University of Southern California
Department of Contracts and Grants
Los Angeles, CA 90089-1147

DIstributed VIRtual System (DIVIRS) Project

formerly

Center for Experimental Research in Parallel Algorithms, Software, and Systems

Semiannual Progress Report #10
June 1993

Principal Investigator:
Herbert Schorr
Co-principal Investigator
B. Clifford Neuman
USC/Information Sciences Institute

Prepared under NASA Cooperative Agreement NCC 2-539
for Henry Lum, Technical Officer
NASA Information Sciences Division 244-7

Semiannual Progress Report
Covers period 1 November 1992 through 30 April 1993

As outlined in our continuation proposal 92-ISI-50R (revised) on contract NCC 2-539, we are (1) developing software, including a system manager and a job manager, that will manage available resources and that will enable programmers to program parallel applications in terms of a virtual configuration of processors, hiding the mapping to physical nodes; (2) developing communications routines that support the abstractions implemented in item one; (3) continuing the development of file and information systems based on the Virtual System Model; and (4) incorporating appropriate security measures to allow the mechanisms developed in items 1 through 3 to be used on an open network.

The goal throughout our work is to provide a uniform model that can be applied to both parallel and distributed systems. We believe that multiprocessor systems should exist in the context of distributed systems, allowing them to be more easily shared by those that need them. Our work provides the mechanisms through which nodes on multiprocessors are allocated to jobs running within the distributed system and the mechanisms through which files needed by those jobs can be located and accessed.

The Prospero Resource Manager

Conventional techniques for managing resources in parallel systems perform poorly in large distributed systems. To manage resources in distributed parallel systems, we have developed resource management tools that manage resources at two levels: allocating system resources to jobs as needed (a job is a collection of tasks working together), and separately managing the resources assigned to each job. The Prospero Resource Manager (PRM) presents a uniform and scalable model for scheduling tasks in parallel and distributed systems. PRM provides the mechanisms through which nodes on multiprocessors can be allocated to jobs running within an extremely large distributed system.

The common approach of using a single resource manager to manage all resources in a large system is not practical. As the system grows, a single resource manager becomes a bottleneck. Even within large local multiprocessor systems the number of resources to be managed can adversely affect performance. As a distributed system scales geographically and administratively, additional problems arise.

PRM addresses these problem by using multiple resource managers, each controlling a subset of the resources in the system, independent of other managers of the same type. The functions of resource management are distributed across three types of managers: system managers, job managers, and node managers. The complexity of these management roles is reduced because each is designed to utilize information at an appropriate level of abstraction.

During the reporting period, we continued development of the Prospero Resource Manager (PRM), software that manages tasks in parallel and distributed systems. We added support for I/O tasks

allowing remotely executing applications to read and write files on workstations that do not otherwise export their file systems and we improved the communication mechanisms on which PRM is implemented. A beta release was distributed to more than 50 Internet sites. A paper describing the Prospero Resource Manager was prepared, submitted to, and accepted for presentation at the Second International Symposium on High Performance Distributed Computing and will be presented in July. A copy of that paper is attached.

In our present configuration the nodes are Sun workstations running SunOS, Hewlett-Packard workstations running HP-UX, and an Intel 486 personal computer running Mach. Communication between the job, system, and node managers, and between tasks in a job, is supported by a reliable delivery protocol based on the user datagram protocol (UDP). To run a parallel application, programmers link executables for their tasks with the communication library we provide, they create a job description file, and they invoke the job manager passing it the name of the job description file.

We have used PRM to run the Ocean program from Stanford University's SPLASH benchmark suite, which studies the role of eddies and boundary currents in influencing large-scale ocean movements by solving a set of partial-differential equations. We started with a message-passing version of Ocean available for the Connection Machine (CM-5). To port this program to the PRM platform, we wrote a set of macros and routines to implement the CM-5 communication library functions using equivalent calls from our own library. The host program from the CM-5 version was incorporated into a terminal I/O task and handles interactive input. We are also using PRM to develop a simulator for large networks of neurons.

Our plans for the next year include continued development of PRM and integration of our resource management mechanisms with Parallel Virtual Machine (PVM) to allow existing users of PVM to use PRM without modifying their programs. We hope also to use PRM to make a prototype embeddable touchstone multi-processor (built by the EV project at ISI) available to Internet users. We have started to develop debugging and performance tuning tools for parallel applications. These tools take advantage of the user level job manager. Work is planned to support suspension and subsequent migration of tasks.

The Prospero File System and Directory Service

During the reporting period, we continued development of the Prospero File System and Directory Service, a file system and directory service based on the Virtual System Model. In March 1993, software supporting Version 5 of the Prospero protocol was released. These changes allow us to develop and extend additional applications (e.g. document and file system browsers, hypertext systems) to support Prospero. New access methods have been added to Prospero, including an e-mail method supporting non-real-time access to files available through e-mail requests to designated electronic mail addresses.

A prototype server supporting NFS access to files named by a Prospero virtual system was implemented. More work is needed to support access to more than one virtual directory. When completed, this server will allow existing applications to transparently use the Prospero File System without

relinking. A similar server supporting file access through the stackable layers file system interface from UCLA is planned.

We have begun integration of the Prospero Resource Manager with the Prospero Directory Service. This allows information about the configuration of parallel programs to be maintained as attributes of the programs themselves, rather than in a separate configuration file. This will eventually allow parallel applications to be invoked the same way sequential programs are, simply by typing their name; the configuration information stored in the directory entry for the program would be used to automatically invoke the appropriate job manager.

We have begun implementation of the Prospero Data Access Protocol to support secure retrieval of data from systems running Prospero. The Prospero Directory Service presently only provides directory information about files in the Prospero File System. Access to the files is supported automatically using existing access methods. The Prospero Data Access Protocol will provide a common protocol for access to local files and gateway access to remote files using alternative access methods, thus reducing the number of access methods that must be supported by Prospero applications. A preliminary implementation of the Prospero data access protocol is used by the Prospero Resource Manager for access to files on workstations that do not otherwise export their file system.

Security for Distributed Systems

During the reporting period, we have continued work to integrate appropriate security mechanisms into both Prospero and the Prospero Resource Manager. Continued improvements to the beta release of Version 5 of Kerberos from MIT were made, including support for the forwarding of authentication credentials, a mechanism that is necessary to securely allow remotely executing tasks to perform operations with the privileges of the user. These changes were fed back to MIT and were included in the latest release from MIT.

The widespread use of the computing and information infrastructure we are developing as part of the DIVIRS project requires an underlying security infrastructure to provide fine-grained access control mechanisms to protect such resources and accounting mechanisms to manage their use. We presented a paper at the 13th International Conference on Distributed Computing Systems discussing the need for such a security infrastructure and describing a possible mechanism to provide it. A copy of that paper is attached. We have implemented the Prospero Resource Manager and the Prospero Directory Service so that it can be easily integrated with such a system if it becomes available. In a separate proposal we have requested funding to develop such security infrastructure.

We have been notified that our proposal for an Augmentation Award for Science and Engineering Research Training (AASERT) attached to the DIVIRS contract has been selected for funding. This award will cover the cost of hiring an additional graduate student who will study computer security. Ari Medvinsky will be starting as an research assistant under that award in June.

APPENDIX A - GLOSSARY

AASERT	Augmentation Award for Science and Engineering Research Training
ARPA	Advanced Research Projects Agency
DIVIRS	Distributed Virtual Systems
EV	Embeddable Variant
I/O	Input/Output
ISI	Information Sciences Institute
MIT	Massachusetts Institute of Technology
NFS	Sun's Network File System
PRM	Prospero Resource Manager
PVM	Parallel Virtual Machine
UCLA	University of California, Los Angeles
UDP	User Datagram Protocol
USC	University of Southern California

APPENDIX B - PAPERS

The following papers were prepared and accepted for publication during the reporting period. Copies of the papers are attached to this report.

B. Clifford Neuman and Santosh Rao. Resource Management for Distributed Parallel Systems. In *Proceedings of the 2nd International Symposium on High Performance Distributed Computing.* Spokane, July 1993.

B. Clifford Neuman. Proxy-Based Authorization and Accounting for Distributed Systems. In *Proceedings of the 13th International Conference on Distributed Computing Systems.* Pages 283-291. Pittsburgh, May 1993.

Resource Management for Distributed Parallel Systems

B. Clifford Neuman   Santosh Rao

Information Sciences Institute
University of Southern California

Abstract

Multiprocessor systems should exist in the the larger context of distributed systems, allowing multiprocessor resources to be shared by those that need them. Unfortunately, typical multiprocessor resource management techniques do not scale to large networks. The Prospero Resource Manager (PRM) is a scalable resource allocation system that supports the allocation of processing resources in large networks and multiprocessor systems.

To manage resources in such distributed parallel systems, PRM employs three types of managers: system managers, job managers, and node managers. There exist multiple independent instances of each type of manager, reducing bottlenecks. The complexity of each manager is further reduced because each is designed to utilize information at an appropriate level of abstraction.

1 Introduction

Conventional techniques for managing resources in parallel systems perform poorly in large distributed systems. We believe that multiprocessor systems should exist in the broader context of distributed systems, allowing them to be shared by those that need them. To manage resources in distributed parallel systems, we have developed prototype resource management tools that manage resources at two levels: allocating system resources to jobs as needed (a job is a collection of tasks working together), and separately managing the resources assigned to each job.

This paper describes the model and implementation of these tools which were developed for use by the Prospero operating system, under development at the University of Southern California's Information Sciences Institute. The Prospero Resource Manager (PRM) presents a uniform and scalable model for scheduling tasks in parallel and distributed systems. PRM provides the mechanisms through which nodes on multiprocessors can be allocated to jobs running within an extremely large distributed system.

It is our belief that the common approach of using a single resource manager to manage all resources in a large system is not practical. As the system to be managed grows, a single resource manager becomes a bottleneck. Even within large local multiprocessor systems the number of resources to be managed can adversely affect performance. As a distributed system scales geographically and administratively, additional problems arise.

PRM addresses these problem by using multiple resource managers, each controlling a subset of the resources in the system, independent of other managers of the same type. The functions of resource management are distributed across three types of managers: system managers, job managers, and node managers. The complexity of these management roles is reduced because each is designed to utilize information at an appropriate level of abstraction.

While the development of PRM was motivated by the desire to support parallel computing across organizations in a distributed system, the same techniques can improve the scalability of scheduling mechanisms within independent tightly coupled multiprocessor systems. The abstractions provided also naturally suggest extensions that support fault-tolerant and real-time applications and debugging and performance tuning for parallel programs.

Throughout this paper we use the term *node* to denote a processing element in a multiprocessor system, or a workstation or other computer whose resources are made available for running jobs. A *job* consists of a set of communicating tasks, running on the nodes allocated to the job. A *task* consists of one or more threads of control through an application and the address space in which they run.

2 Contemporary approaches

Figure 1 shows the functions involved in the execution of a parallel application in a distributed environment. In the first step (1), the application is compiled and installed and information about resource require-

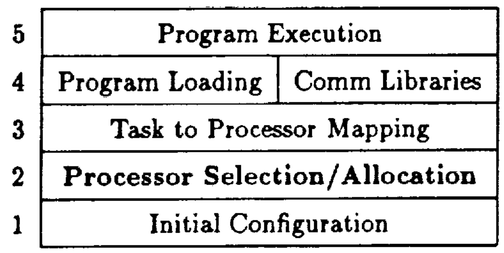


Figure 1: Support for distributed execution

ments and available resources are specified by the user or programmer. This information is used in (2) to select and allocate nodes on which the program will run. The tasks are mapped to the allocated nodes in (3) and the executable modules (the tasks) are loaded onto the appropriate nodes in (4). The execution of the program (5) depends on run-time communication libraries (also at 4) which in turn use information about the mapping of tasks to nodes (3).

Locus [10], NEST [1], Sprite [5], and V [13] support processor allocation, and remote program loading and execution (2,4,5) to harness the computing power of lightly loaded nodes. They primarily support sequential applications where task-to-task communication is not required. A critical issue for processor allocation in these systems is the maintenance of the database of available nodes. In Locus the target node for remote execution is selected from a list of nodes maintained in the environment of the initiating process. This approach is inflexible because the nodes available for remote execution do not change with changing load.

In NEST [1], idle machines advertise their availability, providing a dynamically changing set of available nodes; each user's workstation maintains the list of servers available for remote execution. Locus and NEST both require the application to maintain information about every possible target node, limiting the size of the pool from which nodes can be drawn. Additionally, resource allocation decisions in these systems are made locally by the application without the benefit of a high level view across jobs. This causes problems when applications run simultaneously.

Sprite [5] uses a shared file as a centralized database to track available nodes. Clients select idle nodes from this file, marking the entry to flag its use. While this approach appears simple, it requires a solution to problems related to shared writable files, including locking, synchronization and consistency. Fault-tolerance is also poor since failure of the file server on which the shared file resides disables the allocation mechanism completely. This approach does not scale beyond a few tens of nodes.

Theimer and Lantz experimented with two approaches for processor allocation in V [13]. In a centralized approach a central server selects the least loaded node from a pool of free nodes and allocates it. Nodes proclaim their availability based on the relationship of the local load to a cutoff broadcast periodically by the server. This approach has limited scalability and poor fault tolerance since the central server is a critical resource. In the distributed approach a client multicasts a query to a group of candidate machines selecting the responder with the lowest load. This approach suffers from excessive network traffic and was found to perform worse than the central server approach.

The UCLA Benevolent Bandit Laboratory (BBL) [6] provides an environment for running parallel applications on a network of personal computers. Like the other systems discussed, BBL provides processor allocation, and remote program loading and execution (2,3,4,5), incorporating the notion of a user-process manager separate from a systemwide resource manager. While this is an important step towards scalable resource management techniques, a single resource manager will be unable to handle all allocation requests for a large system.

Parallel Virtual Machine (PVM) [12] and NetExpress [9] allow users to run parallel applications on a collection of workstations. In the initial configuration phase, users specify a list of nodes on which they have started daemon processes. Based on this configuration, PVM and NetExpress map a job's tasks to nodes, load and execute the tasks, and support communication between tasks (3,4,5). There is no support for high-level resource allocation functions (2) that assign nodes to jobs with the goal of efficient system utilization. All nodes specified by the user are available to the job whether or not they are already in use by other jobs.

Efficient management of a pool of processors becomes very important when the system scales to large numbers of nodes, spanning multiple sites. The emphasis of the Prospero Resource Manager is the allocation of nodes across and within jobs (2). The job manager eliminates the need for users to enumerate all hosts on which their applications might run, while the system manager efficiently manages the system's resources. While PRM also supports task mapping, program loading, and execution (3,4,5), it is the allocation function (2) that distinguishes it from PVM and NetExpress. We encourage the integration of the PRM allocation methods with both packages.

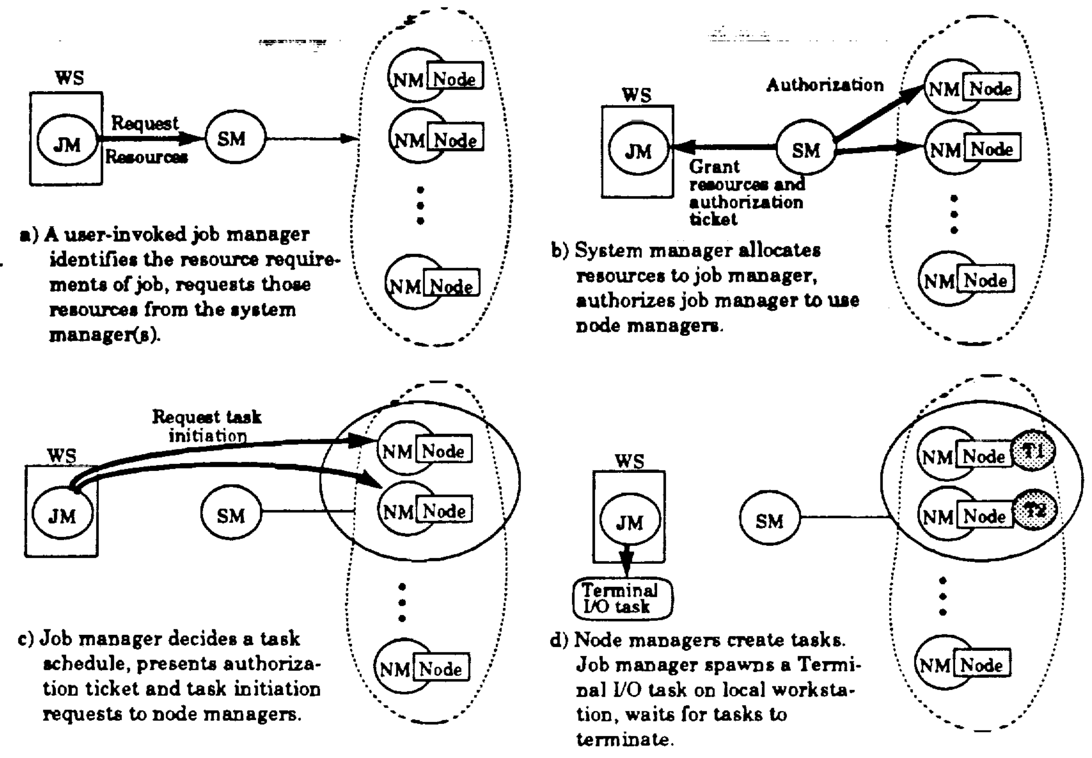

WS
JM
Request Resources
SM
NM Node
a) A user-invoked job manager identifies the resource requirements of job, requests those resources from the system manager(s).
Authorization
Grant resources and authorization ticket
b) System manager allocates resources to job manager, authorizes job manager to use node managers.
Request task initiation
c) Job manager decides a task schedule, presents authorization ticket and task initiation requests to node managers.
Terminal I/O task
T1
T2
d) Node managers create tasks. Job manager spawns a Terminal I/O task on local workstation, waits for tasks to terminate.

Figure 2: Running a job with PRM

3 Scalable resource management

Users have difficulty dealing with extremely large systems because, although only a small subset of the available resources are needed, it is difficult to identify the resources that are of interest among the clutter of those that are not. Today's users are able to cope because only a tiny portion of the world's resources are available to them. Managing the world's resources is a daunting task, but the problem is simplified when it is reduced to managing only a subset of the resources.

We believe it should be possible to organize *virtual systems* in which resources of interest are readily accessible, and those of less interest are hidden from view. The organization of such systems should be based on the conceptual relationship between resources and the mapping to physical locations should be hidden from the user. These concepts form the basis of the Virtual System Model, a new model for organizing large distributed systems [8].

To apply the concepts of the Virtual System Model to the allocation of resources in large systems, we have chosen to divide the functions of resource management across three types of managers: the system manager, the job manager, and the node manager. The system manager controls a collection of physical resources, allocating them to jobs when requested. The job manager is responsible for requesting the resources needed by a job, and once allocated, assigning them to the individual tasks in the job. The node manager runs on each processor in the system, loading and executing tasks when authorized by the system manager and requested by the job manager. Each manager makes scheduling decisions at a different level of abstraction, some concerned with the high level performance of the system, and others concentrating on particular jobs.

3.1 The system manager

The full set of resources that exist in a system will be managed by a set of system managers. For example, one or more system managers might manage the nodes in a parallel computer, or the resources local to a particular site. System managers allocate their resources across jobs as needed. We do not believe that it is possible to build a single system manager to manage all resources in a large system. As the system to be managed grows, a single system manager would become a bottleneck. To avoid this problem, our system supports multiple system managers, each responsible for a collection of resources. For example, one system manager can be responsible for the processors on a multiprocessor system and when necessary for performance or other reasons, multiple system managers may exist, each controlling a disjoint subset of the processors on the multiprocessor. An independent system manager might manage the resources available on one or more workstations.

The system manager is a hierarchical concept. Several sets of resources may be managed by different system managers, with a higher level system manager responsible for the entire collection. The control of resources could then be transferred from one system manager to another as directed by the higher level manager.

The system manager keeps track of the resources for which it is responsible, maintaining information about the characteristics of each resource, whether it is currently available, and if assigned, the job to which it is assigned. The system manager responds to status updates from node managers and resource requests from job managers. Status updates provide information needed to make allocation decisions including availability and load information. Resource requests identify the resources required by a job, their characteristics, as well as connectivity constraints, but only in well defined ways. It is possible to extend the system manager to accept messages from higher level managers (or other entities) adding or removing resources from its control.

When a resource request is received from a job manager, the system manager determines whether suitable resources are available as defined by the characteristics specified in the request. If so, the system manager assigns them to the job, notifies the node managers responsible for each resource that the resource has been assigned to a specific job manager, and informs the job manager of the resources that have been assigned. If the requested resources are not available the system manager can, at the job managers option (and subject to the scheduling policy of the system manager), assign a subset of the requested resources and/or reserve the resources for assignment when they become available.

3.2 The job manager

Although multiple system managers are necessary for scalability, the application needs a single point of contact for requesting resources. In our system, this point of contact is the job manager. The job manager acts as an agent for the tasks in a job, providing a single entity from which the tasks will request resources. In this capacity the job manager provides the abstraction of a virtual system to a job, managing the resources that have been allocated to a job by the system managers responsible for each resource. Although it is possible for a job to have more than one job manager, in most cases only one exists.

The job manager is part of a job and is aware of the specific requirements and communication patterns of the tasks it manages. As such, the job manager is better able than the system manager to allocate resources to the individual tasks within a job. This is the same argument used in favor of user-level thread management on shared-memory multiprocessors [2]. In fact, we allow the job manager to be written by the application programmer if specific functionality is required, though we do not expect this to be a common practice.

We plan to eventually provide alternative job managers to support fault-tolerant and real-time applications. Such job managers would add additional requirements to the resources requested from system managers, and might assign individual tasks to multiple nodes. Similarly, a job manager is planned that will collect information needed for debugging and performance tuning. The programmer would then be able to select job managers tailored to the needs of the application or the phase of program development; when an application is ready for production use, a different job manager could be substituted.

At the time a job is initiated, the job manager identifies the job's resource requirements. Using the Prospero Directory Service [7], if available, or a configuration file otherwise, it locates system managers with jurisdiction over suitable resources and sends allocation requests. If the system managers respond affirmatively, the job manager allocates the resources to the tasks in the job, contacting the node manager for each resource to initiate the loading of programs onto the appropriate processors. If the system manager refuses the allocation request, the job manager will try to identify alternate resources from other system managers. If necessary, the job manager will additionally create tasks to handle I/O to the terminal or to files on the local system.

Once the job has been initiated on the assigned nodes, the job manager monitors the execution of the program. During program execution the job manager responds to requests from the job's tasks for additional resources, reallocating them from other tasks or requesting additional resources from suitable system managers. In this phase the job manager also maintains information about the mapping of logical task identifiers to node identifiers, for use by the communication library.

3.3 The node manager

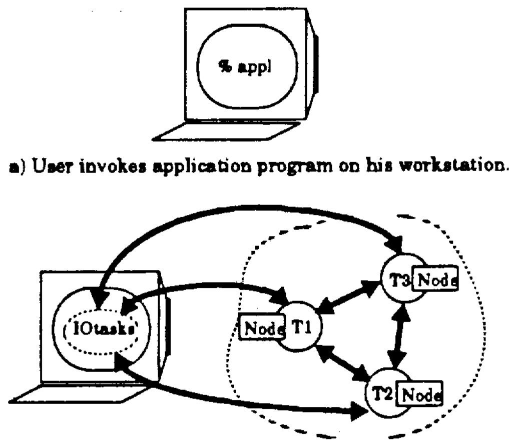
The third component of our resource management suite is the node manager. A node manager runs on each processor in the system, eventually as part of the kernel but in the current implementation as a user-level process. The node manager accepts messages from the system manager identifying the job managers that will load and execute programs. When requested by an authorized job manager, the node manager loads and executes a program. The node manager notifies the job manager about events such as the termination and failure of tasks. The node manager also keeps the system manager informed about the availability of the node for assignment. The node manager caches information needed to direct messages for other tasks to the node on which the task runs.

3.4 Application invocation

Each program that executes under PRM has associated with it information about the virtual system on which it will run. This information is stored either in a configuration file or as attributes of the program in the Prospero Directory Service [7]. When a program is invoked, a new job manager is created and the job manager finds a suitable processor or set of processors by contacting system managers identified by the virtual system associated with the program.

Figure 2 shows the steps involved in running an application using PRM. Our goal is for users to invoke programs as if they were local to a workstation. When the program is invoked, a job manager is automatically started on the workstation[1]. The job manager determines the resource requirements of the job and sends requests to one or more system managers. If the requested resources are available the system manager informs the node manager responsible for each resource that the resource has been assigned to a particular job manager and it returns a list of the assigned resources to the job manager. The job manager further allocates the assigned resources to the job's tasks then contacts the node manager for each resource to invoke the application. Upon receipt of a request from the authorized job manager, each node manager loads the application task.

During job execution, the job manager responds to requests from the job's tasks for additional resources (additional processors for example) and to preemption and migration requests from the system managers responsible for the resources in use and, if necessary, attempts to obtain additional resources from other system managers. The job manager acts as an agent for



a) User invokes application program on his workstation.

b) A set of tasks are created. Tasks execute the program, communicate with each other and perform terminal I/O.

Figure 3: Input/Output using PRM

the user, hiding the details of parallel execution; the user's environment and shell are maintained on the workstation and the job manager decides where each command is to execute, hiding the details from the user for whom local sequential execution and parallel execution appear identical.

3.5 Discussion

By separating the job and system manager, the system manager becomes much simpler. The system manager is concerned only with the allocation of resources between jobs, eliminating application specific functionality. The job manager is part of a job, and has more information about the requirements and communication patterns for the tasks it manages. Thus, the job manager is in a better position to allocate resources to tasks once resources have been allocated to the job. Because the job manager is part of the job, it can be customized or even rewritten if application specific functionality is required.

4 Implementation and performance

The current implementation of the Prospero Resource Manager runs on a collection of Sun3, Sparc, and HP9000/700 workstations, connected by a local or wide-area network. Even though PRM's emphasis is on resource allocation, we provide an application programming interface (in the form of macros and libraries) that provide the user with a programming and execution environment similar to that on a multicomputer. Heterogeneous execution environments are supported; the tasks in a job may execute on different processor types, and the set of nodes executing a job need not share a common filesystem. In the latter case PRM handles program loading and I/O to shared files. This is shown in figure 3.

[1] Though in some cases, it might migrate to another node.

Msg len (bytes)	4	64	1024	4096	8192
Latency (msec)	4.4	4.6	5.9	15.7	28.5

Table 1: Effect of message length on latency

Application programs are based on message-passing and are coded in C. A library is provided with PRM for sending and receiving tagged messages, broadcasting, and global synchronization. In designing this library, a layered approach has been used to facilitate easy integration with low-level communication protocols. In the present implementation, these library functions use an Asynchronous Reliable Delivery Protocol (ARDP) that transmits and receives sequenced packets over the Internet using UDP. We are currently implementing a version of the communication library layered on top of the Mach *port* mechanism [4]. The global synchronization and reduction primitives rely on centrally maintained state information. To the extent possible, we plan to develop distributed implementations of these primitives, perhaps building on the primitives provided by ISIS [3].

PRM's transparent message routing mechanism frees the programmer from having to explicitly keep track of task to node mappings. At the application level, messages are addressed using *task-identifiers* (*tids*), which are translated to an internet-address/port pair by the communication library. The node manager assists tasks in this translation using a mapping table furnished by the job manager. Such translations are then cached in the local address space of the task to reduce address translation overhead for subsequent communication with the same task.

4.1 Running jobs under PRM

Multiple users may share a common PRM environment. Once setup, system and node managers run as server processes. The system manager maintains the *availability status* for each node, and allocates available nodes to jobs when requests are received. Depending on options specified at setup, a node manager may make its processor unconditionally available for running jobs, or available only within specified time windows, or when no user is logged onto the workstation (or a combination of the latter two options). Node managers notify their system manager of any changes in node availability.

A user initiates a job by invoking a job manager process on the workstation, specifying a configuration file from which resource requirements of the job are to be read. Typically, this file contains the number of nodes required to run the job, the number and location of I/O tasks, the path names for executable files and the host names of one or more system managers that can potentially satisfy the resource requirements. The current implementation requires the user to write a configuration file for the job. In a future release, static resource requirements will be generated by a compiler (possibly with some help from the programmer) and stored as file attributes. At run time, the user may override static specifications and specify runtime requirements as command line arguments to the job manager.

Terminal and file I/O are handled by two special tasks created with every job. The job manager schedules these I/O tasks on nodes with local I/O devices. For example, an I/O task may run on a file-server, performing file-I/O on behalf of the tasks. The terminal I/O task supports interactive execution. It is analogous to the host task in the host-node model of execution. Users can customize this task for job initialization functions, such as prompting the user for interactive input and distributing this input to the appropriate tasks.

Number of tasks	Message Length				
	4	64	1024	4096	8192
4	33.3	34.6	37.8	181.8	342.9
8	68.4	70.0	83.7	347.3	560.6

Table 2: Latencies for broadcast operation (msec)

4.2 Performance

Table 1 shows the measured average latencies for synchronous *send* operations as a function of message size. Experiments were conducted with a pair of tasks running on two Sun 4/60 workstations. After an initial message to obtain the internet-address and port of the destination from the node manager, the sender executed a loop, repeatedly sending messages to the receiver, which sent back an acknowledgment. On the sender, the total time spent in the loop was measured using the `gettimeofday()` system call. The increase in latency for messages up to 1K bytes is due primarily to variations in the overhead imposed by the ARDP library and Unix system calls. Longer messages are split into 1 K byte chunks and require the processing of additional packets. We are improving our code to reduce the overhead imposed by the ARDP library.

Table 2 shows latencies for broadcast operations as a function of message length and number of tasks. Broadcast is implemented in two phases. Data is first transmitted sequentially but asynchronously to all recipients using *send* operations. A second series of sends provides synchronization, assuring recipients that all other recipients have received the data. Broadcast latencies are of the order of $2n$ times the latency of a *send* operation were n is the number of recipients.

We are also testing our system using real scientific applications. The *Ocean* program from the SPLASH benchmark suite from Stanford University [11] studies the role of eddies and boundary currents in influencing large-scale ocean movements by solving a set of partial-differential equations. We started with a message-passing version of Ocean available for the Connection Machine (CM-5). To port this program to the PRM platform, we wrote a set of macros and routines to implement the CMMD library functions [14] using equivalent calls from our own library. The host program from the CM-5 version was incorporated into the terminal I/O task and handles interactive input. We are also using PRM to develop a simulator for large networks of neurons.

5 Future directions

The current implementation demonstrates only a few of the benefits of our resource management model. The greatest benefit of the model is its flexibility. The prototype provides a framework within which we can try experimental solutions to interesting problems, and upon which interesting tools may be built.

Among our planned experiments are the use of interchangeable scheduling policies by the system manager. Because our model supports multiple system managers, multiple scheduling policies can be applied simultaneously to disjoint sets of resources. We also plan to explore options for hierarchical configuration and dynamic reconfiguration of the nodes for which a system manager is responsible.

We plan to extend the job manager to make use of information such as task memory and I/O requirements, and intertask communication graphs to find an optimal assignment of tasks to nodes. We also plan to add support for preemptive scheduling across tasks when insufficient processing resources are available for the job. As discussed in section 3, we plan to develop special job managers to support fault-tolerant and real-time applications, debugging, and performance tuning.

We must also extend the node manager to support suspension of tasks and to allow the job manager to initiate non-interfering operations on the resources of the suspended tasks that it manages. In particular this will allow the migration of a task to an alternate node. The rôle of the node manager in translating task identifiers to host addresses would then include invalidation of stale translations cached by the tasks.

Our communications primitives are still quite rough. We intend to fine tune the current mechanisms, and where communication between nodes is possible using high-performance mechanisms specific to the host system (*e.g.*, within a single multiprocessor) we will use those mechanisms. We will also support execution of user programs originally developed for other platforms by linking them with the appropriate set of communication libraries. The macros for CMMD library calls in the Ocean program already represent a step in this direction.

I/O to files is still a limiting factor for many applications. This is especially troublesome when the processors on which an application runs are separated geographically from the disk on which a file is stored. The file I/O task plays an important role, supporting read and write operations to files on computers that do not export their file system. We are extending this protocol to support file caching.

Finally, with the ability to run applications on multiprocessor systems across wide-area networks, security will become a critical problem. It is unlikely that sites would make their resources available to others if there are no methods for protection. Security mechanisms are needed to control access to remote nodes, to allow remote tasks to securely retrieve data that might be stored across a wide-area network, and to account for the use of processing resources. We plan to incorporate further security mechanisms into the software we develop, concentrating initially on authentication and authorization mechanisms to be applied when a request is received by the system and node managers.

6 Conclusions

The growth of a distributed system brings with it increased complexity for the management of available resources. While it is possible to manage resources in different parts of the system separately, such an approach makes life difficult for the users and programmers who must interact with more than one entity to obtain the resources needed by an application. Unfortunately, centralized management techniques for parallel systems are not suitable for large distributed systems either.

The Prospero Resource Manager provides a solution between the two extremes. An individual manager controls a subset of the resources in the system independent of other managers of the same type. The system manager controls resources that are physically or administratively related. The job manager controls resources that are logically related, *i.e.* those resources needed by a particular job. When resources are needed, an application requests resources from its job manager which in turn requests the resources from one or more suitable system managers. The resources obtained are then reallocated across the tasks in the job. In this capacity the job manager acts as an agent for the user, presenting the abstraction of a virtual system.

The current prototype demonstrates the usefulness of the model and provides a framework within which we can experiment with different approaches and upon which interesting tools may be built. Section 5 highlighted the flexibility of the model. The true test of the model will be the extent to which it is employed in future systems.

Acknowledgments

Ed Lazowska, John Zahorjan, and Hank Levy contributed to discussions about the Virtual System Model. The initial idea of separate job and system managers for scheduling tasks on a multiprocessor system was proposed by Stockton Gaines and Dennis Hollingworth. Rafael Saavedra provided code for the CM-5 version of Ocean and also guided the implementation of the communication library. Bradford Clark, Ali Erdem and Ronald Wood implemented some of the functions in the communication library as part of a class project at USC. Celeste Anderson, Steven Augart, Gennady Medvinsky, Rafael Saavedra, and Stuart Stubblebine commented on drafts of this paper.

References

[1] R. Agrawal and A. K. Ezzat. Processor sharing in NEST: A network of computer workstations. In *Proceedings of the first International Conference on Computer Workstations*, pages 198–208. November 1985.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[3] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 123–138, November 1987.

[4] David L. Black, David B. Golub, Daniel P. Julin, Richard F. Rashid, Richard P. Draves, Randall W. Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel operating system architecture and Mach. In *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 11-30, April 1992.

[5] F. Douglis and J. Ousterhout. Transparent process migration for personal workstations. Technical Report UCB/CSD 89/540, Computer Science Division, University of California, Berkeley CA 94720, November 1989.

[6] R. E. Felderman, E. M. Schooler, and L. Kleinrock. The Benevolent Bandit Laboratory: A testbed for distributed algorithms. *IEEE Journal on Selected Areas in Communications*, 7(2):303–311, February 1989.

[7] B. Clifford Neuman. The Prospero File System: A global file system based on the Virtual System Model. *Computing Systems*, 5(4), Fall 1992.

[8] B. Clifford Neuman. *The Virtual System Model: A Scalable Approach to Organizing Large Systems.* PhD thesis, University of Washington, June 1992.

[9] ParaSoft Corporation, Pasadena, CA 91107. *NetExpress 3.2 Introductory Guide*, 1992.

[10] G. Popek and B. Walker, editors. *The Locus Distributed System Architecture.* M.I.T. Press, Cambridge, Massachusetts, 1985.

[11] J. P. Singh and J. L. Hennessey. Finding and exploiting parallelism in an ocean simulation program. *Journal of Parallel and Distributed Computing*, 15(1), May 1992.

[12] V. S. Sunderam. PVM: A framework for parallel distributed computing. Technical Report TM-11375, Oak Ridge National Laboratory, 1991.

[13] M. A. Theimer and K. A. Lantz. Finding idle machines in a workstation-based distributed system. *IEEE Trans. Software Engineering*, 15(11):1444–1458, November 1989.

[14] Thinking Machines Corporation, Cambridge, MA. *CMMD Reference Manual*, Oct. 1991.

This research was supported in part by the Advanced Research Projects Agency under NASA Cooperative Agreement NCC-2-539. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA or NASA. To obtain the software described in this paper, send mail to info-prospero@isi.edu asking for information about the Prospero Resource Manager. The authors may be reached at USC/ISI, 4676 Admiralty Way, Marina del Rey, CA 90292-6695, USA. Telephone +1 (310) 822-1511, email bcn@isi.edu, srao@isi.edu.

Proxy-Based Authorization and Accounting for Distributed Systems

B. Clifford Neuman
Information Sciences Institute
University of Southern California

Abstract

Despite recent widespread interest in the secure authentication of principals across computer networks there has been considerably less discussion of distributed mechanisms to support authorization and accounting. By generalizing the authentication model to support restricted proxies, both authorization and accounting can be easily supported. This paper presents the proxy model for authorization and shows how the model can be used to support a wide range of authorization and accounting mechanisms. The proxy model strikes a balance between access-control-list and capability-based mechanisms allowing each to be used where appropriate and allowing their use in combination. The paper describes how restricted proxies can be supported using existing authentication methods.

1 Introduction

The problem of authentication across computer networks has received much attention in recent years. Authentication is often only a step in the process of authorization or accounting. The goal is to verify that the individual making a request is authorized to do so, or to guarantee that the correct individual is charged for an operation. Despite the close ties among these problems, little progress has been made in providing secure, widespread, distributed mechanisms for authorization and accounting. To date, authorization and accounting have most often been supported locally by a server, instead of by the use of distributed authorization or accounting services. Such authorization and accounting services will be critical as the network is used more and more for electronic commerce and other applications where clients and servers not previously known to one another must interact. By generalizing the authentication model to support restricted proxies, distributed authorization and accounting can be easily supported.

This paper presents a unified model for authentication, authorization, and accounting that is based on proxies. Section 2 defines the term proxy and briefly describes how proxies can be supported by existing authentication mechanisms. The use of proxies for authorization is demonstrated in Section 3. The proxy model strikes a balance between access-control-list and capability-based mechanisms allowing each to be used where appropriate and allowing their use in combination. Section 4 discusses the necessary features of a distributed accounting service and shows how accounting fits the model. Section 5 discusses related work on distributed authorization and accounting. Integration of the described mechanisms with existing authentication systems is discussed in Section 6, and Section 7 discusses some of the more useful restrictions that can be supported. Section 9 draws conclusions.

2 Restricted proxies

A *proxy* is a token that allows one to operate with the rights and privileges of the principal that granted the proxy. Naturally, it must be possible to verify that a proxy was granted by the principal that it names. This is an authentication problem. In fact a principal with the credentials[1] needed to authenticate itself can often grant a proxy to another principal simply by passing on those credentials.

Implementing proxies in this manner has several shortcomings. First, the proxy can be used by anyone that gets hold of it. This won't always be a problem, but in many cases one should be able to specify the principal that is to act on one's behalf. Second, a proxy is all or nothing. The individual who has been granted the proxy can do anything that the grantor could do on any service to which the original credentials applied.

[1] Credentials consist of an encrypted certificate together with information needed to use the certificate.

Certificate: $[restrictions, K_{proxy}]_{grantor}$
Proxy-key: K_{proxy}

Figure 1: A restricted proxy

A *restricted proxy* is a proxy that has had conditions placed on its use. A principal possessing authentication or authorization credentials can generate a restricted proxy, a new set of credentials which are more restricted than the original credentials; it is not possible to remove restrictions. It must be possible for the server to which a restricted proxy will be presented (the end-server) to verify that the restrictions have not been tampered with. Among the restrictions that are often specified are that the proxy may only be used by a designated principal, or that the operations that may be performed are to be restricted.

When a principal issues a restricted proxy to another principal, the second principal is authorized to perform all operations for which the first principal is authorized on the server or servers for which the proxy is applicable, subject to any restrictions recorded in the proxy. In the discussion that follows, the *grantor* is the principal on whose behalf a proxy allows access. The *grantee* is the principal designated to act on behalf of the grantor. The *end-server* is the server to which the proxy must be presented to perform an operation.

The implementation of restricted proxies relies on the use of encryption-based authentication of the original grantor of the proxy. Either conventional or public-key cryptography may be used. In this section I describe the implementation at a high level, independent of the authentication mechanism in use. The description assumes that the infrastructure needed to authenticate the original grantor of a proxy is in place and messages required by the underlying authentication protocol (e.g., for key distribution) are omitted for clarity. These details, which are specific to the underlying authentication mechanism, are described in Section 6.

A restricted proxy has two parts: 1) a certificate signed by the grantor establishing the proxy, enumerating any restrictions, and establishing an encryption (or integrity) key[2] to be used by the end-server to verify that the proxy was properly issued to the bearer, and 2) a proxy key, an encryption (or integrity) key corresponding to the key embedded in the certificate, that will be used by the grantee to prove proper possession of the proxy. Figure 1 shows the contents of a restricted proxy; square brackets indicate a signature by the principal indicated in the subscript, or under

[2]Depending on the authentication mechanisms in use, this key may require additional protection from disclosure.

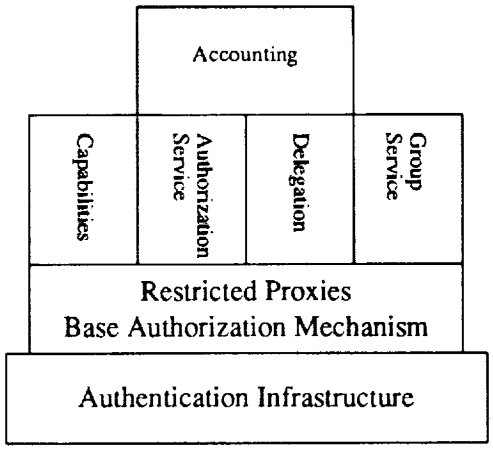


Figure 2: Relationship of security services

a separate encryption (or integrity) key. When a restricted proxy is transferred from the grantor to the grantee, care must be taken to protect the proxy key from disclosure.

There are two classes of proxies: bearer proxies and delegate proxies. A bearer proxy may be used by anyone. A delegate proxy may only be used by a principal named in a list of delegates (encoded as a restriction), or by someone with a suitable additional proxy issued by a named delegate.

To present a bearer proxy to an end-server, the grantee sends the certificate to the server and uses the proxy key to partake in an authentication exchange with the end-server using the underlying authentication mechanism. Usually this exchange involves sending a signed or encrypted timestamp or server challenge, proving possession of the proxy key.

To present a delegate proxy, the grantee sends the certificate to the end-server and then authenticates itself to the end-server under its own identity. The end-server validates the certificate and verifies that the client is included in the list of delegates specified by the proxy.

3 Authorization

Restricted proxies provide the vehicle for implementing a wide range of authorization mechanisms in distributed systems. In this section I describe several such mechanisms and show how they can be supported. Accounting mechanisms are described in Section 4 and build upon the authorization mechanisms described here. Figure 2 shows the relationship of such mechanisms to restricted proxies and to the authentication infrastructure on which they depend.

3.1 Capabilities

A capability can be thought of as a bearer proxy that is restricted to limit the operations that can be performed and the objects that can be accessed. No restrictions are placed on the identity of the grantee who is free to pass the capability to others. When presented to the end-server, the grantor's rights (as limited by the restrictions) are available to the bearer.

For example, to create a read capability for a particular file, a user authorized to read that file requests a restricted proxy for use at the file server containing the file (the end-server), but with the restriction that it can only be used to read the named file. The capability is then passed to others who can themselves pass it on. To use a capability, the bearer presents it to the file server in place of (or in addition to) the bearer's own credentials. If the request is to read the file named in the capability, the operation is performed with the rights of the grantor of the proxy.

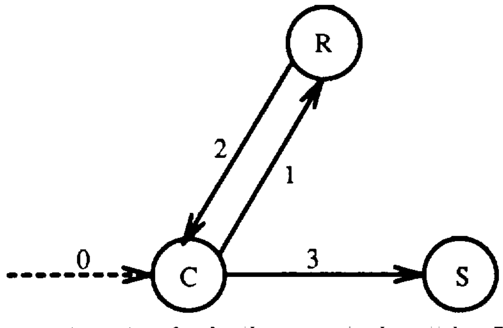
A capability as described above differs from traditional capabilities in several ways. One of the most important distinctions is that in presenting a capability (restricted proxy) to the end-server, the bearer does not send the entire proxy across the network. Instead, the bearer sends the certificate part of the proxy and proves possession by taking part in an authentication exchange using the proxy key as described earlier. The result is that an attacker can not obtain such a capability by tapping the network to observe the presentation of capabilities by legitimate users.

A second distinction is that, as described above, a capability allows a restricted impersonation of the grantor, not direct access to the named object. This means that one can revoke a capability by changing the access rights available to the grantor of the capability. Such a change would affect all capabilities that had been issued by that grantor (as well as any copies), but not those that had been issued by others. If the only principal with *a priori* access to an object is its owner, this distinction disappears as there can be only one original grantor.

A final distinction, as implemented on most authentication systems, is that the resulting capability would have an expiration time. This is a feature. If a non-expiring capability is desired, the expiration time can be set sufficiently far in the future.

3.2 An authorization server

An authorization server implemented using restricted proxies does not directly specify that a particular principal is authorized to use a particular service or access a particular object. Instead, when requested by an authorized client, the authorization server grants a restricted proxy allowing the authorized client (the grantee) to act as the authorization server for the purpose of asserting the client's rights to access particular objects. The restrictions in the proxy (in this case a list of authorized actions) are determined by consulting the authorization server's database or applying other suitable heuristics.



1. Authenticated authorization request (operation X)
2. [operation X only]R, {Kproxy}Ksession
3. [operation X only]R, authentication using Kproxy

Figure 3: The authorization protocol

Figure 3 shows the messages involved when client **C** uses authorization server **R** for authorization to end-server **S**. The solid lines represent messages in the authorization protocol. The initial request for authorization is authenticated using the underlying authentication protocol. The authorization credentials (a restricted proxy) returned in 2 consist of a certificate and a proxy key. The proxy key is returned protected from disclosure by encrypting it under the session key exchanged during authentication with **R** (encryption is represented by curly braces {}). To use the proxy, the client presents the proxy to the end-server, partaking in an authentication exchange as described in Section 2. Message 0, the dashed line in the figure, represents *a priori* knowledge about the authorization credentials needed for server **S**. This information might be specified as part of the application protocol, retrieved from a name server, or obtained from the end-server directly.

An end-server wishing to use the services of an authorization server would grant full or the maximum desired access to the authorization server (this is described in detail in Section 3.5).

3.3 A group server

A group server implemented using restricted proxies grants proxies that delegate the right to assert membership in a particular group. The protocol is the same as that for the authorization server in figure 3; the authorized operation is the assertion of group membership.

Certificate: $[restrictions1, K_{proxy1}]_{grantor}$
Certificate: $[restrictions2, K_{proxy2}]K_{proxy1}$
Certificate: $[restrictions3, K_{proxy3}]K_{proxy2}$
Proxy-key: K_{proxy3}

Figure 4: Cascaded proxies

A group server might maintain more than one group. The name of a group as asserted by the group server is unique only for a particular group server (or a small set of servers). As such, a global name of a group is composed of the name of the group server, and the name of the group on that server.

It should be possible for the name of a group to appear in authorization databases anywhere that the name of any other principal might appear. This might be on the end-server, or in an authorization server, or even on another group server. An end-server wishing to use a group server would include the name of a group in its authorization database. A client would obtain a group proxy from the group server and send it to the end-server when requesting an operation. The end-server would verify the authenticity of the proxy and the identity of the client, and if valid perform the operation.

If the end-server's authorization database is maintained by an authorization server, then the client would present the group proxy to the authorization server, and if all checks out, the authorization server would return an authorization proxy to be used by the client as described in the previous subsection.

3.4 Cascaded authorization

In a paper on cascaded authentication [11], Sollins proposed a method to pass authorization from party to party when a task involves cascaded operations by parties that do not completely trust one another. A similar mechanism is supported more efficiently by restricted proxies.

By its definition, a proxy allows one principal to perform an operation on behalf of another. An *intermediate server* that has been granted a bearer proxy can pass that proxy to a *subordinate server* (the next server in the pipeline) with additional restrictions applied. Restrictions are added by signing a new proxy with the proxy key from the original proxy. The new proxy specifies any additional restrictions and a new proxy key. The certificates from both proxies are provided to the subordinate server, but only the proxy key from the final proxy in the chain is provided. Figure 4 shows a chain of proxies that might be provided to a subordinate server.

Cascaded authorization is a little different for delegate proxies. To pass a delegate proxy to a subordinate, an intermediate server provides the subordinate with the certificate from the original proxy. Because the intermediate server is explicitly named in the original proxy, it also grants the subordinate a new proxy allowing the subordinate to act as the intermediate server for the purpose of executing the original proxy. Instead of signing the new proxy with the proxy key from the original proxy, it is signed directly by the intermediate server. An important difference between the two approaches to cascaded authorization is that the use of a delegate proxy leaves an audit trail since the new proxy identifies the intermediate server.

A distinct difference between the cascaded authentication approach described by Sollins and the approach described here is that in Sollins's approach the end-server has to contact the authentication server to verify the authenticity of a chain of proxies.

3.5 Access-control-lists and capabilities

By basing authorization on the proxy model, application servers can easily combine the benefits of access-control-lists and capability-based authorization mechanisms. Application servers would be designed to base authorization on a local access-control-list. Where a capability-based approach is required, the access-control-list would contain a single entry naming the principal (perhaps the server itself) authorized to grant capabilities for server operations.

Similarly, when appropriate to hand off the authorization function to a centrally maintained authorization or group server, the name of the authorization or group server would be added to the local access-control-list. In fact, if local autonomy is desired, local users might appear directly in the access-control-list together with the name of an authorization server to which the function of authorizing remote users has been assigned.

Since the same access-control-list abstraction should be used on the authorization servers as on other servers, access-control-list entries can support an associated list of restrictions. On an authorization server, the restrictions field of a matching access-control-list entry can be copied to the restrictions field of the resulting proxy. These would be in addition to restrictions transferred from any proxies presented to the authorization server or those imposed by the server itself.

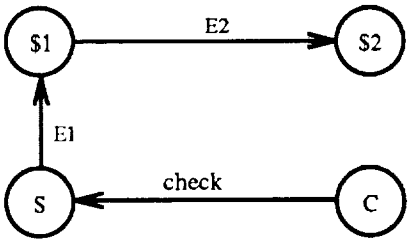
Finally, by supporting compound principal identifiers in access-control-list entries, it becomes possible to require the concurrence of multiple principals for certain operations. Among other things, this functionality allows one to specify the need for both user and host credentials for certain operations as well as the separation of privilege so that a single user can't act alone. Proxy-based authorization allows a user to obtain proxies from more than one grantor for a particular operation, providing the mechanism by which the user would assert such concurrence.

4 Accounting

Section 3 showed how restricted proxies support a wide range of authorization mechanisms. Accounting is closely tied to authorization; in fact, the two are interdependent. Authorization depends on accounting when a server verifies that a client has been allocated sufficient resources (e.g, quota) to perform an operation. Conversely, accounting depends on authorization to control the transfer of resources from one account to another.

In our design, accounts are maintained on accounting servers. At a minimum, each account contains a unique name, an access-control-list, and a collection of records, each record specifying a currency and a balance. Accounting servers support multiple currencies, either monetary (dollars, pounds, or yen) or resource specific (disk blocks, cpu cycles, or printer pages). Quotas are implemented by transferring funds of the appropriate currency out of an account when the resource is allocated and transferring the funds back when the resource is released. Accounts are identified as the composition of the principal identifier for the accounting server and the name of the account on the server. It is possible to transfer resources from an account on one server to one on another.

The transfer of resources can be accomplished through two distinct mechanisms. The simplest mechanism is used when no guarantee is required that sufficient resources exist. A principal authorized to debit an account (the payor) issues a numbered delegate proxy (a check) authorizing the payee to transfer funds from the payor's account to that of the payee. This check limits the resources that can be transferred, and the payee transfers up to that limit. If the payor uses a different accounting server than the payee, the payee grants its own accounting server a cascaded proxy (endorsement) for the check allowing the accounting server to collect the resources on its behalf. Subsequent accounting servers repeat the process until the payor's accounting server is reached. Once a check is paid, the accounting server keeps track of the check number until the expiration time on the check. If, within that period, another check with the same number is seen, it is rejected.



check: [ckno,amount,S]C
E1: [ckno,amount,S]C [dep ckno to $1]S
E2: [ckno,amount,S]C [dep ckno to $1]S [dep ckno to $2]$1

Figure 5: Processing a check

Figure 5 shows the messages involved in issuing and clearing such a check. In the figure, accounting servers are labeled by **$**s. The first message represents a check signed by **C** drawn on **C**'s accounting server **$2** made payable to server **S**. Upon completion of **C**'s request, **S** endorses the check and deposits it with its accounting server in message E1. The endorsement is a restricted proxy that will be used for cascaded authorization. A restricted endorsement (e.g. for deposit only) is a delegate proxy, an unrestricted endorsement is a bearer proxy.

In this case, **C** and **S** do not share the same accounting server, so **$1** marks the resources added to S's account as uncollected, adds its own endorsement and forwards the check to **$2** in message E2. If necessary, such endorsements can be repeated until the check reaches the client's accounting server, but in this case only one additional step is necessary. This distributed method for accounting requires out-of-band mechanisms to deal with checks returned for insufficient resources, or because they are forged or misdrawn, but the same is true in the real world.

The second approach for transferring resources is used when a server requires a guarantee that sufficient resources have been allocated to the client, as is often the case when maintaining quotas. The approach is analogous to that of a certified check. The client draws a check and provides the details (the check number, the party to be paid, and the amount) to the accounting server. The accounting server places a hold on the resources and returns an authorization proxy to the client certifying that the client has sufficient resources to cover the check. The client presents the authorization proxy and the check to the end-server along with its application request.

Once the requested operation is performed, the end-server negotiates the check as described earlier. When the check reaches the client's accounting server, the accounting server looks for the check in its list of outstanding certified checks, and if found, makes the transfer. Cashier's checks are also easily supported by this accounting model; the details are left as an exercise for the reader.

5 Related work

This section describes other work that has been done on authorization and accounting for distributed systems. Some of the earliest work in the area is found in Grapevine [2] where end-servers query registration servers to determine whether a client is a member of a particular group. A similar approach is employed in Sun's Yellow Pages where centrally maintained files such as /etc/group are consulted for authorization purposes. In both approaches, the authorization decision remains with the local system. With the distributed authorization and group services supported by restricted proxies, the authorization decision can be delegated to a remote server.

There have been several proposals concerning forwarding and delegation of authentication in distributed systems. Karger [6] proposed a server that keeps track of special passwords that are established when a user logs in. These passwords are passed to other systems which act on the user's behalf for operations that require the cascaded use of multiple servers. This scheme is not encryption-based, but relies on secure channels for passing the special passwords. These channels might be implemented on top of an end-to-end encryption mechanism.

A mechanism that comes close to restricted proxies is the cascaded authentication mechanism described by Sollins [11] in which restrictions can be added as credentials are passed from system to system. The differences between Sollins' approach and proxy-based cascaded authorization was described in Section 3.4.

The proxy model described by this paper was designed for use in Version 5 of the Kerberos authentication system. Support for proxies was first included in the Kerberos protocol specification in mid 1989 [7]. At about the same time, another mechanisms for delegation was developed as part of the Digital Distributed System Security Architecture [4, 5]. In the DSSA, principals generate and sign delegation certificates to allow intermediate systems to act on their behalf. An important difference is that in the DSSA, restrictions are supported only by creating separate principals, called roles, and by generating a delegation certificate for one of the roles instead of for the original principal. The delegation then supports only access specifically authorized for that role. The creation of a new role is cumbersome when delegating on the fly

Certificate: $\{restrictions, K_{proxy}\}K^{-1}_{grantor}$

Proxy-key: K^{-1}_{proxy}

Figure 6: A public-key restricted proxy

or when granting access to individual objects. Roles can not be used to implement the authorization server described in Section 3.2.

Functionality similar to that of the authorization and group services of Sections 3.2 and 3.3 has been proposed as part of the European Computer Manufacturers Association standard for security in open systems [1]. The ECMA standard defines Privilege Attributed Certificates (PACs) signed by an authority and certifying that the bearer or a named principal possess certain privileges.

Work is underway for the Open Software Foundation's Distributed Computing Environment that uses restricted proxies as supported by Kerberos to pass authorization information. In particular, they have implemented a privilege attribute server that signs certificates asserting a principal's unique identifier and a set of user groups to which the principal belongs. Plans are in place to extend their mechanism to support delegation [3].

Surprisingly little attention has been paid to the issue of accounting in distributed systems. Sentry [9] lays the groundwork for accounting by describing a mechanism that would be co-located with an authentication and authorization server. Although they share a common mechanism, it seems apparent now that there is little to be gained by requiring all three services to be co-located. Like the accounting mechanism described here, Sentry pointed out the need to support multiple currencies.

Amoeba [8] supports a distributed bank server identical in purpose to the accounting server based on restricted proxies. The protocol used by Amoeba's bank server is significantly different, however. In Amoeba, a client must contact the bank and transfer funds into the server's account before it contacts the server. The server will then provide services until the pre-paid funds have been exhausted. Like the mechanism described here, Amoeba supports multiple currencies.

6 Integration with existing systems

It is straightforward to implement restricted proxies using encryption-based authentication mechanisms based on either public-key or conventional cryptography. This section shows how proxies can be implemented with either approach and describes the specific details of their support in Version 5 of the Kerberos authentication system.

6.1 Public-key cryptography

The certificate for a public-key proxy contains a proxy key generated by the grantor, the expiration time of the proxy, and the restrictions imposed its use. The proxy key embedded in the proxy certificate is a public key from a public/private key pair. The proxy key provided to the grantee is the other key from that pair. All fields are signed by encrypting them with the grantor's private key. Figure 6 shows a proxy generated in this manner. The signed proxy is additionally tagged with the name of the grantor to enable those needing to verify the proxy to select the correct key.

If the authentication system is purely public-key, a public-key digital signature algorithm can be used in place of the encryption system and the encryption step would be replaced by the sealing of the certificate with a cryptographic checksum. If a hybrid authentication system is used, where subsequent keys are from a conventional cryptosystem, then the proxy key is a conventional key generated by the grantor and the proxy key must be additionally encrypted in the public key of the end-server to protect it from disclosure.

The proxy is returned to the grantee. When the grantee presents the proxy to an end-server, the end-server decrypts the proxy using the public key of the grantor (obtained from an authentication/name server), verifies the authenticity of the proxy, accepts additional authentication from the grantee (either personal authentication for a delegate proxy or proof that it knows the proxy key for a bearer proxy), checks the restrictions, and if all checks out, performs the requested operation.

6.2 Restricted proxies in Kerberos

A proxy implemented using an authentication system based on conventional cryptography is identical to that in figure 6 except that the proxy is accompanied by credentials authenticating the grantor to the end-server. The proxy certificate is encrypted using the session key generated by an authentication server, the session key also having been earlier sealed in the credentials. The proxy key is a secret key generated by the grantor. This key is both sealed in the proxy certificate and securely passed to the grantee. The remainder of this section describes the integration of restricted proxies with Kerberos [12], an authentication system based on conventional cryptography developed as part of MIT's Project Athena.

Kerberos credentials are issued by an authentication server and presented by a client to prove its identity to a particular end-server. Credentials consist of two parts: a ticket, and a session key. The ticket contains the name of the authenticated principal and a session key. It is encrypted using the secret key shared by the end-server and the Kerberos server. The session key is never sent across the network in the clear. The session key is returned to the client encrypted in the session key shared by the client and the Kerberos server.

To prove its identity, a client sends the ticket to the end-server along with an authenticator which has been encrypted using the session key. The authenticator proves that the client actually possesses the session key included in the ticket. Without this step an attacker would be able to reuse a ticket that it obtained by eavesdropping on an earlier exchange.

Kerberos has been in use at MIT since Fall of 1986, and it has been used elsewhere since then. Version 5 of Kerberos [7] is the first major revision of the protocol since its original release and contains several new features important for the practical support of restricted proxies. The inclusion of explicit support for proxies in Version 5 makes their use more transparent to applications which have already been modified to use Kerberos.

The Version 5 ticket and authenticator each have a new field called authorization-data. This field consists of an arbitrary number of typed sub-fields, each of which places restrictions on the use of the ticket. The Kerberos protocol does not specify how the sub-fields are to be interpreted except to stress that restrictions must be additive. Each subfield places additional restrictions on the use of credentials, never removing restrictions or granting additional privileges.

When tickets are requested, the requesting principal can specify that restrictions be placed on their use. When new tickets are issued based on existing credentials, restrictions may be added, but not removed. To add restrictions to an existing ticket, a client generates an authenticator specifying a proxy key in the subkey field and specifying additional restrictions in the authorization-data field. The ticket and authenticator are treated as the new proxy and provided with the new proxy key to the grantee. Once obtained, the grantee can use such a proxy the same way it uses any other credentials issued by the authentication system.

6.3 Discussion

Supporting proxies within an authentication mechanism has several advantages. Transparency is one advantage; a second is that the initial authentication of a user can itself be thought of as the granting of a proxy and restrictions can be placed on the credentials based on the characteristics of the initial exchange with the authentication server.

A disadvantage of using conventional cryptography to implement proxies is that each proxy can be used at only a particular end-server. This is offset by implementing proxies within Kerberos itself since it is possible to issue a proxy for the Kerberos "ticket-granting" service. Such a proxy allows the grantee to obtain proxies with identical restrictions for additional end-servers as needed.

7 Common restrictions

The restrictions field of a proxy should be interpreted as a collection of typed subfields, each type corresponding to a different restriction. This section describes several of the more useful restrictions and some that demonstrate the flexibility of the model. Additional restrictions are described in [10]. Neither should be construed as a complete list.

7.1 Grantee

This restriction specifies a list of principals authorized to use a proxy and the number of principals from the list needed to exercise the proxy (usually one). To use such a proxy a principal must present the authentication credentials of a named grantee, or an additional proxy granted by a named grantee, to the end-server along with the proxy. If the **grantee** restriction is missing, the proxy is a bearer proxy and may be used by anyone possessing it. To exercise a bearer proxy the bearer must take part in an authentication exchange proving possession of the proxy key thus preventing an attacker from using a proxy obtained by eavesdropping on the network.

7.2 For-use-by-group

The **for-use-by-group** restriction specifies the list of groups authorized to use a proxy and the number of groups from the list required. To use such a proxy, the bearer presents the proxy along with additional proxies from appropriate group servers. One way to implement separation of privilege is to require assertion of membership in multiple groups with disjoint members.

7.3 Issued-for

The **issued-for** restriction specifies a list of servers authorized to accept the proxy. This restriction is important for public-key proxies which are otherwise verifiable by and exercisable on all servers.

7.4 Quota

The **quota** restriction specifies a currency and a limit. It limits the quantity of a resource that can be consumed or obtained. It will most often be found in a proxy issued by an accounting server.

7.5 Authorized

The **authorized** restriction specifies a complete list of those objects which may be accessed using the rights granted by a proxy and optionally a list of operations that may be performed on each object. This restriction usually appears in proxies used as capabilities. It also appears in proxies returned by an authorization server. There are no constraints on the form of the object names or the list of operations other than that the grantor and the end-server must agree. These fields are to be interpreted by the end-server.

7.6 Group-membership

This restriction specifies that the grantee is a member of only the listed groups. It would be included in a proxy issued by a group server to limit the groups to which one is a member. Without this restriction, the grantee would be considered a member of all groups maintained by the group server granting the proxy.

7.7 Accept-once

The **accept-once** restriction tells an end-server that it is only to accept a proxy one time. This restriction takes an identifier as an argument. Any subsequent proxy from the same grantor bearing the same identifier and received by the end-server within the expiration time of the first proxy is rejected. A real life example of such an identifier is a check number.

7.8 Limit-restriction

Restrictions that are defined only for particular end-servers are sometimes needed. If a proxy can be used on a server to which some restrictions do not apply, those restrictions must be associated with the name of the server to which they do apply. This is accomplished with the **limit-restriction** restriction which takes a list of servers and a list of other restrictions. The restrictions embedded within this restriction will be enforced by the named servers and ignored by others.

7.9 The propagation of restrictions

Authentication, authorization, and group servers accept proxies and issue proxies. If a proxy is issued based upon a proxy that includes restrictions, those restrictions should be passed on to the proxy to be issued. If a restriction is limited (see **limit-restriction**) then the restriction may be left out if it can be guaranteed that the proxy to be issued, and any proxies that might later be derived from it, can not be used for any of the servers listed in the limited restriction.

8 Status

A beta release of Kerberos Version 5 is available. The release includes support for restricted proxies. Information on the Kerberos release is available from **info-kerberos@mit.edu**. Authorization and accounting services built with restricted proxies are being developed at the Information Sciences Institute of the University of Southern California.

9 Discussion and conclusions

The problems of authentication, authorization, and accounting are closely related. By subtly changing the way one thinks about the problems, the similarities become apparent. By extending an authentication system to support restricted proxies, it becomes possible to support flexible distributed authorization and accounting mechanisms. The proxy model strikes a balance between access-control-list and capability-based mechanisms allowing each to be used where appropriate and allowing their use in combination.

This paper has shown how restricted proxies can be supported using existing authentication systems and how they are used for authorization and accounting. The resulting mechanisms scale and appear natural when compared with their analogues in society.

Acknowledgments

I would like to thank Celeste Anderson, Steven Augart, Steve Bellovin, Deborah Estrin, David Keppel, John Kohl, Ed Lazowska, Joe Pato, Karen Sollins, Bill Sommerfeld, Stuart Stubblebine, and Prasad Upasani for discussions of restricted proxies and comments on drafts of this paper.

References

[1] European Computer Manufacturers Association. Security in open systems: Data elements and service definitions, December 1989. Standard ECMA-138.

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

[3] Marlena E. Erdos and Joseph N. Pato. Extending the OSF DCE authorization system to support practical delegation. In *Proceedings of the PSRG Workshop on Network and Distributed System Security*, pages 93–100, February 1993.

[4] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The Digital distributed system security architecture. In *Proceedings of the 1989 National Computer Security Conference*, pages 305–319, October 1989.

[5] M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 20–30, May 1990.

[6] Paul A. Karger. Authentication and discretionary access control in computer networks. *Computer Networks and ISDN Systems*, 10(1):27–37, 1985.

[7] John T. Kohl and B. Clifford Neuman. The Kerberos network authentication service: Version 5 draft protocol specification. August 1989. Revised November 1989, October 1990, December 1990, June 1991, September 1992, April 1993.

[8] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4):289–299, 1986.

[9] B. Clifford Neuman. Sentry: A discretionary access control server. Bachelor's Thesis, Massachusetts Institute of Technology, June 1985.

[10] B. Clifford Neuman. Proxy-based authorization and accounting for distributed systems. Technical Report 91-02-01, Department of Computer Science and Engineering, University of Washington, March 1991.

[11] Karen R. Sollins. Cascaded authentication. In *Proceedings of the 1988 IEEE Symposium on Research in Security and Privacy*, pages 156–163, April 1988.

[12] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 Usenix Conference*, pages 191–201, February 1988.

This research was supported in part by the National Science Foundation (Grant No. CCR-8619663), the Washington Technology Centers, Digital Equipment Corporation, and the Defense Advance Research Projects Agency under NASA Cooperative Agreement NCC-2-539. The views and conclusions contained in this paper are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any of the funding agencies. The author may be reached at USC/ISI, 4676 Admiralty Way, Marina del Rey, CA 90292-6695, USA. Telephone +1 (310) 822-1511, email bcn@isi.edu.