REXEC: A Decentralized, Secure Remote Execution Environment for Parallel and Sequential Programs

Brent N. Chun and David E. Culler

University of California at Berkeley
Computer Science Division
Berkeley, CA 94720
Tel: (510) 642-8299
Fax: (510) 642-5775
{bnc,culler}@cs.berkeley.edu

Abstract

Bringing clusters of computers into the mainstream as general-purpose computing systems requires that better facilities for transparent remote execution of parallel and sequential applications be developed. While much research has been done in this area, most of this work remains inaccessible for clusters built using contemporary hardware and operating systems. Implementations are either too old and/or not publicly available, require use of operating systems which are not supported by modern hardware, or simply do not meet the functional requirements demanded by practical use in real world settings. To address these issues, we designed REXEC, a decentralized, secure remote execution facility. It provides high availability, scalability, transparent remote execution, dynamic cluster configuration, decoupled node discovery and selection, a well-defined failure and cleanup model, parallel and distributed program support, and strong authentication and encryption. The system is implemented and is currently installed and in use on a 32-node cluster of 2-way SMPs running the Linux 2.2.5 operating system.

1 Introduction

With current technology trends, clusters of computers [2, 5, 7, 22] are becoming an increasingly popular design point for building scalable, high-performance computing systems. One of the key issues in the design of such systems is how users discover, select, and transparently access resources on the cluster to run their applications. To address this issue, clusters today often rely on fragile remote execution systems that are built using standard point-to-point facilities such as rsh, which do not provide transparent remote execution. As a result, robustness, management complexity, usability, and performance of these systems often suffer. To push clusters into the mainstream as general-purpose computing systems, better environments are needed for transparent remote execution of parallel and sequential applications.

Transparent remote execution systems for clusters have been explored numerous times in the past in a variety of forms. User-level approaches [4, 11, 13, 14, 18, 30], modification to existing operating systems [1, 3, 15, 23, 24], and entirely new distributed operating systems [20, 25, 27, 29] have all been implemented. Unfortunately, while considerable experience has been gained from these efforts, most of this work remains inaccessible to modern clusters built using off-the-shelf hardware and contemporary operating systems. Implementations are either too old and/or not publicly available, require use of operating systems which are not supported by modern hardware, or simply do not meet the functional requirements demanded by practical use in real world settings.

We have designed and implemented a new remote execution environment called REXEC [1] to address these issues. Building on previous work in remote execution and practical experience with the Berkeley NOW and Millennium clusters, REXEC provides the following features: decentralized control, transparent remote execution, dynamic cluster membership, decoupled node discovery and selection, a well-defined error and cleanup model, support for sequential programs as well as parallel and distributed programs, and user authentication and encryption. It takes advantage of modern systems technologies such as IP multicast and mature OS support for threads to simplify its design and implementation. The system is implemented almost entirely at user-level with small modifications to the Linux 2.2.5 kernel. It is currently installed and in use on a 32-node cluster of 2-way SMPs as part of the UC Berkeley Millennium Project.

The rest of this paper is organized as follows. In Section 2, we state our design goals and assumptions for the REXEC system. In Section 3, we describe the REXEC system architecture and our implementation on a 32-node cluster of 2-way SMPs running the Linux operating system. In Section 4, we discuss three examples of how REXEC has been applied to provide remote execution facilities to applications. In Section 5, we discuss related work. In Section 6, we conclude the paper.

[1] Our REXEC system has no relation to the 4.2 BSD rexec function, nor does it have any relation to the rexec command used in the Butler [18] system or the rexec function in NEST [1].

2 Design Goals and Assumptions

In this section, we describe our design goals and the assumptions made in designing REXEC. Our design goals are based on several years of practical experience as users of the Berkeley NOW cluster, a thorough examination of previous systems work in remote execution, and a desire to combine and extend key features in each of the systems into a single remote execution environment. Our assumptions are typical of remote execution systems and not overly restricting or extensive. Modern clusters built using off-the-shelf hardware and contemporary operating systems are easily configured to satisfy these assumptions.

2.1 Goals

High availability. The system should be highly available and provide graceful degradation of service in the presence of failures. There should be no single point of failure in the remote execution system, nor should there be any artificial dependencies between a failure in one part of the remote execution system and a perfectly functional node (e.g., by having a centralized system coordinator). Any node that is functional and running the REXEC system software should be available to run applications through the REXEC system.

Scalability. Performance of the remote execution system should scale as the size of the system grows. More specifically, as more nodes are added and more applications are run on the system, remote execution overhead should scale gracefully.

Transparent remote execution. Execution on remote nodes should be as transparent as possible, given the implementation approach (e.g., kernel-level implementations can achieve greater transparency than user-level implementations). At a minimum, signals, stdin, stdout, stderr, and the user's local environment (current working directory, user ID, group ID, environment variables) should be propagated. In addition, local job control should be applicable for controlling remote jobs. An implementation of this core set of features provides sufficient transparency for a large number of applications. File access is another important issue and is discussed later in this section.

Minimal use of static configuration files. The remote execution system should rely on as few static configuration files as possible. In particular, static files related to cluster configuration should be avoided. In our experience, node membership in a large cluster tends to change over time as machines are added, removed, crash, get brought down, rebooted, and so forth. Having to manually update configuration files every time a change occurs is error-prone and inefficient. The configuration of the cluster should be discovered dynamically.

Decoupled discovery and selection. The process of discovering which nodes are in the cluster and what their state is should be separated from the selection of which nodes to run an application on. This gives users flexibility to apply their own criteria to select which nodes are best suited for running their applications. For example, one user might want to select the nodes with the lowest CPU utilization, another might want to select nodes based on physical memory available, and yet another might want to run on a specific set of nodes.

Well-defined failure and cleanup models. The system should provide well-defined models for failure and cleanup. Remote processes controlled through a local point of control introduce new failure modes (e.g., network connection failure from a local proxy process to a remote process). Remotely controlled parallel or distributed processes add additional failure modes (e.g., one process in a n process program crashes). In addition, there are practical issues as well that arise due to user-level implementations (e.g., cleanup for remote processes that fork, child processes whose parents may exit causing the child to be inherited by init, etc.). A user-level remote execution system that supports parallel jobs should extend the failure and cleanup model of local sequential execution to account for these differences.

Parallel and distributed program support. The remote execution environment should provide a minimal set of hooks that allow parallel and distributed runtime environments to be built. For example, when running a parallel program, each remote process in the program often needs to know how many processes comprise the program, that process's rank in an ordering of those processes, and information which can be used to resolve network addresses of those processes so communication can be established.

Security. The system should provide user authentication and encryption of all communication. For clusters that are unprotected by firewalls, strong authentication prevents outside attackers from launching brute-force attacks on weak security or reverse engineering and exploiting schemes that rely on security by obscurity (e.g., GLUnix authentication was based simply on the UID passed from the client process to the master). For internal attackers, encryption prevents users from snooping the network to discover passwords, an increasingly common occurrence in large public institutions.

2.2 Assumptions

Uniform file pathnames. We assume that all shared files are accessible on all nodes using the same pathnames. We also assume that most local files on each node are also accessible under the same pathnames (e.g., /bin/ls). Most clusters today are built using off-the-shelf hardware and run either a free operating system such as Linux or a commercial operating system like Solaris. The majority of these systems do not support a completely transparent shared filesystem such as the ones used in Sprite [20] and Solaris MC [15, 24]. Instead, each node tends to use private, local filesystems and some of number of network file systems which are attached at common mount points on all nodes of the cluster. Our cluster, which uses Linux and NFS as our network file system, has this configuration.

All nodes run the same OS with compatible software configurations. We assume all nodes in the cluster run compatible versions of the operating system and have compatible software configurations. Compatible versions means that application binaries are supported on different versions of the operating system (e.g., Linux 2.2.5 versus Linux 2.2.10). Compatible software configurations means that all the necessary support for applications is available for applications. Having the same version of the operating system is not sufficient. (For example, applications may require use of certain shared libraries.) We do not attempt to address heterogeneity issues that meta-computing systems [9, 12] or systems like PVM [26] attempt to address. In our system, all nodes run the same, slightly modified version of the Linux 2.2.5 operating system under the same configuration (identical RPMs installed, same mount points for NFS filesystems, etc.).

Common user ID/account database. We assume each user has a unique user ID and an account which is the same on all the nodes in the cluster. This is implemented by having all nodes in the cluster use the same NIS domain. Assuming unique user IDs and accounts on all cluster nodes is a reasonable assumption for a system on the scale of a machine room, building, or campus. Again, we do not attempt to address the heterogeneity and cross administrative domain issues of wide area meta-computing systems. Requiring a user account on all the cluster nodes is not terribly restrictive and can be implemented fairly easily on the class of systems we address. Furthermore, it does not necessarily preclude a user without an account from accessing resources on the cluster. Access to the cluster could, for example, be made available through well-defined services for popular applications whose server-side implementations run using a real user account. Alternatively, with proper security, guest accounts could even be offered.

3 System Architecture

To address the issues in Section 2, we have built REXEC, a decentralized, secure remote execution environment for clusters of computers. The system consists of three main programs: *rexecd*, a daemon which runs on each cluster node; *rexec*, a client program that users run to execute jobs using REXEC; and *vexecd*, a replicated daemon which provides node discovery and selection services (Figure 1). Users run jobs using the rexec client. The client performs two functions: (i) selection of nodes based on user preferences (e.g., lowest CPU load) and (ii) remote execution of the user's application on those nodes through direct SSL/TCP connections to node rexecd daemons. The system is currently installed and running on a 32-node cluster of 2-way Dell Poweredge 2300 SMPs running a modified version of the Linux 2.2.5 operating system as part of the Berkeley Millennium Project. In this section, we provide details on the key features of REXEC and show how these features address our design goals.

3.1 Decentralized Control

REXEC uses decentralized control to allow for graceful scaling of system overhead as more cluster nodes are added and the number of applications being run increases. Upon selecting a set of remote nodes to run on, the rexec client opens TCP connections to each of the nodes and executes the remote execution protocol with rexecd daemons directly. These direct client to daemon connections allow the work (e.g., forwarding to stdin, stdout, and stderr, networking and process resources, etc.) of managing the remote execution to be distributed between the rexec client and the rexecd daemons. With a large number of nodes, having a centralized entity act as intermediary between users and cluster nodes can easily become a bottleneck as single node resources become an issue (e.g., kernel limitations on # of sockets). Our scheme avoids this problem by distributing this work.

In addition to scalability, a decentralized design by definition avoids single points of failure. By freeing users from depending on intermediate entities to access the nodes they need to run their programs, we ensure that any functional node in the system which is reachable over the network and running an rexecd daemon can always be used to run user

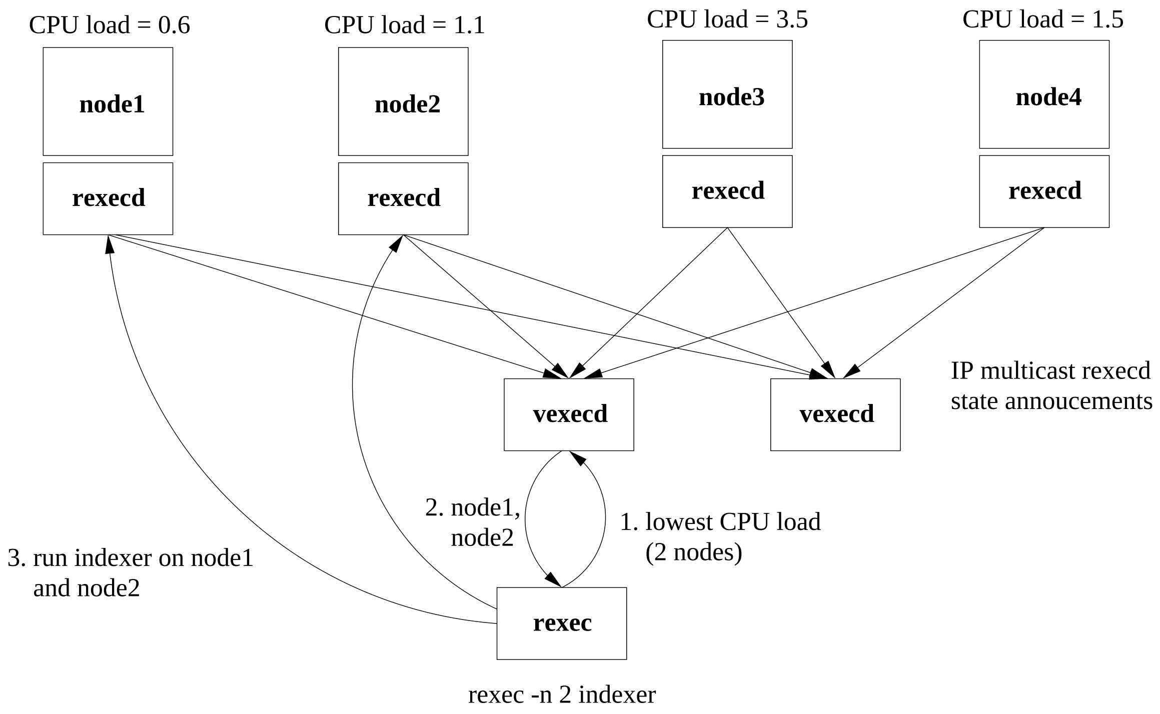

Figure 1: **Overall organization of the REXEC remote execution environment.** The system consists of three main programs: *rexecd*, a daemon which runs on each cluster node; *rexec*, a client program that users run to execute jobs using REXEC; and *vexecd*, a replicated daemon which provides node discovery and selection services. Users run jobs in REXEC using the rexec client which performs two functions: (i) selection of which nodes to run on based on user preferences and (ii) remote execution of the user's application on those nodes through direct SSL/TCP connections to the node rexecd daemons. In this example, there are four nodes in the system: node1, node2, node3, and node4 and two instances of vexecd, each of which implements a lowest CPU load policy. A user wishes to run a program called indexer on the two nodes with the lowest CPU load. Contacting a vexecd daemon, rexec obtains the names of the two machines with the lowest CPU load, node1 and node2. rexec then establishes SSL/TCP connections directly to those nodes to run indexer on them.

applications. REXEC can have any number of "front end" machines. This is in contrast to previous systems such as GLUnix [11] and SCore-D [13], which use a centralized entity as the intermediary between clients and the cluster. In GLUnix, when the master crashes, all 115 nodes of the Berkeley NOW cluster become unavailable for running programs through the GLUnix system. In practice, centralized entities with no backup or failover capabilities can decrease system availability significantly.

3.2 Transparent Remote Execution

REXEC provides transparent remote execution which allows processes running on remote nodes to execute and be controlled as if they were running locally. REXEC uses four mechanisms to accomplish this: (i) propagation and recreation of the user's local environment on remote nodes, (ii), forwarding of local signals to remote processes, (iii) forwarding of stdin, stdout, and stderr between the rexec client and remote processes and (iv) local job control to control remote processes.

To propagate and recreate the user's local environment on each remote node, the rexec client packages up the user's local environment including the user's current working directly, environment variables, user ID, group ID, program to execute, and its command line arguments. Then, as part of the remote execution protocol, it sends this information to each rexecd on each node that has been selected to run the user's application. Upon receiving a new connection request from an rexec client, each rexecd spawns a new client thread to manage that particular rexec instance (Figure 2). Before forking and execing the user's program, this thread, among other things, unpackages the user's local environment and recreates it using standard C library functions (e.g., chdir, setenv, setuid). Each remotely forked process will then execute with the same local environment as the machine where the rexec client is running.

Second, REXEC forwards locally delivered signals to each remote process. REXEC does not implement a true global PID space. Instead, to send a signal to all remote processes in an application, users send the signal to the local rexec client. The rexec client acts as proxy to all remote processes. Upon receiving a signal, the rexec client forwards that signal to an rexecd thread on each node the application is running on. That thread is responsible solely for forwarding signals and stdin received from an rexec client instance to its corresponding remote process. REXEC currently forwards most of the 30 or so Unix signals supported in Linux. SIGUSR1 and SIGUSR2 are used by the pthreads library and so those cannot be caught without breaking it. SIGINT and SIGTSTP, which are related to job control, are handled differently as discussed later. Finally, because REXEC is a user-level implementation, SIGKILL (if the rexec clients crashes, the same effect is achieved through REXEC's error detection and cleanup mechanisms as discussed later) and SIGSTOP on the rexec client cannot be caught.

Third, REXEC forwards stdin from the rexec client to remote processes and forwards stdout and stderr from remote processes to the rexec client. For stdin, there is an additional thread in rexec that is responsible solely for forwarding stdin to remote processes. (stdin is broadcast to all remote processes.) On the remote side, the signals and stdin forwarding thread receives stdin messages and forwards them to remote processes through a Unix pipe. For stdout, rexecd has per-rexec client stdout and stderr threads which use Unix pipes to read output from the remote process and forward this data back to the rexec client [2]. For each remote node, rexec has a thread which, among other things, reads remote stdout and stderr and prints output locally to the user's terminal. Each line of stdout and stderr is prefixed by the node number of the remote process that printed it (e.g., node 0 that prints "hello world" appears as "(0) hello world").

Fourth, REXEC allow users to use standard job control to control remote processes in their applications. Just as we use the rexec client as a proxy to deliver signals, so we use it as a proxy which can be used to control remote processes through job control. More specifically, REXEC supports C-c, C-z, fg, and bg commands from the shell to deliver SIGINT, SIGTSTP, and SIGCONT signals. Because REXEC is implemented mainly at user-level, job control and the delivery of the appropriate signals sometimes requires approximating the desired effect. For example, typing C-z to a local process delivers SIGTSTP to the process which suspends execution of that process. Delivering SIGTSTP to a remote process, however, is not meaningful since that process is not attached to a terminal. Thus, for SIGTSTP (C-z) and SIGINT (C-c), rexec catches these signals and delivers SIGSTOP and SIGKILL respectively.

3.3 Dynamic Cluster Membership

REXEC uses a dynamic cluster membership service that avoids use of static configuration files. In our experience with large clusters of computers, we have found that over time the set of nodes in the cluster that are available to perform useful work varies for a number of reasons. Adding nodes, removing nodes, bringing machines down, rebooting machines, and so forth are fairly common events in our system. Thus, rather than use static configuration files to define

[2] Note that due to the way stdout is buffered, applications that run on REXEC must use fflush statements if they want stdout to be written immediately to the stdout pipe so stdout is displayed immediately on the user's terminal.

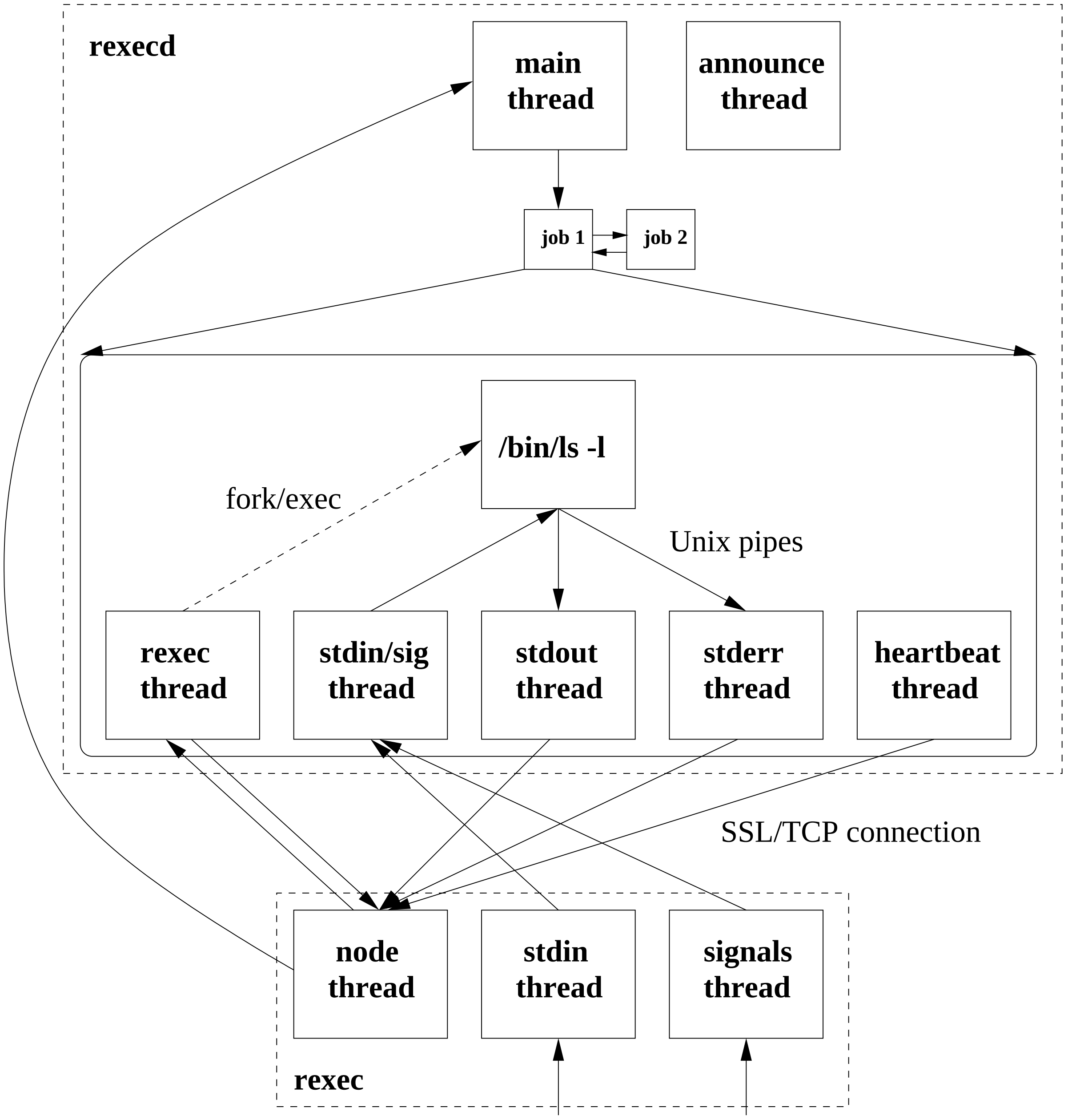

Figure 2: **Internal thread structure and data flows for rexec and rexecd.** rexec consists of a stdin thread for forwarding of stdin, a signals thread for forwarding of signals, and one node thread per node for managing remote process execution including printing remotely forwarded stdout and stderr to the user's local terminal, receiving heartbeat packets (and rexec client monitoring of the SSL/TCP connection), and receiving the exit status of the remote process. (In this example, rexec is running '/bin/ls -l' on a single node so there is only one node thread.) rexecd consists of a main thread which creates new threads for new rexec clients and maintains a list of running jobs, an announce thread for sending multicast state announcements, and a collection of per-rexec-client threads. These per-rexec-client threads include a main rexec thread for the client, the user process forked and execed by the main rexec thread, a stdin/signals thread for forwarding stdin/signals from rexec to the user process, stdout and stderr threads for forwarding stdout and stderr from the user process to rexec, and a heartbeat thread for sending of periodic heartbeat packets to detect failures in the SSL/TCP connection between rexec and rexecd.

the set of nodes in the cluster we use a dynamic cluster membership service that discovers nodes as they join and leave the cluster.

This cluster membership service is built around a well-known cluster multicast address which is used to communicate various information. By using a well-known multicast address, processes on different nodes can communicate with interested receivers without explicitly naming them. Senders who wish to send some information simply send it on the multicast address with a unique message type. Interested parties can elect to receive and interpret information of interest by examining incoming message types. If necessary, processes can even use the multicast channel to bootstrap point-to-point connections (e.g., by sending a multicast request for a server's point-to-point address). The Information Bus [19], an architecture for publish/subscribe services, uses a similar but more general model.

To implement the cluster membership service, we first needed a way for nodes to be added to the cluster membership when they are available and to be deleted from the cluster membership when they are unavailable. To do this, the rexecd daemons multicast announcement packets which include the IP address of the node (a TCP port could be included as well but currently rexecd daemons listen on a well-known port) along with other configuration information (e.g., number of CPUs) and state information (e.g., number of jobs REXEC is controlling, load, etc.). These announcement packets are sent periodically (once every minute) and also whenever a significant change in state observed by rexecd occurs. Currently, the state changes we send announcements for are job arrivals and job departures.

We maintain approximate membership of the cluster by using the reception of an announcement packet as a sign that a node is available and the non-reception of an announcement over a small multiple of the periodic announcement interval as a sign that a node is unavailable. The membership is approximate because information is necessarily delayed and because we are using unreliable UDP/IP multicast as our transport protocol. Using the announcement packets, we use replicated vexecd daemons to discover and maintain the node membership in the cluster by caching and timing out node announcement information. Unlike V [25, 27], which used a similar multicast-based protocol for transmitted load information, we use separate daemons to maintain the membership and load information as opposed to having every node in the cluster do so. Our primary reason for doing this is to avoid context switches on cluster nodes that are running user applications. If each rexecd received all multicast packets, then rexecd would need be scheduled to handle the receiving of these packets.

3.4 Decoupled Discovery and Selection

REXEC decouples node discovery from the selection of which nodes an application should run on. As discussed in the previous section, replicated vexecd daemons are responsible for discovering and maintaining the node membership of the cluster. Within one state announcement period, a new vexecd discovers the entire instantaneous membership of the cluster. With multiple vexecd daemons keeping track of all the nodes, their configuration, and state, a selection policy is simply a mapping that applies some criteria to that list of available hosts and returns a set of hosts.

Because users may have different criteria in how they want nodes to be selected for their applications, discovery and selection are decoupled. The vexecd daemons which do discovery can implement any number of selection policies. The idea with vexecds implementing selection services as well is that we envision that most users will probably choose from a small set of policies in deciding where to run their applications. In a community composed largely of scientific computing users, for example, lowest CPU load may be the most common criteria.

vexecds in our system precompute and cache orderings on the list of available nodes so clients can quickly obtain the results of common selection policies. Under most circumstances, users will contact prewritten vexecd daemons asking for the n "best" nodes, where best is defined according to some selection criteria. The vexecds simply return the top n nodes on their ordered list, which is recomputed each time a state change occurs with adjustments. vexecds are free to implement policies with more complexity as well (e.g., return n nodes with lowest average CPU load over last three days). Users are also free to implement their own selection policies (e.g., by writing their own vexecd). As anyone is able to receive CPU state announcement messages from rexecds, such an implementation would simply need to listen on the cluster IP multicast address, receive announcement messages, and apply a policy.

Since users will want to use vexecds based on the selection policies the vexecds implement, the discovery of vexecd daemons and use of their services cannot be completely transparent. More information is needed from the user either in the form of a list of suitable vexecd servers or a criteria which is expressed in a way that the system can automatically discover which vexecd daemons implement that criteria. Currently, we take the former approach. Users specify a list of suitable vexecd daemons using an environment variable (VEXEC_SVRS, a space-delimited list of hostnames where vexecd daemons run). The rexec client, upon seeing a non-null VEXEC_SVRS, will try each daemon in turn until it succeeds or exhausts the list.

Discovery of the hostnames of the machines that implement suitable vexecd daemons can be done through out-ofband means (e.g., a system administrator uses a web page to post a list of hostnames that are designated to run vexecds with certain policies) or it can be done semi-automatically. We offer both approaches. The former is self-explanatory. The latter involves using the cluster IP multicast channel to multicast to all vexecd daemons in the system asking them what selection policy they implement. Each vexecd, upon receiving a such a request, returns a string that provides a textual description of its policy which the user can then use to construct a suitable list for setting the VEXEC_SVRS environment variable.

3.5 Error and Cleanup Model

REXEC provides a well-defined error handling and cleanup model for applications. If an error occurs on the rexec client, in any of the remote processes, or on any of the TCP connections between the rexec client and any of the remote rexecd daemons, the entire application exits, all resources are reclaimed, and a textual error message is printed to the user. A common shortcoming in many previous remote execution systems, especially those that support parallel execution, is lack of a precise error and cleanup model and insufficient implementations of remote cleanup. Just as point-to-point facilities such as ssh often fail to perform appropriate cleanup (e.g., local C-c on the client does not always kill the remote process), so many user-level remote execution environments also fail to properly relinquish resources on remote nodes. REXEC addresses this by defining a model, addressing the new failure modes associated with remote execution and parallel and distributed programs, and providing a robust implementation.

Transparent remote execution on a cluster and support for parallel and distributed applications introduces new failure modes in the execution of user applications. With remote process control being mediated through use of a local proxy, the rexec client, there are two obvious new modes of failure. First, the rexec client can fail and second the network connection between the rexec client and a remote rexecd can fail. Because the rexec client serves as the point of control for the user and logically represents the remote process(es) itself, we interpret failure of the rexec client as failure of the application and thus we cleanup. When the network connection from rexec to the rexecd fails, this prevents the user from controlling the remote process and thus we view this as a failure as well.

Support for parallel and distributed programs introduces yet another failure mode: individual failure of processes in the program. Here, we take the position that if any process in the program fails, the entire application is aborted. The rationale behind this is that for all but the most trivially parallelizable and distributed programs, there will be communication between processes and thus failure of one process usually means failure of the entire application. Of course, this will not be the case for every parallel or distributed program but we feel it is a reasonable model to present to users for a large class of programs.

In general, there are many potential error and cleanup models the system could support. However, only a handful of them make practical sense to real applications. For example, another useful failure model which we are considering supporting but currently do not implement is the model where all processes are completely decoupled and we leave it up to the application to deal with failures. Such a model might be appropriate, for example, for a parallel application with its own error detection and the ability to grow and shrink based on resource availability and faults.

The implementation of the error and cleanup model is done mainly at user-level but also involves some small kernel modifications. At user-level, we need to monitor remote process failures locally on remote nodes and we need to monitor network failures. To determine whether a process failed, we have a thread in rexecd per remote process on that rexecd which captures the return code returned from the process. Examining the return code, this thread then performs a check to see if the process exited normally or abnormally. If it exited with an abnormal exit code, we interpret this as an error.

To determine whether a network connection has failed or not, we rely on system call error codes returned from network related system calls. In cases where processes fail but the machine's operating system is still functioning properly, network errors are detected quickly due to the OS's implementation of the TCP protocol. However, to ensure timely detection of network failures in cases where data is not sent for long periods of time (e.g., compute bound jobs) and a machine crashes, we use a heartbeat thread on the remote node (Figure 2)). Since this heartbeat sends periodic heartbeats from rexecd to rexec, if the network connection has failed, these send failures will be detected in a timely manner [3]. Note that by interpreting network failures as failures which cause cleanup, we also cover the case where the rexec client is sent a SIGKILL signal which it cannot catch due to our user-level implementation. In this case, rexec exits, its TCP connections are closed, and this causes remote rexecd daemons to perform cleanup.

REXEC uses kernel modifications to ensure that remote resources are reclaimed when performing remote cleanup. Specifically, we wanted to address cleanup issues for remote processes, including those that fork arbitrary process trees

[3]TCP keepalive packets were not an option due to excessively high timeout values on the order of hours and the inability to set timeouts on a per-connection basis.

and have multiple threads. In these cases, keeping track of the process ID of the root of the process tree and sending a SIGKILL signal to that process is not enough to perform cleanup. Other descendents of that process may still be computing. Thus, we added a new system call to specify the root of a process tree (and that all descendents of that process should inherit the fact that they are part of the same tree) and a system call to deliver signals to all members of that tree, regardless of changes in process group, intermediate parents exiting causing their children to be inherited by init, and so forth. When performing cleanup, REXEC simply sends a SIGKILL to all processes in the logical tree, which results in all resources for all threads/processes in the process tree to be freed up. (Note also that we modified the wait system call to deal with process trees so a remote thread waiting for the remote process to exit does not return until all processes in the logical tree have exited.)

3.6 Parallel and Distributed Applications

REXEC supports parallel and distributed applications by allowing users to launch and control multiple instances of the same program on multiple nodes and by providing a set of hooks that allow parallel runtime environments to be built. Starting #nodes instances of the same program is accomplished by adding a -n #nodes switch to the rexec client program which allows the user to specify a program should be run on #nodes nodes of the cluster.

The hooks we provide for runtime environments are a fairly minimal set. Each remote process has four environment variables set by REXEC: REXEC_GPID, REXEC_PAR_DEGREE, REXEC_MY_VNN, and REXEC_SVRS. REXEC_GPID is a globally unique identifier for a particular execution of a user's application. It is implemented as a 64-bit concatenation of the 32-bit IP address of the interface the rexec client uses to communicate with rexecds and the local 32-bit process ID of the rexec client. REXEC_PAR_DEGREE is a 32-bit integer which specifies the number of nodes the application is running on. Within an n node program, REXEC assigns an ordering on the nodes from 0 to REXEC_PAR_DEGREE - 1. On each node, REXEC_MY_VNN specifies the position of that node in that ordering. Finally, REXEC_SVRS contains a list of REXEC_PAR_DEGREE hostnames (or IP addresses) for each of the nodes the user's application is running on.

Using these four environment variables, a parallel runtime environment, for example, would have enough information to identify which nodes should work on various parts of a dataset and to use whatever means employed to discover network addresses of peer endpoints. An example of how REXEC's parallel and distributed application support is used in practice is given in Section 4.2, where we discuss an REXEC-based MPI runtime.

3.7 Authentication and Encryption

REXEC provides user authentication and encryption of all communication between rexec clients and rexecd daemons. More specifically, REXEC uses the SSLeay version 0.9.0b implementation of the Secure Socket Layer (SSL) protocol [10] for authentication and encryption of all TCP connections between these entities. Each user has a private key, encrypted with 3DES, and a certificate containing the user's identity and a public key that is signed by a well-known certificate authority who verifies user identities. In our system, we use a single trusted certificate authority for certificate signing and use user names as identifies in certificates.

Each time a user wants to run an application using REXEC, the user invokes the rexec client on the command line and types in a passphrase which decrypts the user's private key. The system then performs a handshake between the rexec client and rexecd, negotiates a cipher, uses a Diffie-Hellman key exchange to establish a session key, uses RSA to verify that the user's certificate was signed by the trusted certificate authority, and checks that the username in the certificate exists and that it matches that of corresponding local user ID that was propagated from the rexec client. Once the user's identity has been established, all communication over the corresponding TCP connection is encrypted with 3DES using the shared session key.

4 REXEC Applications

In this section, we present three examples of how REXEC has been applied to provide remote execution facilities to applications. In the first example, we describe how the REXEC system is used in its basic form to provide remote execution for parallel and sequential jobs. In the second example, we describe an MPI implementation using a fast communication layer on Myrinet that uses REXEC as its underlying remote execution facility. Finally, in the third example, we provide an example of how REXEC has been extended to provide remote execution on Berkeley's Millennium cluster which uses market-based resource management techniques [6].

4.1 Parallel and Sequential Jobs

The rexec client provides the minimal amount of support needed to transparently run and control parallel and sequential programs on a cluster. Users run the rexec client as follows: rexec -n #nodes progname arg1 arg2 .. argn, where #nodes is the number of nodes the program should be executed on and progname arg1 arg2 .. argn is the command line the user would type to run program progname with arguments arg1, arg2, .., argn on a single node. Node selection is done through use of vexecd daemons by specifying a list of suitable vexecd daemons through the VEXEC_SVRS environment variable. Alternatively, if the user wants to run an application on a specific set of nodes, the user can set the REXEC_SVRS environment variable. A non-null REXEC_SVRS always takes precedence over VEXEC_SVRS. Parallel and distributed programs can be launched using the basic rexec client. It is responsibility of runtime layers or the application to make use of REXEC's environment variable support for parallel and distributed programs to decide how data and computation should be partitioned and how communication between processes is established.

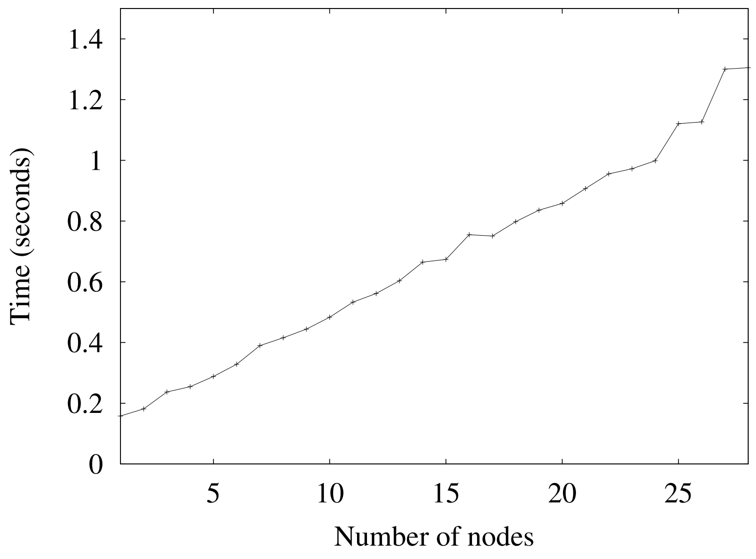

Figure 3: **Measured execution time to run a null parallel program with REXEC as a function of number of nodes.** This graph shows the measured execution time of a parallel program that starts and immediately exits on multiple nodes of the cluster. The measurements illustrate the basic costs associated with running jobs through the REXEC system. The start-up and cleanup cost for a running a single node program with REXEC is 158 ms. As the number of nodes increases, total execution time scales linearly with an average per-node cost of 42.8 ms. (Note that we have not aggressively pursued optimizations so absolute performance numbers can still be improved.)

4.2 MPI/GM on Myrinet

Using REXEC's basic hooks for parallel and distributed programs, we have modified Myricom's MPI implementation over the GM (MPI/GM) fast communication layer [17] to use REXEC as its underlying remote execution mechanism. The original version of MPI/GM was configured by default to use rsh with static node/port number configuration files for remote execution to each node. With our modifications, users are able to run MPI programs on the cluster with GM while obtaining all the benefits of the REXEC system. Running an MPI program on multiple nodes is no different from running a normal application using REXEC. Two practical examples of problems we solve are elimination of static host configuration files and elimination of GM ports conflicts due to straggler processes, a problem which was immediately evident once we tried to use C-c to kill an MPI/GM program.

Modifying the MPI ADI layer to use REXEC required mapping REXEC environment variables to MPI variables and addressing port (communication endpoint) naming issues. MPI, like other parallel runtime environments, provides a set of library functions that allow each process in a program to obtain the total number of processes in the program and that process's unique ID according to some naming of those processes. These variables are most often used to decide how data or computation should be partitioned amongst multiple processes in a parallel program. For MPI/GM, these variables are the MPI size and rank and they are easily set for each process by simply setting them to the values of the REXEC_PAR_DEGREE and REXEC_MY_VNN environment variables.

In MPI/GM, each process uses a single port to communicate with all other processes in the program. Each node, or more specifically each Myrinet LANai4 network interface card, can support up to seven user ports ranging from 0 to 6. A port address is a combination of a hostname and an integer port number. The default configuration for MPI/GM relies on each user having a configuration file which specifies which port number is used by each process in the user's MPI program. With multiple programs running on the cluster, though, this manual configuration forces the user to deal with port name conflicts due to other programs competing for and using the same GM resources.

We solved the naming problem by first extending the GM library to support a function that binds to any free GM port and returns that port number to the application. This allows us to avoid having to explicitly specify port numbers. However, doing this now meant that each process in unaware of which port numbers its peer processes are bound to. Thus, the next thing we did was to allow peer processes to exchange names through bind and lookup operations on enameserver, a centralized nameserver used in the Berkeley NOW project. To do this, we took advantage of the two remaining environment variables, REXEC_GPID and REXEC_SVRS. The nameserver allows processes to bind (key,value) pairs and do to value lookups based on keys. We use the concatenation of REXEC_GPID and a process's rank separated by a ':' for keys. Because REXEC_GPID is unique throughout the system, this ensures that a n node program uses n unique keys. For values, we use the GM port numbers returned from the port creation function we added. Upon binding a (key,value) pair for its port, each process does a lookup on port numbers for each of its peers. The results of these lookups are then saved and used later for communication.

4.3 Computational Economy

As part of the Berkeley Millennium Project, we have extended the REXEC remote execution environment to operate in the context of a market-based resource management system. In this system, users compete for shared cluster resources in a *computational economy* where nodes are sellers of computational resources, user applications are buyers, and each user sets a willingness to pay for each application based on the personal utility of running it. By managing resources according to personal value, two key benefits are obtained. First, by allocating resources based on willingness to pay, the cluster can optimize for overall user satisfaction as opposed to system-centric global performance metrics (e.g., mean job completion time) which ignore personal user valuations of the resources. Second, by using a pricing system to mediate access to the resources, the system can provide the proper incentives for users to use the system in a socially responsible way. We hypothesize that market-based sharing can deliver significantly more value to users than traditional approaches to resource management.

To support a computational economy, we extended the REXEC system in three ways. First, we added a new command line switch (-r maxrate) to the rexec client to specify the maximum rate, expressed in credits per minute, the application is willing to pay for CPU time. Using this switch, users express the personal of CPU time to their applications as a maximum rate their application is willing to pay for CPU time. Second, rexecd was modified to to use an economic front end (a collection of functions that implement the CPU market) which performs proportional-share CPU allocation using a stride scheduler [28] and charging of user accounts for CPU usage. This ensures that applications run through REXEC compete for CPU time through the computational economy. Third, we modified rexecd to include in its announcement packets the current aggregate willingness to pay of all REXEC applications competing for its resources. By augmenting the announcement packets with this information, this allows vexecd daemons to implement policies that involve CPU price (e.g., find the n nodes with the lowest price).

5 Related Work

Research efforts in remote execution environments for clusters have been going on for over a decade. Each has succeeded in addressing and to various extents solving different subsets of the key problems in remote execution systems. None of these systems, however, has addressed the range of problems that REXEC does. Built on previous work and practical experience with a large-scale research cluster, REXEC addresses a wide range of practical needs while providing useful features which address important issues such as error handling and cleanup, high availability, and dynamic cluster configuration. To accomplish its goals, REXEC is implemented at user-level on a commodity operating system with small modifications to the OS kernel. Such an approach is an example of one of three distinct implementation strategies: (i) user-level approaches (ii) modification of existing operating systems, and (iii) completely new distributed operating systems.

GLUnix [11], SCore-D [13], Sidle [14], Butler [18], HetNOS [4], and Load Sharing Facility (LSF) [30] are examples of user-level implementations. Compared to REXEC, each of these systems implements a subset of REXEC's features. GLUnix and Score-D, for example, are the only two systems in the list that support parallel programs. However, both of them also rely on centralized control and manually updated cluster configuration. Butler and LSF support different forms of replicated discovery and selection. However, neither supports an error and cleanup model as extensive as REXEC or strong authentication and encryption. One notable feature that has been implemented in some of these systems which REXEC currently does not support is a programmatic interface to the system. GLUnix, Butler, HetNOS, and LSF, for example, allow users to write applications which link with a C library of remote execution related functions. Using this model, applications such as a shell which automatically decides whether to execute a job locally or remotely have been developed.

MOSIX [3], NEST [1], COCANET Unix [23], and Solaris MC [15, 24] are examples of modifying and extending an existing operating system. Again, each of the systems supports only a subset of REXEC's features. NEST, for example, supports transparent remote execution but does not support features such as dynamic cluster membership or parallel and distributed application support. MOSIX, for example, provides transparent remote execution but does so in a fairly limited context which is mainly targeted for load balancing amongst a set of desktop machines to exploit idle time. One notable feature supported by MOSIX which REXEC currently does not support is process migration. Mechanisms to implement it, however, are well-known [8, 16, 21, 27] in both user-level and kernel-level implementations and under various constraints. Another notable difference between REXEC and these kernel-level implementations is the degree of transparency in the remote execution system. Kernel-level implementations can achieve greater levels of transparency than user-level approaches. Solaris MC, for example, implements a true single system image with real global PIDs, a global /proc filesystem, and a global cluster-wide filesystem.

Sprite [20], V [25, 27], and LOCUS [29] are examples of completely new distributed operating systems which support transparent remote execution. Like the other systems described, these distributed operating systems also support only a subset of REXEC's features. V, for example, supports a publish-based, decentralized state announcement scheme very much like REXEC. On the hand, V does not support parallel applications, does not support flexible selection policies, nor does it implement strong authentication and encryption. Like MOSIX, all three of these systems support process migration which REXEC currently does not implement. In addition, like the kernel-level implementations previously described, these new operating systems also achieve greater levels of transparency due to implementations at the operating system level and, in the case of Sprite and LOCUS, cluster-wide global filesystems.

6 Conclusion

To bring clusters of computers into the mainstream as general-purpose computing systems, better facilities are needed for transparent remote execution of parallel and sequential applications. While much research has been done in the area of remote execution, much of this work remains inaccessible for clusters built using contemporary hardware and operating systems. To address this, we designed and implemented a new remote execution environment called REXEC. Building on previous work in remote execution and practical experience with the Berkeley NOW and Millennium clusters, it provides decentralized control, transparent remote execution, dynamic cluster configuration, decoupled node discovery and selection, a well-defined failure and cleanup model, parallel and distributed program support, and strong authentication and encryption. The system is implemented and is currently installed on a 32-node cluster of 2-way SMPs running the Linux 2.2.5 operating system. It currently serves as the remote execution facility for market-based resource management studies as part of the UC Berkeley Millennium Project.

References

[1] AGRAWAL, R., AND EZZAT, A. K. Location independent remote execution in nest. *IEEE Transactions on Software Engineering 13*, 8 (August 1987), 905–912.

[2] ANDERSON, T. E., CULLER, D. E., PATTERSON, D. A., AND THE NOW TEAM. A case for now (networks of workstations). *IEEE Micro* (Feb. 1995).

[3] BARAK, A., LA'ADAN, O., AND SMITH, A. Scalable cluster computing with mosix for linux. In *Proceedings of Linux Expo '99* (May 1999), pp. 95–100.

[4] BARCELLOS, A. M. P., SCHRAMM, J. F. L., FILHO, V. R. B., AND GEYER, C. F. R. The hetnos network operating system: a tool for writing distributed applications. *Operating Systems Review* (October 1994).

[5] Chien, A., Pakin, S., Lauria, M., Buchanon, M., Hane, K., and Giannini, L. High performance virtual machines (hpvm): Clusters with supercomputing apis and performance. In *Proceedings of 8th SIAM Conference on Parallel Processing for Scientific Computing (PP97)* (1997).

[6] Chun, B. N., and Culler, D. E. Market-based proportional resource sharing for clusters. Submitted for publication, September 1999.

[7] Culler, D., Arpaci-Dusseau, A., Arpaci-Dusseau, R., Chun, B., Lumetta, S., Mainwaring, A., Martin, R., Yoshikawa, C., and Wong, F. Parallel computing on the berkeley now. In *Proceedings of of 9th Joint Symposium on Parallel Processing* (Kobe, Japan, 1997).

[8] Douglis, F., and Ousterhout, J. Transparent process migration: Design alternatives and the sprite implementation. *Software—Practice and Experience 21*, 8 (August 1991).

[9] Foster, I., and Kesselman, C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications 11*, 2 (1997), 115–128.

[10] Freier, A. O., Karlton, P., and Kocher, P. C. The ssl protocol version 3.0 (internet-draft). 1996.

[11] Ghormley, D. P., Petrou, D., Rodrigues, S. H., Vahdat, A. M., and Anderson, T. E. Glunix: a global layer unix for a network of workstations. *Software—Practice and Experience* (Apr. 1998).

[12] Grimshaw, A., Ferrari, A., Knabe, F., and Humphrey, M. Legion: An operating system for wide-area computing. Tech. Rep. CS-99-12, University of Virginia, Department of Computer Science, 1999.

[13] Hori, A., Tezuka, H., , and Ishikawa, Y. An implementation of parallel operating system for clustered commodity computers. In *Proceedings of Cluster Computing Conference '97* (March 1997).

[14] Ju, J., Xu, G., and Tao, J. Parallel computing using idle workstations. *Operating Systems Review* (July 1993).

[15] Khalidi, Y. A., Bernabeu, J. M., Matena, V., Shirriff, K., and Thadani, M. Solaris mc: A multi computer os. In *Proceedings of the 1996 USENIX Conference* (1996).

[16] Litzkow, M., Tannenbaum, T., Basney, J., and Livny, M. Checkpoint and migration of unix processes in the condor distributed processing system. Tech. Rep. 1346, University of Wisconsin-Madison, April 1997.

[17] Myricom. The gm api. 1999.

[18] Nichols, D. A. Using idle workstations in a shared computing environment. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (1987).

[19] Oki, B., Pfluegl, M., Siegel, A., and Skeen, D. The information bus: An architecture for extensible distributed systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (1993), pp. 58–68.

[20] Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B. The sprite network operating system. *IEEE Computer 21*, 2 (February 1988).

[21] Plank, J. S., Beck, M., Kingsley, G., and Li, K. Libckpt: Transparent checkpointing under unix. In *Proceedings of the 1995 USENIX Winter Conference* (1995).

[22] Ridge, D., Becker, D., Merkey, P., , and Sterling, T. Beowulf: Harnessing the power of parallelism in a pile-of-pcs. In *Proceedings of IEEE Aerospace* (1997).

[23] Rowe, L. A., and Birman, K. P. A local network based on the unix operating system. *IEEE Transactions on Software Engineering 8*, 2 (March 1982).

[24] Shirriff, K. Building distributed process management on an object-oriented framework. In *Proceedings of the 1997 USENIX Conference* (1997).

[25] Stumm, M. The design and implementation of a decentralized scheduling facility for a workstation cluster. In *Proceedings of the 2nd IEEE Conference on Computer Workstations* (March 1988), pp. 12–22.

[26] SUNDERAM, V. S. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience 2*, 4 (December 1990), 315–339.

[27] THEIMER, M. M., LANTZ, K. A., , AND CHERITON, D. R. Preemptable remote execution facilities for the v-system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (1985).

[28] WALDSPURGER, C. A., AND WEIHL, W. E. Stride scheduling: Deterministic proportional-share resource management. Tech. Rep. MIT/LCS/TM-528, Massachusetts Institute of Technology, 1995.

[29] WALKER, B., POPEK, G., ENGLISH, R., KLINE, C., AND THIEL, G. The locus distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles* (1983), pp. 49–70.

[30] ZHOU, S., WANG, J., ZHENG, X., AND DELISLE, P. Utopia: A load sharing facility for large, heterogenous distributed computer systems. *Software—Practice and Experience* (1992).