different servers. Servers must coordinate in order to filter out duplicated requests.

This section describes client-to-service and intra-service mechanisms that together ensure exactly-once execution of client requests ver view. The first subsection describes a reliable RPC mechanism that clients use to ensure at-least-once delivery of requests to the session service. The second subsection describes the mechanisms

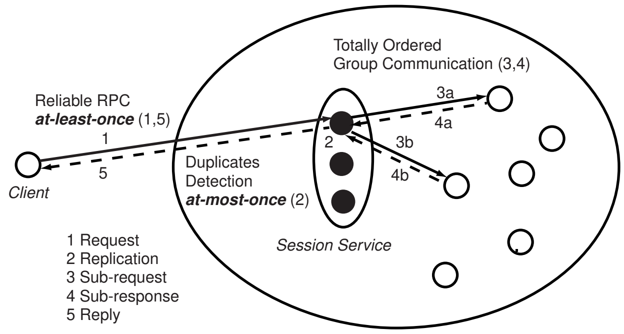

Figure 3.2: Exactly-once execution of client requests.

that the session service uses to ensure at-most-once execution of received client requests. Together they provide the desired behavior. (See Figure 3.2).

3.4.1 Reliable RPC with At-Least-Once Semantics

Clients access their sessions by sending requests to session servers and receiving responses. RPC requests and responses are sent over TCP/IP connections. Before the first request, a client must select one of the session servers. For this, the client maintains a *confidence value* between 0 and 1 for each session server. When initiating an RPC to the service for the first time, the client picks the server that has the highest confidence value. A confidence value of 0 implies that the server is likely down, while 1 implies that the server is likely up. Initially, the value is set to 0.5, and 0.1 is added to it every minute until it reaches 1. When a connection breaks, the confidence value of the corresponding server is halved, and a new server is chosen.

Connections to session servers are cached until they break or until the connection table in the operating system fills. In case of such communication problems, the client keeps choosing new servers and retransmitting the RPC request until it receives a corresponding response to the RPC request. That is, the client never gives up, and so at-least-once semantics are achieved.

For each new RPC request, the client creates a new RPC identifier that consists of the client address and a sequence number that is incremented for each new RPC request. Retransmissions of the same RPC request, typically sent to different servers, carry the same RPC identifier.

Clients may obtain server addresses using the *White Pages service* (WP). This service maps ASCII service names (such as "Session Service") to a list of TCP/IP addresses for servers that implement the named service. Each server periodically sends a *Refresh* request to the WP service (currently, once a minute). The WP service lists those servers that have refreshed in this way during the last ten minutes. Each institute runs its own WP service. The RPC mechanism automatically contacts the WP service at the institute of choice (in this case, where the client wants to create the session). The WP service is replicated in way similar to the session service described in the next section.

3.4.2 Replication with At-Most-Once Semantics

This section describes mechanisms that the session service uses to detect duplicate requests and to ensure at-most-once execution of each client request. Combined with the at-least-once delivery provided by our RPC facility, exactly-once execution of each client request is achieved.

Session servers and process servers that serve the same processor group use group membership and totally ordered communication protocols. These protocols ensure that all servers agree on the delivery order of messages and on group membership changes relative to the message ordering. Agreement simplifies the task of consistency maintenance of replicated objects.

Since the membership of the server group should be relatively stable, and most server communications is within a small subset of processors (either among session servers or between a process server and session servers), the cost of using group communication is minimal.

A client request is delivered initially to one of the servers. We call this server the *request manager*. Requests that can modify server state must be handled by all servers in the same order and at most once. One way of achieving this is *active replication*: the request manager multicasts the request to the session service group, and each server executes the request [Sch93]. If a request arrives at two different servers at approximately the same time, then they are both multicast, but the second to be delivered can be filtered automatically. In general, this approach to replication is costly; because each server has to execute the request and, if request processing is non-deterministic, it requires further communications among servers to keep the state of replicas consistent. In PEX, selection of processors can be non-deterministic.

An alternative approach is *primary-backup* where one of the servers, called the *primary*, carries out the request, then multicasts a *state update* message to the other servers [BMST93]. The state update message describes how the backups should modify their state to replicate the primary. Request managers forward their requests to the same primary; otherwise a request might still be executed more than once. The main disadvantage of this approach is that a failure of the primary means that a long fail-over delay is incurred (due to the length of failure detection and the subsequent agreement protocol among the backups).

PEX uses a hybrid of active and primary-backup replications. Upon receiving a client request, the request manager carries out the non-deterministic part of computation but without actually changing state. The request manager then forwards the request, along with the result of the non-deterministic computation, to all servers (including itself). Since the session service group uses totally ordered communication, all servers receive these forwarded requests in the same order and will therefore stay synchronized with each other. Each session server then carries out the remaining deterministic part of the computation, if needed, as described in the following.

When a session server receives a forwarded request, the server checks whether it has a client-request object with the same RPC identifier. If it does not, then the request is new and the server creates a client request object for it. Next, the session server carries out the deterministic portion of processing, which may include generating, but not sending, *sub-requests* to one or more process servers. After the processing completes, only the request manager forwards sub-requests, if any, to the selected process servers. Process servers that receive a sub-request do processing and reply to all session servers when that processing is done. When all sub-requests have completed, each session server deterministically generates a reply and stores it in the server's client-request object. The request manager also sends the reply to the client. This way the correct state of the process is known by the session service, even if the request manager fails.

Figure 3.3 shows pseudo-code for handing incoming requests. If a session server receives a client request for which it already has a client-request object, then it checks to see if the current request manager has failed (that is, removed from the process group). If it has, then the recipient of the retransmitted request becomes the new request manager. The new request manager checks the current state of the request. If the processing is completed, then it sends back the reply to the client. If it is still being processed, then it resends any sub-requests for which no replies were received, since it is possible that the sub-requests were lost. (To avoid duplicate processing of sub-requests, these messages contain a unique identifier as well.) After that, the protocol can be completed as normally.

3.5 Restoring Consistency after Partitions

This section describes how the session service deals with server failures and network partitions. Instead of replicating entire processes themselves, various objects are replicated. Much of this section is devoted to discussing how consistency of these objects is ensured.

Crashes and partitions are announced by the group membership service in the form of *view changes*. In a partitionable system, a processor that joins a group might not be a newly started member with no state, but an old member that has

```
let handle_request origin newreq =
   let redo_request request =
      let resend subreq =
         match subreq.substage with
          | PreProcess () ->
               send_request subreq
          | _ ->
               ()
      in
         List.iter resend request.subreqs
   in

   try
      let request = request_lookup newreq.id in
         if (is_dead request.mgr) then begin
            request.mgr <- origin ;
            match request.stage with ->
             | PreProcess () ->
                  reset_request request ;
                  do_request request
             | Process () ->
                  if origin = myendpt then
                     redo_request request
             | Done reply ->
                  if origin = myendpt then
                     send_client_reply request reply
             | Fail error ->
                  if origin = myendpt then
                     send_client_error request error
         end
   with NotFound ->
      do_request newreq
```

Figure 3.3: Handling incoming requests.

partitioned away sometime ago and continued processing. In this case, it is possible for replicas of a single object to have divergent states at view changes. The simple approach of adopting the primary partition view is unsatisfying when dealing with long running computations; by throwing away the state of a minority partition it is possible to lose the result calculated in a computation that has run a long time, simply because the primary partition did not know about it.

In general, constructing a consistent state out of divergent versions is application dependent. We designed PEX with a small set of replicated objects having states for which construction of consistent states is relatively easy: the process servers do not maintain replicated objects at all, and session servers maintain only three types of replicated objects: the processor database, sessions (which contain process objects), and client-request objects. We present merge algorithms for each of these.

3.5.1 Reconstruction of the Processor Database

PEX keeps the processor database consistent by reconstructing it from scratch when there is a view change, rather than by trying to reconcile divergent versions of the database. Reconstruction is possible because the session service only uses the entries of accessible processors. The entries of inaccessible processors may differ at the different servers, but this does not matter.

The state of an accessible processor is known by the process server that is running on the processor. After a view change, each process server multicasts its state to all session servers so that they can independently recreate the processor database. Ultimately, this scheme may not scale, since this means that potentially hundreds of processors broadcast a message on the network at the same time. Although we have not found these broadcasts to be a problem, we may adopt in future a more sophisticated algorithm similar to those used in the other replicated objects described below.

3.5.2 Merging Divergent Session Object Replicas

A session is described by a session object, which consists of a set of process objects. Merging two versions of a session object involves making sure that every replica of a resultant session object contains the same set of process objects, and that each process object refers to a single process[1]. PEX accomplishes this in two phases. In the first phase, the following subsets of the process objects are built:

live_set: processes that are up and running.

dead_set: processes that have terminated.

kill_set: duplicate processes that need to be terminated.

restart_set: failed processes that need to be restarted.

Note that processes in the dead set are terminated only in this partition. They may still be running and be accessible in other partitions.

In the second phase, the processes in the kill set are terminated and the processes in the restart set are restarted. (For efficiency, PEX actually reconciles the client-

[1] More precisely, merging ensures that each process object refers to a single process that a process server that is a member of the partition can access, since only such processes are accessible in the session service.

request objects in between the two phases, since this merge may make additional changes to the sets.) We will now discuss these phases in more detail.

Group membership services allow the election of one member as *coordinator* of a partition. When partitions merge, the coordinators of the (old) partitions broadcast the state of the sessions in their respective partitions. Each session server merges these states pairwise. This ensures that the set of process objects in each session object is the same across all replicas provided that the merge operation used is deterministic and commutative.

The algorithm for merging two versions of the same process object is shown in Figure 3.4. If the two versions refer to the same process, only one is added to the set; if only one of the processes is accessible, the live is used; if both are dead, it is placed into the kill set or the restart set depending on the client's failure recovery instruction; otherwise (both are live processes) the rank of process server managing conflicting processes is used to pick a process to keep alive deterministically.

Merging two session objects takes the union of all process objects known by either session, checking uniqueness of each process object. The union might include old processes that clients tried to remove, but this approach requires less space than maintaining the histories of session operations in each partition, which would be required to do a better job. Furthermore, clients can always get the state of sessions after merging and kill those processes that are not needed.

The same process identifier can refer to different processes. This happens when a client is partitioned from the original process and starts a new one. After the session objects merge, and only if both process servers are available, the one with

```
let svr1 = proc1.svr
and svr2 = proc2.svr
in
   if (svr1 = svr2) & (is_alive svr1) then
      add live_set proc1
   else if (is_alive svr1) & (is_dead svr2) then
      add live_set proc1
   else if (is_dead svr1) & (is_alive svr2) then
      add live_set proc2
   else if (is_dead svr1) & (is_dead svr2) then begin
      if to_restart proc1 then
         add restart_set proc1
      else
         add dead_set proc1
   end
   else if (rank_of svr1)  < (rank_of svr2) then begin
      add live_set proc1 ;
      add kill_set proc2
   end
   else begin
      add live_set proc2 ;
      add kill_set proc1
   end
```

Figure 3.4: The algorithm for merging two versions of a process object.

the lowest rank is chosen and the other is terminated. If both process servers are unavailable, the process is restarted, possibly creating a third version of it.

After ensuring that each session replica contains the same process objects, the session service makes sure that there is exactly one physical process running for each process object with running state by killing processes in the kill set and restarting processes in the restart set.

The process server restarts a process by executing the same command used to start the process originally. It is up to the application to save its state, if necessary. Restarting a process is similar to creation of a new process in that it involves nondeterministically picking a processor. However, rather than replicating the processor selection (which takes extra rounds of messages), one session server is put in charge of selecting a process server and sending a request to it. The process server in turn sends its reply back to all session servers. This way, the fact that the restarted process exists will be recorded, even if the session server that sent the request to the process server failed.

3.5.3 Merging Divergent Client-Request Object Replicas

As in merging session objects, merging client-request objects proceeds in two phases. A client-request object contains sub-request objects, each representing a message that has been sent to a different process server. Since request processing involves a non-deterministic selection of processors, processing the same client request again may not necessarily result in the same set of sub-requests. Rather than dealing with individual sub-requests, PEX simply adopts one of the client-request object versions at random in its entirety.

This approach might overlook some processes for requests that call for operations on all processes in a session (e.g., `SessionWait`), if processes not known by the partition where requests were issued were later merged in. This is not a problem, since the same scenario can happen in a failure-free case, when a client adds a process while an old session-wide request is still being processed.

The first phase of merging client-request objects is similar to that of session objects. It consists of the coordinator of each partition broadcasting the state of each replicated object on a view change, and having each session server independently merge received states pairwise.

Similar to process objects, the up/down status of the request manager is used to decide which version of the client-request object to keep, breaking ties using the ranks of the request managers.

As a result of processes terminating and restarting, some of the requests being served may not make sense (e.g., waiting for a dead process) or may become automatically done (e.g., killing an already dead process). The second phase, the *clean-up phase*, of merging client-request objects makes sure that requests represented are either being served or, if they are done, a reply is sent back to the client. This phase is best understood by considering the stages in the existence of a client-request object. These are:

NonExistent: A session server received a request from a client. Before forwarding the request to other session servers to create a replicated client-request object, the session server that received the request does any non-deterministic processing, such as selecting random processors.

PreProcess: Session servers received a forwarded client request and created the replicated object representing the request. Since requests are forwarded using totally-ordered group communication, each session server can deterministically filter out duplicate requests. The session server that originally received the requests becomes the request manager for that request.

Process: Session servers are done carrying out the deterministic part of request processing. Sub-requests to forward to process servers have been generated. During this stage, each sub-request goes through stages of its own:

SubProcess: Session servers have not received a reply for the sub-request.

SubDone: Session servers have received a reply for the sub-request.

SubFail(reason): The sub-request has failed.

Done(reply): Session servers have received a reply for every sub-request. The request manager has sent a reply back to the client.

Fail(reason): A sub-request has failed. The request manager has sent a error back to the client.

If the session server that received a client request fails while the request is in the `NonExistent` or the `PreProcess` stage, then no harm is done. The reliable RPC mechanism will soon retransmit the request, which will trigger continuation of the request processing. The death of any other session servers is harmless during these stages also, since other session servers are backups for replicated objects.

A view change that happens in the `Done` or the `Fail` stage requires no action. The client-request objects are kept at this stage only so that delayed retransmissions can be filtered out. A separate garbage collection mechanism will eventually clear old client-request objects.

Thus, the only difficult stage with regard to a view change is the `Process` stage, which may involve communicating with process servers. Not all requests go through the `Process` stage. Requests that do not involve forwarding requests to process servers (e.g., `SessionCreate`) jump directly from the `PreProcess` stage to the `Done` stage. There are only three types of requests that involve forwarding of requests to process servers:

- `ProcessCreate`
- `ProcessWait` and `SessionWait`[2]
- `ProcessKill` and `SessionKill`

The algorithms for `ProcessWait` and `SessionWait` are essentially the same, because `SessionWait` is broken down into multiple wait sub-requests for the individual processes. The same is true for `SessionKill` and `ProcessKill`.

Figure 3.5 shows the clean-up phase of merging algorithm for client-request objects. It divided into three cases—one for each type of requests. It uses the dead set and the restart set generated during merging of process objects. It produces a

[2]In practice, no physical sub-requests are sent for these. Sub-requests for a `Wait` request are implied by sending of the `ProcessCreate` request. Process servers send a sub-reply automatically to session service when the process is done.

set of requests called the *check set*. After going through the appropriate algorithm based on the type of request, the session servers go though each request in the check set to see if all sub-requests are complete (or failed). If they are, session servers mark the request as done; and, if it is the request manager, send a reply back to client.

ProcessCreate. A sub-request of a `ProcessCreate` request is useless if it is sent to a dead or unreachable process server. In our initial implementation, we have taken the approach of marking useless sub-requests as failed but continuing with the request rather than abandoning the whole request or redoing useless sub-requests. Abandoning the whole request could be useful, for example, when running a performance test where an exact number of lightly loaded processors are needed. Redoing the sub-request could be useful, if all components of an application must eventually run. We may add support for abandoning and redoing in a later implementation. For now, we mark useless sub-requests as failed and add the request to the check set for later checking of a request completion.

SessionWait and ProcessWait. If its destination process is to remain dead, a sub-request of a `Wait` request is useless. A useless sub-request of a wait request is marked failed and the request is added to the check set.

SessionKill and ProcessKill. If its destination process is to remain dead, a sub-request of a `Kill` request is essentially done. Such a sub-request is marked as done and the request is added to the check set.

```
let clean-up_request request =
   let f subreq =
      match subreq with
       | IProcessCreate _ ->
            if (is_dead subreq.dest) then begin
               subreq.stage <- (SubFail "procsvr failed") ;
               add check_set request
            end
       | IProcessWait (sessid, procid) ->
            let proc = lookup_process sessid procid in
               if (mem dead_set proc) then begin
                  subreq.stage <- (SubFail "procsvr failed") ;
                  add check_set request
               end
       | IProcessKill (sessid, procid) ->
            let proc = lookup_process sessid procid in
               if (mem dead_set proc) then begin
                  subreq.stage <- (SubFail "procsvr failed") ;
                  add check_set request
               end
               else if (mem restart_set proc) then begin
                  remove restart_set proc ;
                  subreq.stage <- SubDone ;
                  add check_set request
               end
   in
      List.iter f request.subreqs
```

Figure 3.5: The clean-up phase of merging client-request objects.

If the destination process of a `Kill` sub-request is to be restarted, it is more efficient not to restart the process in the first place. So such a process is removed from the restart set, the sub-request is marked as done, and the request is added to the check set.

3.6 Restart and Migration

3.6.1 Process Restart

PEX automatically restarts failed processes if the client instructs PEX to do so. Internally, the mechanism for restarting is much like that for creating processes. In creating a process, the client specifies the types of processors the process should on and the actual command used to create processes (which may depend on types of the processor selected). This information is saved. When a process fails, a session server repeats the processor selection according to the saved specification and sends the corresponding command to the process server of the selected processor.

PEX runs in a heterogeneous environment. Thus, unlike an execution service that runs in a homogeneous environment, PEX cannot expect applications to be completely transparent to restarting from the middle. If application processes keep some internal state, that state must be converted into some intermediate state that can be transferred between different processor architectures.

An earlier version of PEX also had a mechanism for application processes to checkpoint states. Application processes could periodically send their states to PEX. PEX restarted application processes by sending the saved state along with the command. We have since decided that it does not really matter whether this service is provided by PEX or some other entity. Regardless of how this service is provided, application processes must convert their state into an intermediate format and have it stored in a safe place. In fact, it is more flexible not to restrict what service saves state. Applications might have many states that can be saved with different collections of services. For example, a multimedia application may want to save audio and video in separate storage systems.

3.6.2 Process Migration

Once a mechanism to support restarting processes is in place, process migration can easily be supported. Earlier systems supporting migration, having been designed for a homogeneous environment, aimed to provide complete transparency to the application. If we remove this requirement and expect applications will save and retrieve processor independent states upon restarting, then migration becomes like restarting. The only difference being that in migration, the old process is still running when the new process is started.

PEX's approach to supporting migration is to inform the new process where the old one is and to have the new one engage in direct communication with the old one. To enable this, PEX requires that all migratable applications handle two special command arguments: `id` and `ancestor`. The `id` flag specifies the identifier given to the process by PEX and the ancestor flag specifies the address to be used by the starting process to communicate with the old one.

A migratable process is expected to eventually send PEX a port address. This message must be tagged by its identifier passed using the `id` argument. PEX saves the port address with the process object of the sender. When PEX decides to migrate the process, it starts a new one and passes the complete port address (processor:port) of the old one using the `ancestor` argument. When a process is started with an `ancestor` argument specified, it is to be understood that the process is a migrating process and it is replacing the specified ancestor process.

The application can be flexible about what to communicate between the old and the new processes. For example, the new process can tell the old one to exit right away or to complete on going transactions and send the current state before exiting.

3.7 Summary

This chapter described a partitionable execution service, called PEX. PEX provides easy access of remote resources based on on various criteria including data sets, allows users to organize related processes, supports an heterogeneous environment, and tolerates failures. PEX synthesizes existing and new techniques for maximum efficiency while not sacrificing fault-tolerance. It also supports process migration in the heterogeneous environment.

Chapter 4

Performance of PEX

4.1 Performance Testing Environment

4.1.1 Implementation of PEX

We implemented PEX in Ocaml [Ler98] over the Ensemble group communication system [Hay98]. Ensemble is a layered system, also written in Ocaml. Ensemble applications gain flexibility by stacking the minimum set of layers needed to produce the properties they need. Ensemble stacks can be optimized for normal cases using a separate static optimizer based on formal methods.

PEX is designed so that it can meaningfully coexist with an existing environment. Applications that do not need a sophisticated placement service can run on a processor managed by PEX without using PEX operation. PEX uses resource usage information from the underlying operating system. When it is not serving commands, PEX uses essentially no CPU.

Table 4.1: Configuration variables relevant to PEX.

variable name	description
ENS_ID	Ensemble user ID
ENS_MODES	Ensemble transport
ENS_PORT	Ensemble transport port
ENS_GROUPD_HOSTS	hosts where groupd run
ENS_GROUPD_PORT	port where groupd run
ENS_GOSSIP_HOSTS	hosts where gossip run
ENS_GOSSIP_PORT	port where gossip run
ENS_EXECSVR_HOSTS	hosts where sessvr run
ENS_EXECSVR_PORT	port where sessvr run
ENS_DBDEFAULT	(optional) processor database input file

PEX consists of two basic programs: `sessvr` for session servers and `procsvr` for process servers. To set up PEX as a service, at least one session server and one process server as well as an Ensemble group membership server `groupd` must be started using the same configuration. Table 4.1 lists the relevant configuration variables.

4.1.2 PEX Clients

To test the performance of PEX, we built two types of PEX clients: a distributed shell-like program called `dsh` and a parallel make program called `omake`.

Distributed Shell: `dsh`

`dsh` is a remote execution shell similar to Unix `rsh` but `dsh` allows programs (distributed or not) to be started on multiple machines. `dsh` reads in a list of commands to execute along with pre-defined composable machine selection criteria; it then is-

Table 4.2: Machine specification options available with `dsh`.

option	description
-name machine_name	specific processor
-arch machine_architecture	machine architecture
-os os_type	operating system type
-data data_set	has specific data set
-leastload n	n least loaded
-loadbound load	bounded load
-random n	n random processors

sues an appropriate `ProcessCreate` request to PEX.

The syntax of `dsh` is as follows:

```
dsh -c commands [machspec] [-p procspec]
```

Here, `machspec` are zero or more machine specification options, and `procspec` specifies process properties. Multiple commands to be executed in parallel are separated by commas. Table 4.2 lists the predefined machine selection criteria. Procspec is a list of attribute-value pairs separated by semicolons. e.g.,

```
-p "project=ensemble;owner=takako"
```

Here is an example use of `dsh`. The following starts two programs, producer and consumer, on two "least loaded of random" four processors (a criteria that works well even when processor load information is old [Mit97]).

```
dsh -c "producer,consumer" -random 4 -leastload 2
```

Parallel Make: `omake`

The parallel make program `omake` is another PEX application. Like Unix `make` [mak], `omake` takes a list of objects to make and a list of rules that describe how to make objects using a file *Makefile*; it builds any objects that are not up-to-date. Each rule associates with a set of target objects, objects that they depend on, and a list of commands. Target objects are out-of-date if any that they depend on are newer or if those objects do not exist. If objects are out-of-date, then the targets are created by executing associated commands. Each object may depend on other objects. Thus, checking whether an object is up-to-date is a recursive procedure.

Internally, `omake` repeats the following procedure until no more objects need to be made.

- Starting with objects to be made as specified by the user, recursively check dependencies for any objects that are out-of-date.
- Put together commands associated with objects to be made into a request to PEX.
- Wait for a reply from PEX.

PEX takes the list of commands given by `omake` and runs them in parallel. Objects made may depend on architecture. For this, `omake` lets users optionally specify types of processor and operating system where commands should run. In addition to sending this information to PEX, `omake` converts this information into two variable definitions `MACHTYPE` and `OSTYPE` for use in interpreting dependency rules in Makefile. Using these variables, users can share a single makefile for making targets for multiple different architectures.

This approach can be extended. Some objects made on processors of different architecture can be shared while others cannot. For example, Ocaml bytecode used in the Ensemble system is architecture independent. In building the Ensemble system, results of all but the final linking command can be shared across different architectures. Thus, rather than repeating `omake` for each different architecture, it is more efficient to execute `omake` once, which does all but the final commands in parallel. Then, in parallel, PEX executes the final command for each architecture type. This would require substantial modification to the syntax of Makefiles.

`omake` also takes an optional argument that specifies the directory where every command should run. When this directory is specified, `omake` prefixes every command with a change directory command before sending it to PEX.

4.1.3 Hardware

We ran our tests on Intel Pentium PCs running LINUX and on Sun SparcStations running either SunOS4 or Solaris. The processors were of various speeds and connected by 10 and 100 Mbit/sec Ethernets.

Table 4.3: Time to do basic operations. Measured in milliseconds over a collection of 60 processors.

Command	#Processes	Time (ms)			
		Total	`SessCreate`	`ProcCreate`	`SessDestroy`
`dsh`	1	208	10	51	3
`dsh`	2	293	9	56	2
`dsh`	4	326	9	66	2
`dsh`	8	462	9	126	2
`dsh`	16	506	9	119	2
`dsh`	32	817	8	201	2
`uptime`	1	140			
`sh`	1	210			
`rsh`	1	1700			

4.2 Performance Measurements

4.2.1 Basic Operations

PEX is primarily intended for running long sessions. However, in order to measure the overhead of creating sessions and processes, we used sessions that only ran one short command (namely, Unix `uptime`). We measured the time between having `dsh` send a request and receiving a response for the `SessionCreate`, `ProcessCreate`, and `SessionDestroy` operations. We ran our tests with session servers on three Pentium processors and process servers on SparcStations. Processor selection was random.

Table 4.3 shows the results for varying numbers of processes run over collection of 60 processors. It takes longer to start many processes rather than a few, but even the longest time is negligible if processes are expected to run for hours, days, or even weeks. Time to restart a process is similar to time to start process.

We also measured the time it took to run uptime using `sh` and `rsh` on Sparc20 stations. For these measurements, we ran the command 100 times and took the average. These times are not strictly comparable to the `dsh` numbers presented, since we used the Unix `time` facility for measuring the overhead rather than `gettimeofday` system calls that we used inside our own programs. Despite this potential shortcoming of our measurements, it is instructive to note that run time using `dsh` is comparable to the best-case of starting the command locally. The reason that `dsh` is not faster is that it uses sh for parsing commands.

`rsh` is almost an order of magnitude slower than `dsh`. Doing this comparison is not entirely fair, since `rsh` and `dsh` have different goals. Among other overheads in `rsh`, a new `rsh` daemon is started for every invocation and the output of the process that is created has to be captured and returned to the user. With `dsh`, no daemons are started, and all processes run under a guest account so that authentication is unnecessary.

4.2.2 Restoring Consistency of Replicated Objects

One of the concerns with replicated objects is the time overhead involved in maintaining consistency. PEX restores consistency of replicated objects at view changes by either reconstructing objects of merging different versions. This section demonstrates that time overhead of restoring consistency is negligible compared to the time it takes to install a view change by the group membership service.

We did two experiments. In the first experiment, we measured the time it takes

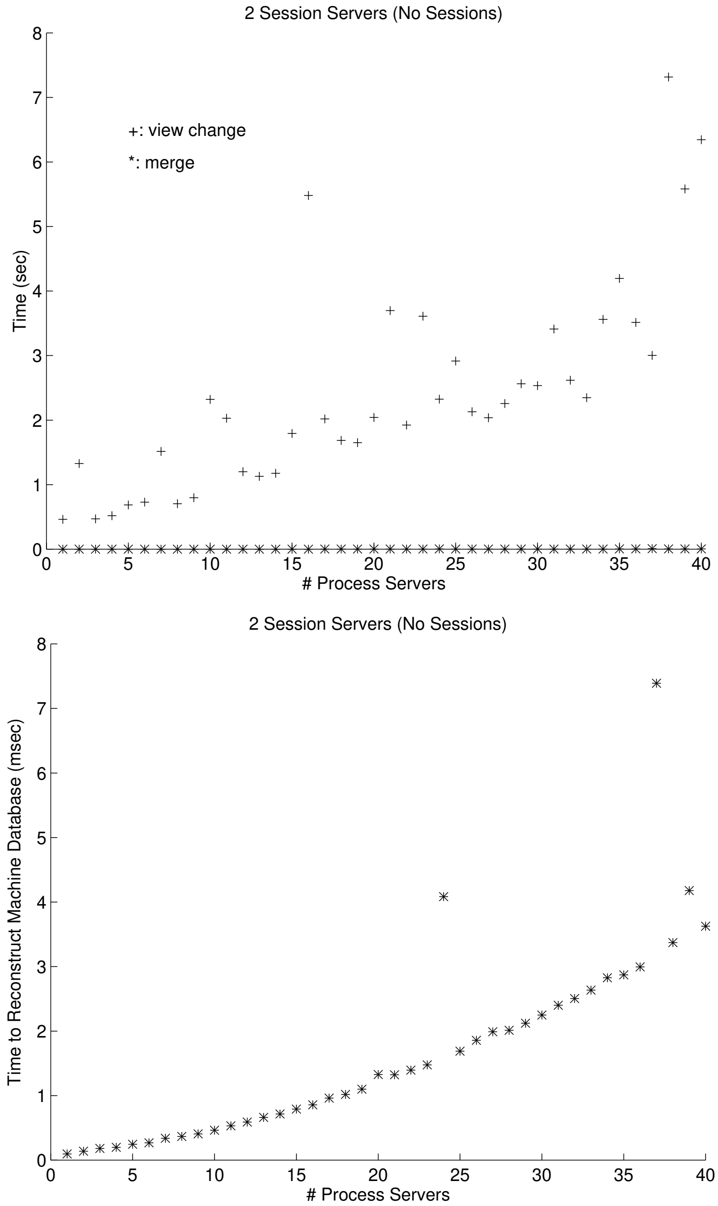

Figure 4.1: Time to reconstruct the processor database. The graph on the top includes the overhead of a new membership installation, while the graph on the bottom only shows the overhead of the database reconstruction.

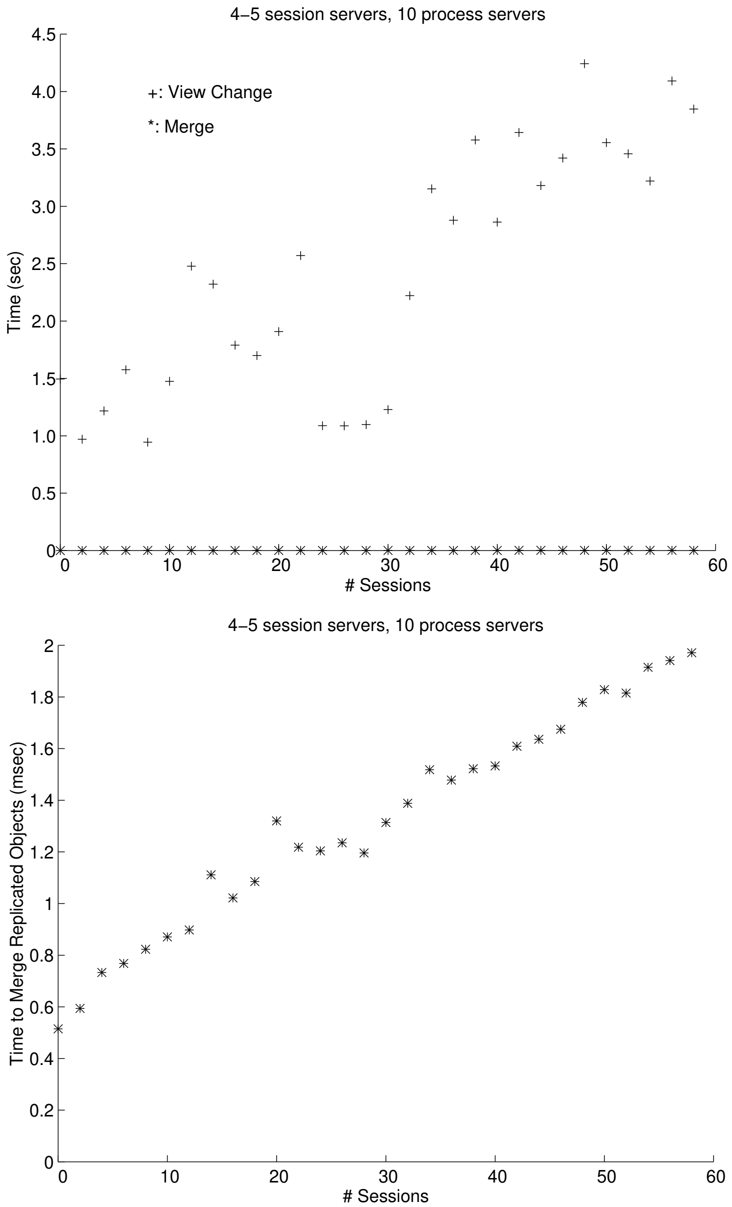

Figure 4.2: Time to merge divergent replicas of all objects. The top graph includes the overhead of membership installations, while the bottom graph excludes this overhead.

to reconstruct the processor database, while varying the number of process servers in the group. In the second experiment, we measured the time to merge session objects and request records, while varying the number of sessions. Each session contained a single process running Unix `sleep`. In both experiments, we ran the session servers on Pentium processors running LINUX and the process servers on SparcStations running SunOS.

PEX merges objects at view-change installations. Causing view changes to happen for our experiments requires adding or killing a session or process server. This happens automatically in the first experiment. For the second experiment, we induced view changes by periodically starting and crashing session servers.

Figure 4.1 shows the time it takes to reconstruct the processor database as a function of the number of process servers. The time to reconstruct the processor database is negligible compared to the view-change overhead. (The experiment was run only once for each number of process servers, and therefore outliers are visible.)

Figure 4.2 shows the time it takes to merge *all* replicated objects as a function of the number of sessions. This time includes reconstructing the processor database, but the overhead is approximately constant, since the number of process servers is now fixed. The merge overhead is approximately linear in the number of sessions and negligible compared to the overhead of view changes.

Is it worth using a group membership service, since it is the source of most overhead in PEX? We believe the answer is yes. The membership service provides

- automatic location and incorporation of new session and process servers,

- detection of their inaccessibility, so client requests do not stall, and
- ordered communication that facilitates replication.

Were we to build such facilities within PEX, then we would have had to write significantly more code, and we would probably not see much in the way of performance improvements at the end of that exercise.

4.2.3 Load Distribution

Ideally, a parallelizable task should take $1/n$th of the time using n processors compared to using a single processor of comparable type. In practice most tasks are not completely parallelizable; and, even if they are, the overhead involved in dividing a task into subtasks, distributing the subtasks, collecting the results, and merging them into a collective result is usually considerable.

A coarse-grained parallel system such as `omake` has limitations in how tasks can be parallelized. For example, `omake` cannot divide commands for each object into finer subcommands. Thus, the granularity of subtasks in `omake` is limited by commands for a single object to be made; therefore, the time a parallel task (in this case, making of multiple objects) takes is bounded below by the time it takes for its longest subtask (making of a single object).

Given this restriction, the ideal way is to divide a set of commands into such a way that the total execution time at each processor are roughly equal (or as equal as the set of commands can make). However, this requires knowledge of how long each task takes a-priori, and given those requires solving an optimal scheduling problem,

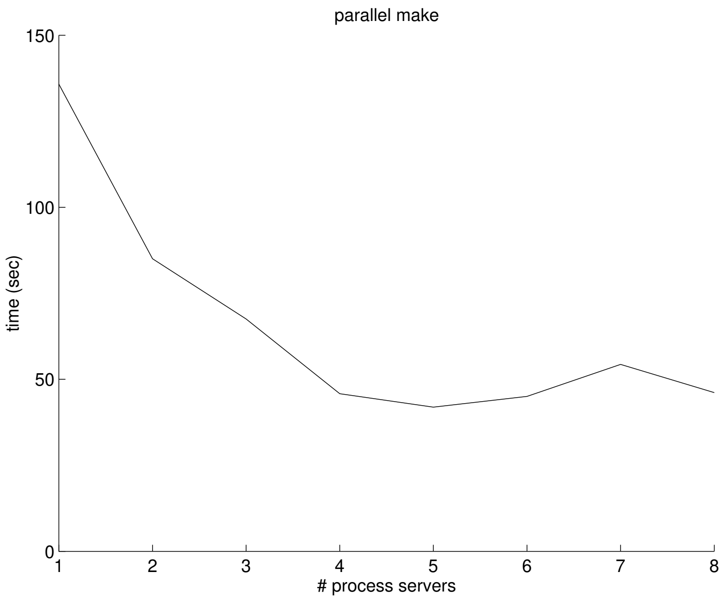

Figure 4.3: Time to make PEX using `omake`. It consists of 8 stages each with 3 to 10 concurrent tasks.

which is NP-complete [GJ79]. Currently `omake` simply assigns an equal number of jobs to each selected processor. This is a reasonable strategy if each subtask takes roughly the same amount of time. Even if this were not the case, it is not a large loss to not optimize `omake`. Given enough processors, building an object using `omake` is still faster than building an object using make, and cycles not used by some `omake` can be utilized by other unrelated tasks.

We measured the performance of `omake` by compiling various software systems. Figure 4.3 and Figure 4.4 show the result for building PEX and the Ensemble system respectively. For comparison, building these applications using make takes 130 seconds for PEX and 570 seconds for Ensemble on a comparable machine (SparcStation). We believe that the inefficiency of `omake` particularly for Ensemble, which

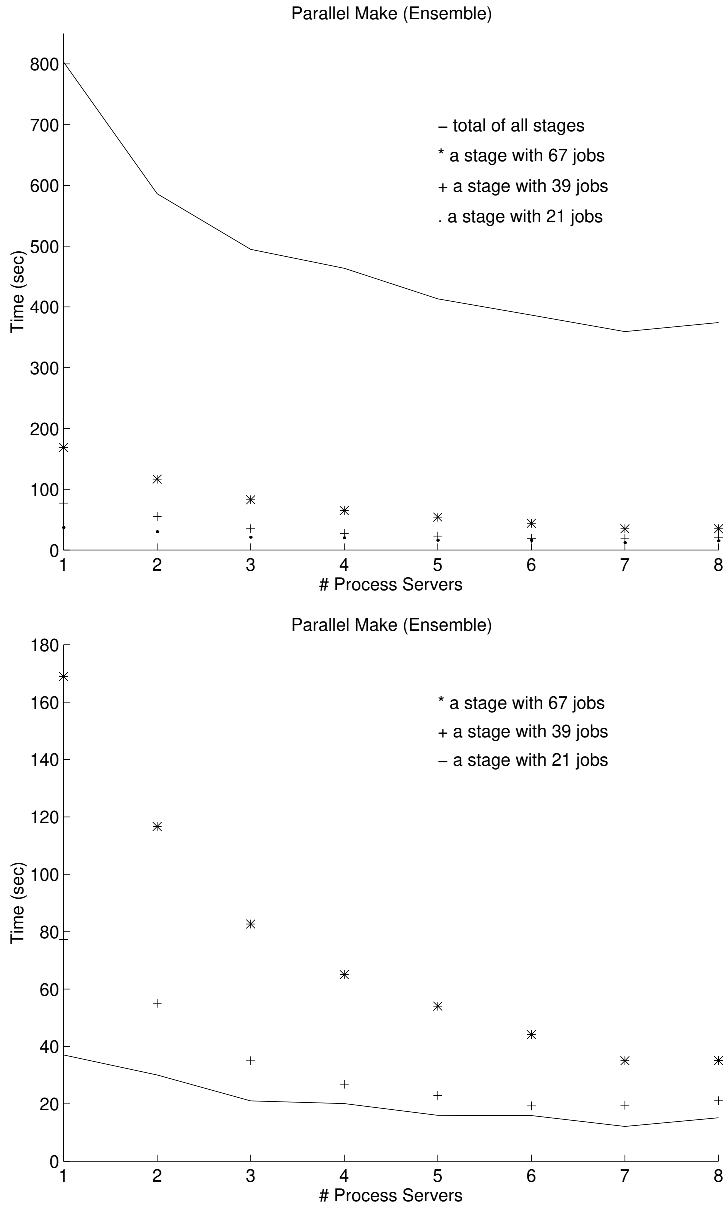

Figure 4.4: Time to make the Ensemble system using `omake`. It consists of 22 stages each with 1 to 67 concurrent tasks.

Table 4.4: Stages in making the Ensemble system. It consists of 22 serial stages, each consisting of two phases: find and make. In the find phase, it finds objects that should be and can be made. In the make phase, it makes objects found in the find phase in parallel via PEX. The time it takes in each phase using a single processor is shown.

stage	#objects	Total	Find	Make
1	3	53.84	51.07	2.77
2	4	45.08	42.15	2.93
3	1	45.02	40.57	4.45
4	3	38.03	30.65	7.38
5	3	37.08	14.28	22.80
6	21	37.04	10.51	26.53
7	8	21.06	6.95	14.11
8	7	18.03	4.19	13.84
9	16	23.20	1.90	21.30
10	39	77.26	1.25	76.02
11	15	32.04	1.91	30.13
12	13	25.09	0.29	24.80
13	14	41.07	0.15	40.92
14	12	28.08	0.12	27.96
15	67	168.94	0.31	168.63
16	13	35.11	0.24	34.88
17	8	31.19	0.04	31.15
18	2	9.01	0.06	8.95
19	1	24.03	0.01	24.02
20	1	9.02	0.06	8.96
21	1	1.00	0.01	0.99
22	1	1.00	0.01	0.99
total	253	801.26	206.72	594.54

is a much larger task, comes from using non-optimal data structures in `omake`. We hope to fix this in future.

The performance gain for executing a task obtainable with respect to an increase in the number of processors used for execution depends on how well the task can be parallelizable. Making PEX consists of eight sequential stages, each making 6, 10, 5, 6, 5, 5, 5, and 3 objects in parallel. Since most stages are parallelizable to 5 or 6 subtasks, we would hope to obtain a liner speed-up when using 5 or 6 processors. The result shows that, although worse than linear, `omake` does gain in performance up to 4 processors.

Table 4.4 shows stages of making the Ensemble system. It consists of 22 stages, each making 1 to 67 objects. Each stage consists of two phases: find and make. In the find phase, `omake` checks for objects that should be made (no up-to-date version exist and has not yet been attempted to be made in this pass of `omake`), and can be made (all its dependents are up-to-date). In the make phase, `omake` executes commands associated with objects to be made resulting from the find phase in parallel using PEX. The last two stages in making Ensemble involves making objects that have no associated commands (the last object is a phony object "all", which depends on another phony object "install" which involves copying some libraries into appropriate directories), and thus by checking for such cases, the time spent for the make phase can be eliminated. The first five stages involve making some directories, compiling architecture-dependent compilers, and collecting sources. In stages 6 to 8, most commands are compiling small objects, and some commands are linking libraries, which tend to be large.

The overall performance speed up of building the Ensemble system using 8 processors is 1/2, which is far worse than the ideal speed up of 1/8. This is because many stages are not parallelizable. The find phase is entirely sequential, and many stages make less than 8 objects in parallel. The gain is better in stages where parallelism is high. For example, the speed up is 1/4 using 8 processors for the stage that makes 67 objects.

Chapter 5

Conclusion

Distributed computing—seamlessly sharing of physically scattered information and resources—has been the Holy grail of networking computers. While the performance of individual computers has continued to grow, so has the data access rate required by emerging applications. In fact, the amount of data stored in the Internet has far exceeded the bandwidth the Internet is capable of carrying. The gap is likely to continue to widen for some time.

The goal of this thesis is to explore techniques for providing high-quality services in this environment. The two characteristics of the Internet are that it is low bandwidth and partitionable.

Making services available during partitions requires replication, but allowing replicas to be modified during partitions may introduce inconsistencies. In the first part of the thesis, we surveyed replication techniques suitable for a partitionable network. These techniques provide continuous availability while managing inconsistencies among replicas.

In the second part of the thesis, we presented a fault-tolerant execution service in a partitionable low bandwidth environment: PEX. PEX is designed for organizing long running parallel computations on massive data. Examples of such applications are scientific computations that detect patterns in observational data. The amount of observational data is so huge that the Internet is not capable of transferring it in a reasonable amount of time. Computations on this data are usually extremely parallelizable and long running. PEX allows remote users to place tasks near where the data is stored and monitor related tasks easily. Internally PEX replicates state so that it can tolerate partition failures.

We implemented PEX over a group communication system. The performance of PEX is reasonable. The implementation demonstrates the usefulness of the powerful group communication paradigm.

BIBLIOGRAPHY

[ACM83] ACM SIGOPS. *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, Bretton Woods, New Hampshire, October 1983.

[ACM97] ACM SIGACT-SIGOPS. *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, Santa Barbara, CA, USA, August 1997.

[ADKM92] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication subsystem for high availability. In *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[BDMS97] Ozalp Babaoglu, Renzo Davoli, Alberto Montresor, and Roberto Segala. System support for partition-aware network applications. Technical Report UBLCS-97-08, University of Bologna, October 1997.

[BG83] P. A. Bernstein and N. Goodman. The failure and recovery problem for replicated databases. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 114–122, Montreal, Quebec, Canada, August 1983. ACM SIGACT-SIGOPS.

[Bir96] Kenneth P. Birman. *Building Secure and Reliable Network Applications.* Manning, 1996.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BKK+96] Ronald J. Brachman, Tom Khabaza, Willi Kloesgen, Gregory Piatetsky-Shapiro, and Evangelos Simoudis. Mining business databases. *Communications of the ACM*, 39(11):42–48, November 1996.

[BMST93] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In Sape Mullender, editor, *Distributed Systems*, pages 199–215. ACM Press, New York, 2 edition, 1993.

[CAV96] Francois J.N. Cosquer, Pedro Antunes, and Paulo Verissimo. Enhancing dependability of cooperative applications in partititionable environments. In *Proceedings of the Second European Dependable Computing Conference*, October 1996.

[Che88] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[CM92] Henry Clark and Bruce McMillin. DAWGS—a distributed compute server utilizing idle workstations. *Journal of parallel and distributed computing*, 14:175–186, 1992.

[CSL96] Chungmin Chen, Kenneth Salem, and Miron Livny. The DBC: Processing scientific data over the internet. In *Proceedings of the 16th International Conference on Distributed Computing Systems*. IEEE Computer Society, 1996.

[cvs] CVS. The UNIX manual.

[DGMS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[ELvD+96] D.H.J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of Condors: Load sharing among workstation clusters. *Journal on Future Generations of Computer Systems*, 12, 1996.

[ET89] Amr El Abbadi and Sam Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2), June 1989.

[FHS96] Usama Fayad, David Haussler, and Paul Stolorz. Mining scientific data. *Communications of the ACM*, 39(11):51–57, November 1996.

[FM82] Mishael J. Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable netowork. In *Proceedings of the ACM symposium on Principles of Database Systems*, March 1982.

[FPSS96] Usama Fayad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34, November 1996.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, California, December 1979. ACM SIGOPS.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, N.Y., 1979.

[GMPS96] Clark Glymour, David Madigan, Daryl Pregibon, and Padhraic Smyth. Statistical inference and data mining. *Communications of the ACM*, 39(11):35–41, November 1996.

[GMS+98] Henry M. Gladney, Fred Mintzer, Fabio Schiattarella, Julian Bescos, and Martin Treu. Digital access to antiques. *Communications of the ACM*, 41(4):49–57, April 1998.

[Guy91] Richard G. Guy. *The Ficus: A very large scale reliable distributed file system*. PhD thesis, University of California, Los Angeles, June 1991.

[Hay98] Mark Hayden. *The Ensemble System*. PhD thesis, Cornell University, January 1998.

[JdT+95] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 156–171, Copper Mountain Resort, Colorado, December 1995. ACM SIGOPS.

[JPR+83] D. Stott Parker Jr., Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983.

[Lam78] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Ler98] Xavier Leroy. *The Objective Caml system release 1.07*. INRIA, Paris, France, March 1998.

[LLM88] Michael J. Lizkov, Miron Livny, and Matt W. Mutka. Condor – a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, San Jose, California, USA, June 1988. IEEE Computer Society.

[Lot92] M. Lottor. Internet growh (1981-1991), January 1992. RFC 1296.

[mak] make. The UNIX manual.

[MAMSA94] L.E. Moser, Y. Amir, P.M. Melliar-Smith, and D.A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56–65, Poznan, Poland, June 1994. IEEE Computer Society.

[MFSW95] Christoph Malloth, Pascal Felber, Andre Schiper, and Uwe Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. In *Proceedings of workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, TX, October 1995.

[Mit97] Michael Mitzenmacher. How useful is old information? In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing* [ACM97].

[MMSA+96] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, April 1996.

[Moe98] William E. Moen. Accessing distributed cultural heritage information. *Communications of the ACM*, 41(4):44–48, April 1998.

[MW82] T. Minoura and G. Wiederhold. Resilient extended true-copy token scheme for a distributed database system. *IEEE Transactions on Software Engineering*, SE-8(3):173–189, May 1982.

[NAS] NASA. Earth Observation System. http://eos.nasa.gov.

[Nic87] David A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 5–12, Austin, Texas, November 1987. ACM SIGOPS.

[PCGMW98] Andreas Paepcke, Chen-Chauan K. Chang, Hector Garcia-Molina, and Terry Winograd. Interoperability for digital libraries worldwide. *Communications of the ACM*, 41(4):33–43, April 1998.

[PM83] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles* [ACM83], pages 110–119.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 288–301, Saint-Malo, France, October 1997. ACM SIGOPS.

[ROR95] Aleta Ricciardi, Michael Ogg, and Eric Rothfus. The Nile system architecture: Fault-tolerant, wide-area access to computing and data resources. In *Proceedings of Computing in High Energy Physics*, Rio de Janeiro, September 1995.

[SBN82] Michael D. Schroeder, Andrew D. Biirrell, and Roger M. Needham. Grapevine: An excercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[Sch93] Fred B. Schneider. Replication management using the state machine approach. In Sape Mullender, editor, *Distributed Systems*, pages 169–195. ACM Press, New York, 2 edition, 1993.

[SKK+90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[SW84] D. Skeen and D. Wright. Increasing availability in partitioned networks. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 290–299. ACM SIGACT-SIGMOD, April 1984.

[Tic85] Walter F. Tichy. RCS–a system for version control. *Software–Practice and Experience*, 15(7):637–654, July 1985.

[TKDA96] Fredy Tandiary, Suraj C. Kothari, Ashish Dixit, and E. Walter Anderson. Batrun: Utilizing idle workstations for large-scale computing. *IEEE Parallel and Distributed Technology*, 4(2):41–47, summer 1996.

[TLC85] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 2–12, Orcas Island, Washington, December 1985. ACM SIGOPS.

[vRBM96] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[VRC+95] P. Verissimo, L. Rodrigues, F. Consquer, H. Fonseca, and J. Frazao. An overview of the NavTech system. Technical Report RT-95, INESC, 1995.

[Wiz] Network Wizards. Internet domain survey. http://www.nw.com/zone/WWW/report.html.

[WNMMC98] Ian H. Witten, Craig Nevill-Manning, Roger McNab, and Sally Jo Cunningham. A public library based on full-text retrieval. *Communications of the ACM*, 41(4):71–75, April 1998.

[WPE+83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles* [ACM83], pages 49–70.

[Wri83] D. D. Wright. Managing distributed databases in partitioned networks. Technical Report TR83-572, Department of Computer Science, Cornell University, Ithaca, NY, September 1983.