REXEC: A Decentralized, Secure Remote Execution Environment for Clusters

Brent N. Chun and David E. Culler

University of California at Berkeley
Computer Science Division
Berkeley, CA 94720
Tel: +1 510 642 8299
Fax: +1 510 642 5775
{bnc,culler}@cs.berkeley.edu
http://www.cs.berkeley.edu/~{bnc,culler}
Keywords: Clusters, Remote execution, Distributed systems, Decentralized control

January 8, 2000

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

We have designed and implemented a new remote execution environment called REXEC [1] to address the lack of a sufficient remote execution facility for parallel and sequential jobs on clusters of computers. Building on previous work in remote execution and practical experience with the Berkeley NOW and Millennium clusters, the system provides 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. It is implemented almost entirely at user-level with small modifications to the Linux 2.2.5 kernel. The system 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. Section 6 describes future work and in Section 7 we conclude the paper.

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 goals are as follows:

- *High availability*. The system should be highly available and provide graceful degradation of service in the presence of failures.
- *Scalability*. As more nodes are added and more applications are run, remote execution overhead should scale gracefully.
- *Transparent remote execution*. Execution on remote n-

[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 [13] system or the rexec function in NEST [1].

odes should be as transparent as possible.

- *Minimal use of static configuration files*. The remote execution system should rely on as few static configuration files as possible.
- *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.
- *Well-defined failure and cleanup models*. The system should provide well-defined models for failure and cleanup.
- *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.
- *Security*. The system should provide user authentication and encryption of all communication.

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. Our assumptions are as follows:

- *Uniform file pathnames*. We assume that all shared files are accessible on all nodes using the same pathnames and that most local files on each node are also accessible under the same pathnames (e.g., /bin/ls).
- *Compatible OS and software configurations*. We assume all nodes in the cluster run compatible versions of the operating system and have compatible software configurations.
- *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.

3 System Architecture

The REXEC system architecture is organized around three types of entities: *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 on the system using the rexec client. The rexec 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-encrypted TCP connections to node rexecd daemons. REXEC is implemented and 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. 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 for graceful scaling of system overhead as more cluster nodes are added and more applications are being run. 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 the 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 an intermediary between users and cluster nodes can easily become a bottleneck as single node resources become an issue. 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 applications. REXEC can have any number of "front end" machines. This is in contrast to previous systems such as GLUnix [7] and SCore-D [8], which use a centralized entity as the intermediary between clients and the cluster. In GLUnix, for example, 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. It 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.

The implementation of these mechanisms is centered around a collection of rexec client and per-rexec-client rexecd threads. Referring to Figure 2, propagation and recreation of the user's local environment is done by having the node thread in rexec package up the user's local environment and having the rexec thread in rexecd recreate it after forking and before execing the user's job. Forwarding of local signals and stdin is done by having the signals thread and stdin thread in rexec forward signals and stdin to each of the remote stdin/sig threads in the rexecds, which then deliver them to the user's application using signals and Unix pipes. Forwarding of remote stdout and stderr is done by having stdout and stderr threads in rexecd read from stdout and stderr Unix

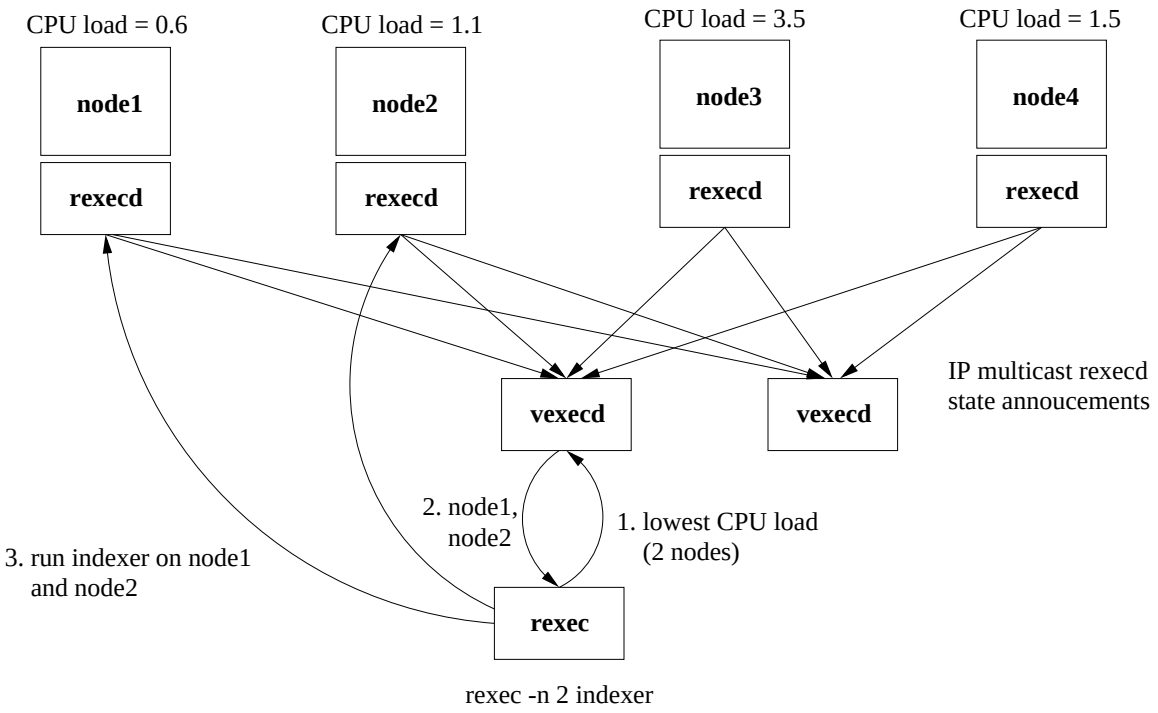

Figure 1: **Overall organization of the REXEC remote execution environment.** The system is organized around three types of entities: *rexecd*, a daemon which runs on each cluster node; *rexec*, a client program that users run to execute jobs using REXEC; and *vexecd*, a replicated daemon which provides 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-encrypted 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-encrypted TCP connections directly to those nodes to run indexer on them.

pipes connected to the user's process and forward that data back to node threads in the rexec client. Local job control is done by forwarding signals as usual but also by translating certain signals to ones which have meaning for remote processes not attached to a terminal. (For example, SIGTSTP (C-z) is translated to SIGSTOP.)

3.3 Dynamic Cluster Membership

REXEC uses a dynamic cluster membership service to discover nodes as they join and leave the cluster using a well-known cluster multicast address. In our experience with large clusters of computers, we have found that over time the set of available nodes tends to vary as nodes are added, removed, rebooted, and so forth. Using static configuration files or manual intervention to track cluster membership is error-prone and inefficient. A dynamic membership service based on multicast avoids this and also has the desirable property that processes can communicate with interested receivers without explicitly naming them. Senders who wish to communicate 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.

Approximate membership of the cluster is maintained by replicated vexecd daemons by using the reception of a multicast rexecd announcement packet as a sign that a node is available and the non-reception of an announcement over a small multiple of a periodic announcement interval as a sign that a node is unavailable. Each rexecd sends announcement packets periodically (once every minute) and also whenever a significant change in state is observed. Currently, announcements based on state changes are sent for job arrivals and job departures. vexecd daemons discover and maintain the node membership in the cluster by caching and timing out node announcement information.

3.4 Decoupled Discovery and Selection

REXEC decouples node discovery from the selection of which nodes an application should run on. Replicated vexecd daemons are responsible for discovering and maintaining the

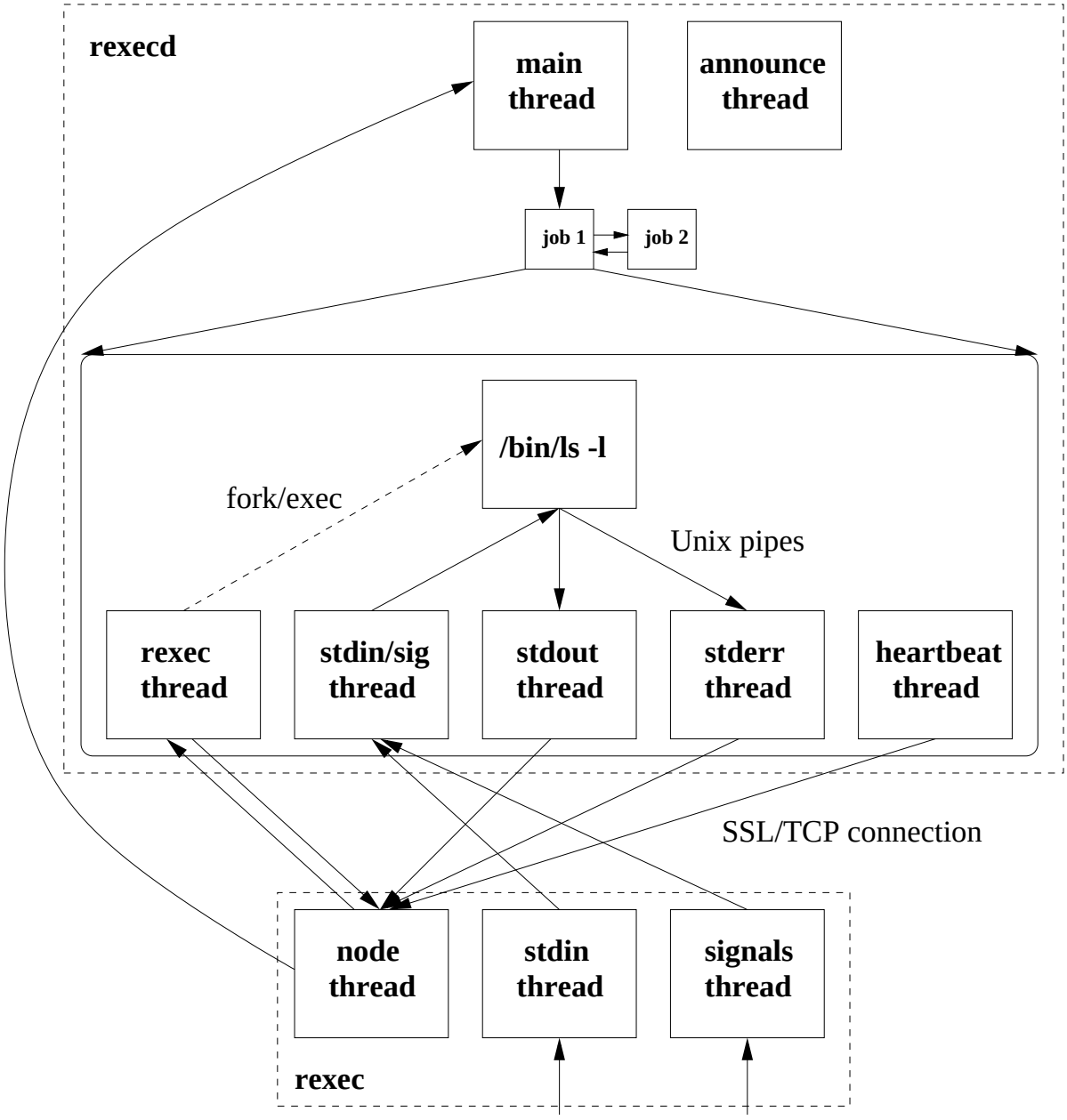

Figure 2: **Internal thread structure and data flows for rexec and rexecd.** rexec consists of a stdin thread for forwarding of stdin, a signals thread for forwarding of signals, and one node thread per node for managing remote process execution including propagation of the user's environment, the request to start the user's application, printing remotely forwarded stdout and stderr to the user's local terminal, receiving heartbeat packets (and rexec client monitoring of the 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 an rexec thread for the client, the user process forked and execed by the 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-encrypted TCP connection between rexec and rexecd.

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 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 (and end users) are free to implement arbitrarily complex selection policies.

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.

Discovery of vexecd hosts that implement suitable selection policies can be done through out-of-band means (e.g., posting on a web page) 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. 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 of parallel and sequential applications introduces new two classes of failures. First, failures can occur in the rexec client (the local point of control) and between the rexec client and the daemons. Since the rexec client logically represents an applications' remote process(es), failure of the rexec client is interpreted as failure of the application and the application is aborted. Second, failures can occur in individual processes of a parallel or distributed program. Since for all but the most trivially parallelizable and distributed programs there will be communication between processes and failure of one process usually means failure of the entire application, we interpret failure of an individual process in the program as a failure of the entire application. While these interpretations may not be true for all applications, we feel they are reasonable assumptions 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, the rexec client and rexecd daemons monitor each other through their TCP connections. Upon detecting an error, the rexec client exits. Upon detecting an error with some rexec client, rexecd frees all resources associated with that client and kills all its threads.

To ensure that proper process cleanup is performed on remote nodes, REXEC uses small modifications to the Linux kernel to track and control a logical set of processes whose first member is a user process forked by rexecd and other members are descendents of that process. To do this, we added a new system call to specify the first member of a logical set of processes (all descendents of that process inherit the fact that they are part of the same set) and a system call to deliver signals to all members of that set, regardless of changes in Unix process group, intermediate parents exiting causing their children to be inherited by init, and so forth. When performing cleanup in response to an error, REXEC simply sends SIGKILL to all processes in the logical set of processes, which results in all resources for all processes in the set being freed. We also modified the wait system call to

deal with logical sets of processes so a process p_x blocked on a wait call waiting for another process p_y to exit does not return until all processes in p_y's logical process set have exited. This feature is used by the per-rexec-client thread (Figure 3.2) so it returns only when all processes forked by the user's original process (i.e., the process which the per-rexec-client thread did a fork/exec on), including itself, have exited.

An alternative approach to cleanup, one which would have resulted in better portability of the system, would have been to send SIGKILL to the process group for the user's process that was forked by rexecd. An implementation of this approach uses standard POSIX interfaces. However, there are limitations and consequences. The biggest limitation is the inability to keep track of process relationships when process groups change. An example of this is a user process forked by rexecd which then forks a child and exits. rexecd would keep track of the parent's process group but since the parent has exited, the child now becomes inherited by init and becomes a member of an orphaned process group. Consequently, it becomes impossible to send a signal to the original process group. The orphaned child will not receive it and thus rexecd has lost track of a process. User processes could also call setpgrp themselves and a similar problem results. By keeping tracking process relationships in the kernel using our new system calls, we ensure that we always are able to kill all processes associated with a user process forked by rexecd.

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.

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 [6] 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 [4].

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.

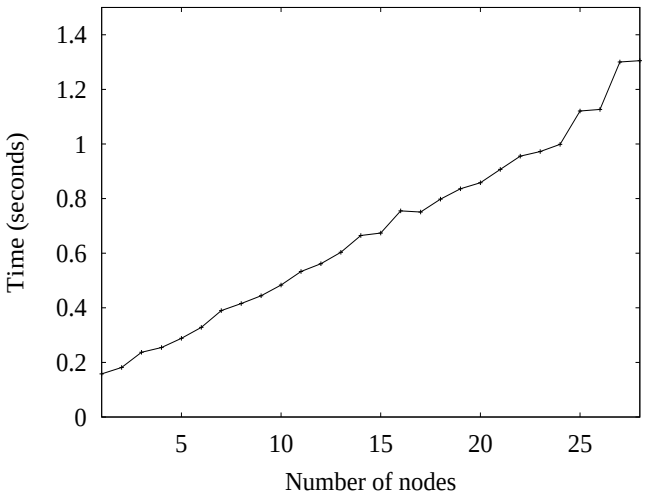

Figure 3: **Measured execution time to run a null parallel program with REXEC as a function of number of nodes.** This graph shows the measured execution time of a parallel program that starts and immediately exits on multiple nodes of the cluster. The measurements illustrate the basic costs associated 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.

Figure 3 illustrates the basic performance and scalability characteristics of REXEC. It shows the measured execution time of running a null parallel program that starts and immediately exits versus the number nodes the program was run on. 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, to date, we have mainly focused on other aspects of REXEC's design as stated in Section 2. We have not aggressively pursued performance optimizations on the system. Thus, absolute performance numbers as shown still have considerable room for improvement. The key point here is that the costs are scaling linearly with the number of nodes. Per-node costs will be optimized in a future version of REXEC.

4.2 MPI/GM on Myrinet

Using REXEC's basic hooks for parallel and distributed programs, we modified Myricom's MPI implementation over the GM (MPI/GM) fast communication layer [12] to use REXEC as its underlying remote execution mechanism. MPI rank and size are set using the REXEC_PAR_DEGREE and REXEC_MY_VNN environment variables. Communication is set up using REXEC_GPID and REXEC_SVRS to do an all-to-all exchange of GM port names using a centralized nameserver. Upon creating a GM port, each process binds a (key,value) = (REXEC_GPID:REXEC_MY_VNN, GM port number) pair in the nameserver then does REXEC_PAR_DEGREE lookups on keys REXEC_GPID:vnn (vnn = 0,1,..,REXEC_PAR_DEGREE-1). Since each GM network address is an IP address and a GM port number and each process knows the hostname (IP address) to VNN mapping from REXEC_SVRS, each process thus knows the GM network address of each process in the program and can then communicate. Using MPI/GM over REXEC, MPI programs can be started and controlled just like any other application run through REXEC.

4.3 Computational Economy

As part of the Berkeley Millennium Project, we 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, 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, a new command line switch (-r maxrate) was added to the rexec client to specify the maximum rate, expressed in credits per minute, the application is willing to pay for CPU time. Second, rexecd was modified 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 [20] and charging of user accounts for CPU usage. Third, we modified rexecd to include in its announcement packets the current aggregate willingness to pay of all REXEC applications competing for its resources for building selection policies based on the economy.

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 prob-

lems 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 [7], SCore-D [8], Sidle [9], Butler [13], HetNOS [3], and Load Sharing Facility (LSF) [22] 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 [2], NEST [1], COCANET Unix [16], and Solaris MC [10, 17] 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 [5, 11, 15, 19] 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 [14], V [18, 19], and LOCUS [21] 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 Future Work

Future work on the REXEC system comes in four areas. First, we intend to add a programmatic interface to REXEC that exposes REXEC's functionality to user applications though a user library. Using this interface, one of the applications we are planning to build is a shell that understands remote execution through REXEC and, in particular, makes it easier and more natural for users to use the computational economy. Second, we intend to add support for transparent remote execution of X applications and secure tunneling of X traffic over SSL. Techniques for implementing such support are well-known and already exist in programs such as the secure shell client (ssh). Third, we plan to pursue performance optimizations of the system to bring per-node costs down. As the Millennium system scales to hundreds of nodes as planned, optimization of such costs will become increasingly important for highly parallel applications. Finally, we intend to work on making REXEC portable across multiple operating systems and eventually plan on making a public release of the source code so others can use it and improve on it.

7 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] 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.

[3] 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).

[4] CHUN, B. N., AND CULLER, D. E. Market-based proportional resource sharing for clusters. Submitted for publication, September 1999.

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

[6] FREIER, A. O., KARLTON, P., AND KOCHER, P. C. The ssl protocol version 3.0 (internet-draft). 1996.

[7] 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).

[8] 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).

[9] JU, J., XU, G., AND TAO, J. Parallel computing using idle workstations. *Operating Systems Review* (July 1993).

[10] 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).

[11] 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.

[12] MYRICOM. The gm api. 1999.

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

[14] 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).

[15] PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. Libckpt: Transparent checkpointing under unix. In *Proceedings of the 1995 USENIX Winter Conference* (1995).

[16] 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).

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

[18] 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.

[19] 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).

[20] 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.

[21] 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.

[22] 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).