An Execution Service for a Partitionable Low Bandwidth Network

Takako M. Hickey and Robbert van Renesse[*]

Abstract

As the amount of scientific data grows to the point where the Internet bandwidth no longer supports its transfer, it becomes necessary to make powerful computational services available near data repositories. Such services allow remote researchers to start long-running parallel computations on the data. Current execution services do not provide remote users with adequate management facilities for this style of computing.

This paper describes the PEX system. It has an architecture based on partitionable group communication. We describe how PEX maintains replicated state in the face of processor failures and network partitions, and how it allows remote clients to manipulate this state. We present some performance numbers, and close with discussing related work.

1 Introduction

Dozens of particle accelerators around the world individually collect terabytes of data each year from experiments [23]. Satellites send terabytes worth of observation to earth each day [19]. 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.

Thus it makes sense to set up computation services close to where the data is stored. Typically, computation on this type of data (observations) can be easily parallelized, resulting in significant speed-up. The individual computations require little or no interaction, and 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 scientific computations require hundreds of processes [23, 19]. Creation of the processes is relatively simple, but keeping track of hundreds is difficult.

As on individual computers, there is a need for a processing 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 be able to terminate a session with a single command. 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. 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 long-running, parallel computation;
- support process placement based on application-defined placement functions (such as data location);
- be tolerant of processor failures and network partitions;
- support heterogeneity of CPU types and operating systems;
- support cooperation between multiple, remote users.

This paper describes the *PEX service*. The contributions of this work are:

- a study of issues related to partitionable operation of distributed services, and remote execution services in particular;
- a demonstration of the use of group communication protocols in such services;

[*]The work reported was supported in part by ARPA/ONR grant N00014-96-J-1014, ARPA/RADC grant F30602-96-1-0317, NASA/ARPA grant NAG-2-893, and AFOSR grant F49620-94-1-0198.

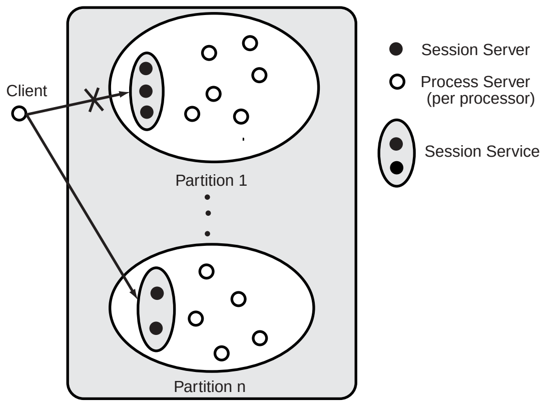

Figure 1. PEX architecture

- a software architecture and prototype implementation of a remote execution service, as well as experimental results.

The rest of the paper is organized as follows. Section 2 presents the architecture. In Section 3, we present the client interface. Section 4 describes a reliable RPC mechanism that is used to implement this interface. In Section 5, we describe how PEX deals with processor failures and network partitions. Section 6 reports on the performance of the initial implementation. Section 7 compares PEX to other work, and Section 8 gives conclusions.

2 The PEX Architecture

PEX makes use of a *partitionable group membership service* [3]. The service provides consistent information to the processes about the partition that they are in, and supports ordered communication within each partition.

Figure 1 shows the PEX architecture. An *institute* is a collection of processors connected by a small collection of local area networks, such as those used in academic departments or medium-sized companies. Each institute organizes its processors in a *processor group*. Each processor runs a *process server*, which handles execution and termination of processes running on that processor. A small number of processors (typically, two or three) each run a *session server*. Session servers together provide the *session service*, which provides an access point for users (within or outside the institute) that want to use processors in the processor group. We will use the term *client* to describe users of the session service.

The partitionable membership service manages both the processor group, which is large, and a group of session servers, which is small. The session service group needs to be small, because it maintains replicated information that would be hard to scale to the entire processor group.

The session service maintains three types of replicated objects: the processor database, sessions, and records for outstanding user requests. The *processor database* maintains information about the processors in the local processor group. For each processor, it maintains attributes such as CPU type and speed, operating system, amount of physical memory, local data sets, accessibility, and load. 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.

Sessions are sets of processes currently running or recently terminated. Having sessions simplifies management of related processes and facilitates cooperation between clients. With PEX arbitrating user requests to the shared session, multiple clients can safely manage an application concurrently. All processes in a session have to run in the same processor group. Sessions and the processor database are replicated to improve availability and to allow load sharing.

Client request records describe on-going and recently completed client requests. They are replicated to avoid the loss of a request or reply when a session server fails, and also to filter duplicate requests submitted to more than one session server.

Before creating a session, a client has to decide on the institute where the session should run. Currently, we assume that the client knows which institute stores the data set that the computation needs; eventually, we hope to use a fault-tolerant directory service for storing and retrieving this information.

3 Management Interface

This section describes the RPC interface that clients use to manipulate sessions. The major types of requests are listed in Table 1.

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 *recovery instruction*. The reply to `Process-Create` 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 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 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`.

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 per partition per view of client requests. 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 that the session service uses to ensure at-most-once execution of received client requests. Together they provide the desired behavior (see Figure 2).

4.1 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

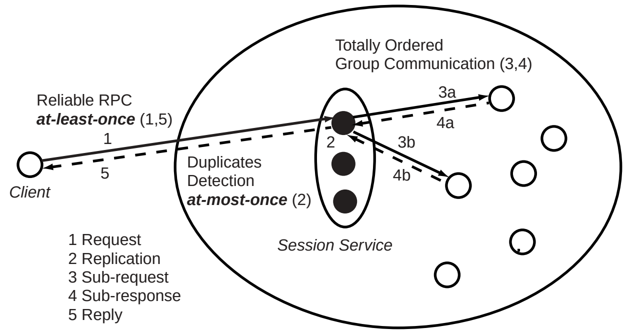

Figure 2. Exactly-once execution

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 certainly down, while 1 implies that it is certainly 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, 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.

4.2 At-Most-Once Semantics

This section describes mechanisms that the session service uses to detect duplicate requests to ensure at-most-once execution of each client request. Combined with the at-

Request	Arguments	Response
SessionCreate		session-id
ProcessCreate	session-id, processor specs, process properties, command list, 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

Table 1. Session management interface

least-once delivery provided by our RPC facility, exactly-once execution of each client request is achieved.

As mentioned previously, 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.

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. A simple way of achieving this is by having the request manager multicast the request to the session service group. Each server executes the request [25]. 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. More importantly, if request processing is non-deterministic, this approach does not work because the state of the servers would diverge. In our case, selection of processors is non-deterministic, as we see later.

These problems can be cured by adopting a primary-backup approach, where one of the servers, called the *primary*, carries out the request. The primary then multicasts a *state update* message to the other servers [4]. The state update message describes how the backups should modify their state to replicate the primary. The main disadvantage of this approach is that a failure of the primary means that a fail-over delay is incurred (due to the length of failure detection and the subsequent agreement protocol among the backups).

To get the best of both approaches, PEX uses a hybrid protocol. 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.

When a session server receives a forwarded request, the server checks whether it has a *client-request record* with the same RPC identifier. If it does not, the request is new and the server creates a client request record 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 record. 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.

If a session server receives a client request for which it already has a client-request record, 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.

5 Restoring Consistency

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

Crashes and partitions are announced by the group membership service in the form of *view-changes*. In asynchronous distributed systems, it is impossible to distinguish the case of a processor being failed from the case of slow processor. The group membership protocol "solves" this problem by erroneously (but consistently) marking a slow processor as having failed. This is practical, because computation on other processors can then continue unimpeded.

A problem created by this solution is that a processor that joins a group might not be a newly started member with no state, but an old member that has 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. To our knowledge, all existing group communication systems, if they support partitioned operations at all, adopt the primary partition state at view-changes. While algorithmically simple, this approach is unsatisfying when dealing with long running computations; by throwing away the state of a minority partition it is possible to lose the results 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: processor database, sessions (which contain process objects), and client-request records. We present state-merge algorithms for each of these.

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 in the future may adopt a more sophisticated algorithm similar to those used in the other replicated objects described below.

5.2 Merging Session Object Replicas

A session is described by a *session object*, which in turn consists of a set of *process objects*. After a partition there may be multiple concurrent versions of the same session object, and ditto for a process object. 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 (live) process. 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 killed
- 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-request records 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, P1 and P2, of the same process object is shown in Figure 3. 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 one is used; if both are dead, it is placed into kill or restart set depending on the client's 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

```
if Svr1 = Svr2 & alive(Svr1) then
    add(live, P1)
else if alive(Svr1) & dead(Svr2) then
    add(live, P1)
else if dead(Svr1) & alive(Svr2) then
    add(live, P2)
else if dead(Svr1) & dead(Svr2) then
    if to_restart(P1) then
        add(restart, P1)
    else
        add(dead, P1)
else if rank(Svr1) < rank(Svr2) then
    add(live, P1)
    add(kill, P2)
else
    add(live, P2)
    add(kill, P1)
```

Figure 3. Merging process object

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 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.

5.3 Merging Client-Request Record Replicas

As in merging session objects, merging client-request records proceeds in two phases. A request record contains sub-request records, 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 request record versions at random in its entirety.

For requests that call for operations on all processes in a session (e.g., `SessionWait`), this approach might overlook some of these processes. This happens if processes not known by the partition where the 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 request records 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 request record 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 of merging request records makes sure that requests represented are either being served or, if they are done, a reply is sent back to the client. [14] describes this algorithm in more detail.

6 Performance

PEX was written in Ocaml [16] over the Ensemble group communication system [13]. We used two applications for performance measurements: `dsh` and `omake`. `dsh` is a simple shell-like application that reads in a list of commands to execute along with pre-defined composable machine selection criteria (e.g., [18] describes criteria that work well even when processor load information is old). These are submitted to the execution service using the API of Section 3. `omake` is a parallel make program.

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

6.1 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 receive 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 2 shows the results for varying numbers of process servers. 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.

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 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.

6.2 Merging Divergent Versions

One of the concerns with replicated objects is the time overhead involved in maintaining consistency. PEX merges versions of replicated objects on view-changes. This section demonstrates that time overhead of merging replicated objects 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 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.

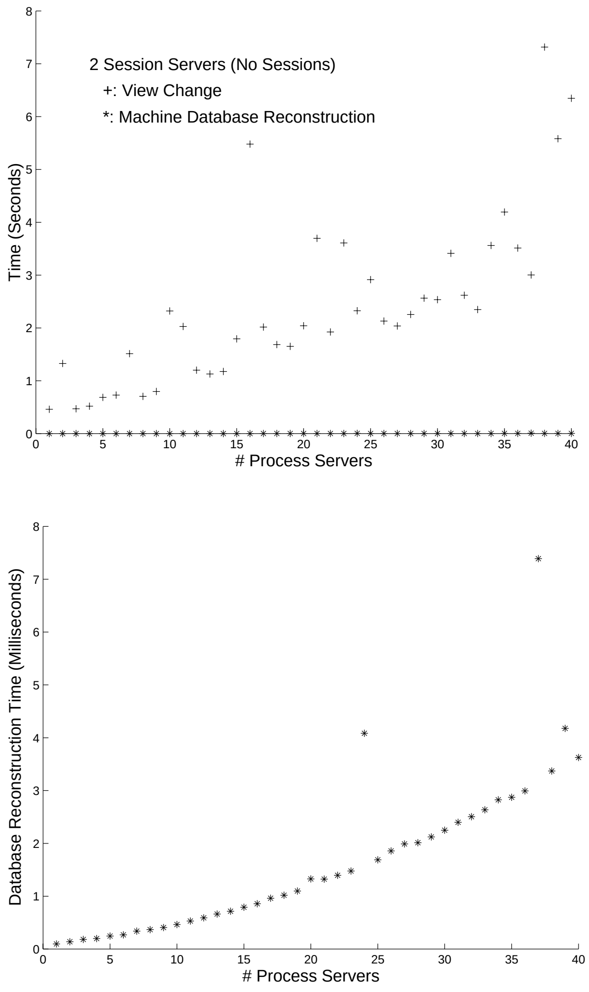

Figure 4. Processor database reconstruction

Figure 4 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 5 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.

Our experiment results raises a question: 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

Command	#Servers	Total (msec)	SessCreate	ProcCreate	SessDestroy
`uptime`	1	140			
`sh`	1	210			
`rsh`	1	1700			
`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

Table 2. Performance of basic operations

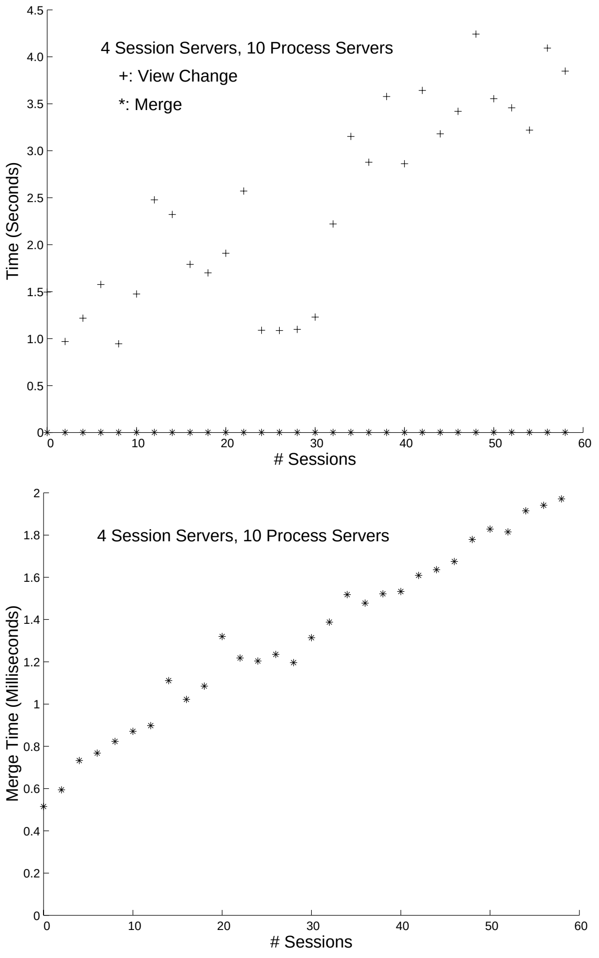

Figure 5. Merging divergent objects

- 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.

6.3 Load Distribution

Ideally, a parallelizable task should speed up by a factor of n when using n processors instead of 1. In practice, most tasks require internal synchronization so that this linear speed-up cannot be attained. Even in the absence of synchronization, 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. These limitations are particularly true for any *parallel make* facility, and our `omake` is no exception.

The most common application of `omake` is to compile and link together the components of some software system. `omake` tries to do as many of its compilations in parallel as possible. `omake` cannot divide compilations into finer subcommands. Given this restriction, the ideal parallelization is to divide the set of all compilations so that the total execution time at each processor is roughly equal. Even if the compilation time of each component was known a priori, this problem is known to be NP-complete [11]. But these compilation times are not known, and there are dependencies between the tasks. That is, some compilations have to complete before others.

Currently, `omake` runs a loop in which it calculates the current set of components that can be compiled (that is, they have no dependencies left), and assigns them to processors as they become available on a first-come, first-served basis. Although not optimal, it does provide a significant speed-up, and unused processor cycles can be utilized by other unrelated tasks.

We measured the performance of `omake` by compiling various software systems. Figure 6 and Figure 7 show the results for building PEX and the Ensemble system respectively. For comparison, building these applications using a standard UNIX *make* takes 130 seconds for PEX and 570 seconds for Ensemble on a single machine.

Making PEX, because of the dependencies between the components, consists roughly of eight sequential stages,

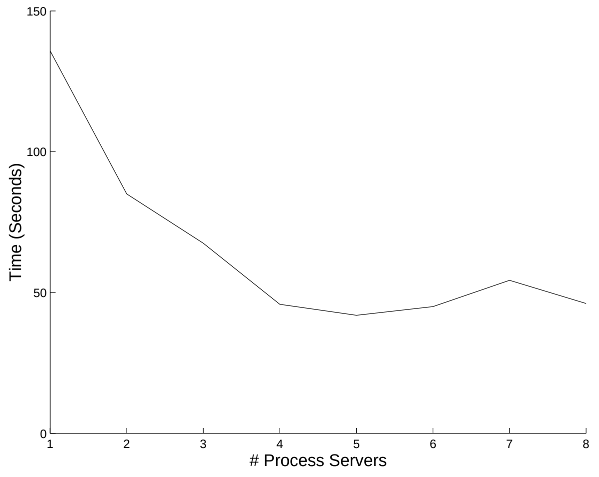

Figure 6. Parallel making PEX

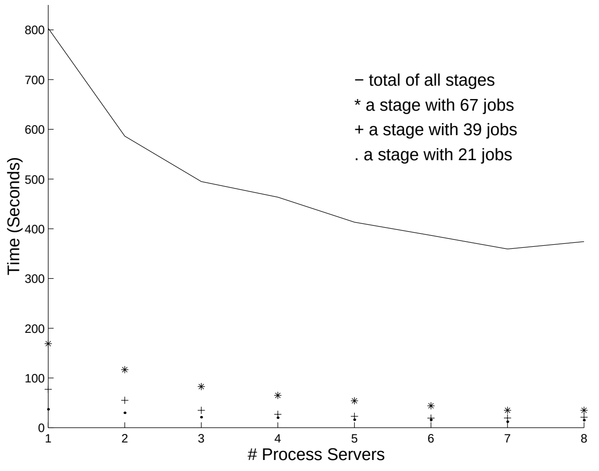

Figure 7. Parallel making Ensemble

each making 6, 10, 5, 6, 5, 5, 5, and 3 objects in parallel, or about 5 on average. Each stage consists of two phases: the *find phase* and the *make phase*. In the find phase, `omake` finds objects that can be made, that is, no up-to-date version exist already, the object is not being made, and all its dependents are up-to-date. In the make phase, `omake` executes the commands to make these objects. The reason that a speed-up of 4 is obtained, rather than 5, comes from the fact that the find phase is not parallelizable, sub-optimal scheduling, and from overheads in PEX itself.

Making Ensemble consists of 22 stages, each making 1 to 67 objects. 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 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 overall performance speed up of building the Ensemble system using 8 processors is only 2, since little parallelism is possible in most stages, and these stages are lengthy. The gain is better in stages where parallelism is high. For example, the speed up is 4 using 8 processors for the stage that makes 67 objects.

7 Related Work

The PEX project is concerned with two sets of issues: load sharing in a distributed system, and partitionable operation. For load sharing, we only consider those projects that do not make restrictions on programming languages used to write applications. Partitionable operation has been primarily studied in the context of database and file systems, not in execution services.

The first wave of interest in sharing of workstations occurred in the early 1980s, when workstations became cheap enough so 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 distributed operating system features that would support such sharing. The efforts include Demos/MP [22], V [27], and Butler [20]. Later projects include DAWGS [6], Condor [17, 10], Batrun [26], DBC [5], and NOW [1].

Most of these systems assumed homogeneous processors, and none allowed placement of processes based on data availability. Only DAWGS replicates objects, namely its processor database. However, DAWGS does not guarantee consistency of this database.

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 selected machine does not have the necessary data, DBC copies the data to the selected machine before starting the job. This works well if the amount of data to be copied is small, but not for large data sets, or data protected by licenses or copyright.

Fault-tolerance and partitionable operation are important topics studied by researchers in databases, network file systems, and version control systems. Here multiple versions of an object are allowed to co-exist and independently. This increases the availability of objects, supports disconnected operation (e.g., for laptops), and allows sharing of an object by multiple users. The main challenges are detecting inconsistencies and reconciling divergent versions of objects.

For database systems, [8] and [9] give good coverage of representative merging techniques. Files have weaker semantics than databases, so simpler techniques can be used there. The LOCUS file system [30] developed a simple conflict detection mechanism based on *version vectors*. A version vector is an integer array with an entry for each replica. Each entry is the number of updates that has been made to the corresponding replica. Version vectors can be used to detect multiple writer conflicts, but not read-write conflicts. The LOCUS designers reasoned that this shortcoming is acceptable for most file systems, where most files are independent of each other. LOCUS also developed a conflict resolution algorithm for directories. For other files, manual intervention is usually necessary. LOCUS also includes a transparent execution environment for homogeneous processors.

Ficus [12], a successor project to LOCUS, concentrates only on the file system. It extends the version vector algorithm so that the number of replicas can be dynamically changed. Many other partitionable file systems that tolerate partitions adopt the basic approach of using version vectors. These include Coda [24], Bayou [21], and Rover [15].

RCS [28] is a source code control system that uses, for merging divergent versions of an object, an algorithm based on *diff listings*. Suppose that two users independently updated the same file. The RCS algorithm then produces two diff listings, comparing the respective versions with the original file. Each diff listing describes how to create the version from the original. Then both diff listings are applied to the original to create a merged version. If two updates are near each other in the same file, that section is marked to warn the user. This does not always work correctly, but it simplifies the task of merging considerably.

Recently partitionable operation has been getting more attention in group communication systems. The latest trend 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 such systems are NavCoop [7] and Relacs [2].

NavCoop is a merging tool built over the partitionable system NavTech [29]. 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 and merge functions based on the relative importance of sub-groups. Relacs is a partitionable group communication system, much like Ensemble, and they suggest a number of partitionable applications. To our knowledge PEX is the only application actually implemented that handles non-trivial merging of replicated objects.

8 Conclusion

PEX is an execution service for a partitionable low bandwidth network. PEX maintains highly available and long-running sessions of processes on behalf of users anywhere on the Internet. Besides the service itself, the contributions of our work include techniques for maintaining replicated objects that can partition and re-merge, and a demonstration of the use of the group membership paradigm.

Acknowledgments

The initial idea for building a remote execution service grew out of discussions with Mark Hayden. We thank Gary Adams, Ken Birman, Jason Hickey, Fred Schneider, and anonymous referees for comments on drafts of this paper.

References

[1] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for now (networks of workstations). *IEEE Micro*, pages 54–64, February 1995.

[2] Ozalp Babaoglu, Renzo Davoli, Alberto Montresor, and Roberto Segala. System support for partition-aware network applications. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, October 1998. IEEE Computer Society.

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

[4] 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.

[5] 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.

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

[7] 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.

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

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

[10] 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.

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

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

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

[14] Takako M. Hickey. *Availability and Consistency in a Paritionable Low Bandwidth Network*. PhD thesis, Cornell University, August 1998.

[15] 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.

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

[17] 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.

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

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

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

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

[30] 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* [**?**], pages 49–70.