Security in Process Migration Systems

Javier Echaiz and Jorge Ardenghi

Laboratorio de Investigación en Sistemas Distribuidos (LISiDi)
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur – Bahía Blanca, Argentina
T.E.: +54 291– 4595135 Fax: +54 291– 4595136
{je,jra}@cs.uns.edu.ar

ABSTRACT

A loosely coupled distributed system is composed by nodes, usually heterogenous, connected by a network. These systems have enormous aggregate computing potential. However most of this potential is not realized unless the underlying software is able to implement the concept of *single system image* (SSI) on the physically distributed system. This way the resources belonging to a node could be accessed transparently from any other node.
This paper discusses the issues of a process migration protocol as an essential component of a distributed system and its extension to the grid computing paradigm. The security issues are specially considered.

Keywords: computer security, clustering, grid computing, load sharing, process migration, distributed systems.

1. INTRODUCTION

During the last two decades the development of low cost powerful microprocessors and high speed computer networks have promoted a change in the computing paradigm. Yesterday big mainframes have been replaced by clusters of small but powerful computers connected by high speed data networks. The end user, instead of working on a dumb terminal connected by a serial line to the mainframe computer, now has a powerful computer at his/her desk that is connected by a network to a large number of other computers. Potentially, all resources that physically reside on any computer on the network are available for use by the user. However, this potential cannot be fully utilized unless the users are able to *transparently* access these resources. By transparency, we mean that the users should be able to access any resource without worrying about (and indeed, without being aware of) its physical location. This is what a distributed operating system does. The goal of such a system is to create the illusion in the minds of the users of a single timesharing system, rather than a collection of independent but connected machines. Such a system would therefore place all the resources of all the computers in the distributed system at the disposal of its users without burdening them with the need to be aware of the details of the distribution. Some of the most outstanding systems are Amoeba [1], Sprite [2], MOSIX [3], Condor [4], etc.. Most such systems work by running a copy of the same operating system on all the participating computers and these copies cooperate to provide a *single system image* (SSI) of the system to its users [5].

Even when these systems look very attractive its popularity has not really grown. This is because of the fact that there is a large existing user and software base for Unix. Several distributed systems therefore try to emulate Unix so that existing applications can be reused with little or no modifications and so that the users get a familiar working environment. However, there is another problem that must be solved by today's distributed system software. This problem arises because the distributed systems of today typically comprise of hardware and operating system software from a variety of vendors. Achieving the single system image in the face of this heterogeneity is a major challenge.

Systems such as Amoeba, etc., do not address the problem of operating system heterogeneity as they assume that all participating machines on the network run the same operating system. There also are solutions available today that address this problem partially. SunNFS [6] and AFS [7] are examples of distributed file systems that are commonly used now to achieve a unified view of the file system on a network of workstations. Various other resources such as printers etc., are now routinely shared on Unix-like systems.

An important resource that is typically not shared transparently on Unix-like systems is the CPU. Several studies have shown that there is a wide disparity in the load of various machines in a distributed system at any given time of the day. While there are some machines that are heavily loaded, others are completely

Supported by a fellowship of the "Consejo Nacional de Investigaciones Científicas y Técnicas", Argentina.

idle. What is desirable is that these machines should share the total processing load requirement of all the users so that the load distribution is more uniform and ultimately the users see an improvement in system performance. While there are user level commands available in Unix-like systems (e.g. `rsh`) that allow users to execute their jobs on any machine of their choice, such mechanisms are clearly not transparent. In an ideal setting, a user would just fire a job and the system would automatically select the best location (i.e., , the least loaded machine) to execute the job. A stricter form of such load sharing is called *load balancing*, wherein the system strives to balance the load on all machines at all times. While load sharing would just require the system to select the best machine for executing a newly submitted job and transparently transfer the job to that machine, load balancing would typically call for migrating a job to another machine, possibly during its execution. These two forms of process migration are usually known as *nonpreemptive* (also known as remote execution) and *preemptive process migration* respectively.

It can be easily seen that nonpreemptive process migration can be more easily implemented than preemptive migration. This is because the latter requires the system to checkpoint the state of a process already in execution and then transfer this state to the target machine. Clearly this is a very hard problem if the two machines are architecturally different, for in that case the checkpointing would have to be done at the source program level and not the executable code level. Tui [8] is an example of a system that allows executing processes to migrate to architecturally different machines. However, in this case, the process of migration is not transparent to the application program, and the application must cooperate with the migration software in order to migrate successfully. Further, the migration process is also costlier in terms of time since the entire state (which might be quite large) has to be transferred to the destination machine. Studies have shown that this additional overhead severely restricts the performance benefit that can be obtained by using preemptive task transfer as compared to nonpreemptive process migration [9]. Nonpreemptive migration does not incur this additional cost since only newly submitted jobs are transferred to other machines, and therefore there is no address space image to transfer. Further, heterogeneity is much more easily accommodated. There are two orthogonal issues related to load sharing. The first one relates to the policies for migration. For example, when should a machine attempt to transfer a process to another machine, which process should be migrated and to which machine? The second issue relates to the mechanisms for transferring processes and with ensuring that a migrated job will get roughly the same environment as it would have on the machine where it actually originated.

The rest of this paper is organized as follows. The next section describes the policies and migration mechanisms of a load sharing system. Section 3 presents the security problems at the cluster level. Section 4 analyzes the issues of extending a process migration system to the grid paradigm. Finally, Section 5 presents the conclusions and future work.

2. POLICIES AND MIGRATION MECHANISMS

Scheduling of tasks in a load sharing distributed system involves deciding not only when to execute a process, but also where to execute it. Accordingly, scheduling in a distributed system is accomplished by two components: the *allocator* and the *scheduler*. The allocator decides where a job will execute and the scheduler decides when a job gets its share of the CPU. Typically, each node in distributed system has its own scheduler to schedule processes on the local processor, usually in some timeshared way, whereas the higher level decisions of assigning a process to a node is carried out by the allocator. Although there are slight variations [10], this scheme seems to be most natural in distributed systems. The reason behind this is twofold. First, each node usually has its own operating system which is capable of scheduling processes. The second reason is modularity, the designers can concentrate more on the relatively complicated load distribution issues without being burdened by every single detail of scheduling.

The allocator and the scheduler implement the allocation and scheduling policies respectively. The allocation policy tries to distribute the overall load of the system to its individual nodes by transferring processes among nodes. Scheduling policy simply checks the set of *runnable* processes available to a node and chooses to run the most suitable process to maximize the overall throughput. The process scheduling policy of any traditional operating system might work as a scheduling policy, even though in some systems special technique can also be used, e.g. coscheduling [10]. In reality, scheduling is mostly done by the local operating system itself. Hence, we should specially concentrate on allocation policy. The allocation policy does not actually perform process migration. Rather, it helps the mechanism for process migration by providing information such as when a node should attempt to migrate a process out, which process is suitable for transfer, to which node a selected process can be sent, etc. Policy modules of different nodes also exchange load information to get the current load information of different nodes in the system to make intelligent decisions.

Unix process management semantics are not easily amenable to extending over a distributed environment. This is because of the inherent assumption of a single processor in the system design, and more importantly the tight coupling of the process subsystem with the other subsystems in the typical Unix-like implementations [11].

3. SECURITY IN PROCESS MIGRATION SYSTEMS

This section describes the CLEX mechanism of authentication [12]. Subsection 3.2 presents improvements to this basic security scheme, general solutions that can be possibly applied to others process migration systems.

3.1. CLEX security

3.1.1. Authentication

The authentication applied on a client request is simply implemented by using a mechanism of privileged ports. The remote execution client sends a request using an UDP socket bound to a privileged port. In every current Unix-like operating system the ports between 1 and 1023 constitute the *privileged ports*. This way only the Super User (*root*) is able to use these ports using the `bind()` call included in the BSD sockets library. The server monitors the port number included as part of the messages and discards them if they are not routed to the right port. This method avoids most of the accidents provoked by an unexperienced user (without root permissions). Since only root is able to use privileged ports[1], non root processes are given temporary permissions invoking the `setuid(0)` system call while they are issuing a `bind` to the privileged port. The *remote execution servers* accept requests at the port 300[2]. Authentication of a client node is also accomplished by the server. The header of a request package includes the NID (*Node Id*) of the client node. The server looks for this NID in its database to assure this node can be part of the cluster and it is correctly configured (Subsection 3.1.3). If this is not the case then the request is discarded. Besides it verifies that the NID matches with the IP address for that particular node.

3.1.2. File system

The current system does not include cryptographic mechanisms to store the information locally since it assumes a safe execution environment.
Moreover, the CLEX protocol assumes the presence of NFS [6]. The Network File System allows to preserve an uniform view of the file system at the cluster level. A very important fact of NFS is its high availability; it is present in virtually every Unix-like operating system, essential feature to make our protocol (easily) portable to different architectures and operating systems. Besides, employing an already existing distributed file system instead of designing a new one shortens the implementation time.

3.1.3. Nodes database

The protocol assumes that the set of nodes part of the load sharing system is formed statically and that every node "knows" the rest of the members of the system. Every node has a list containing all the nodes (IP addresses) of the cluster. This list is simply called the *access list*.
Actually the protocol does not support dynamic configuration of the cluster. This limitation eases the implementation and at the same time adds some security by hindering the incorporation of hostile nodes to the cluster. Anyway it isn't too hard to include an hostile node as part of the cluster, since there are not secure authentication mechanisms. The intruder just needs to replace (to impersonate) a node from the access list, e.g. provoking a DoS (*Denial of Service*) and taking its IP address.

3.1.4. Messages

The CLEX protocol defines a set of messages through which the nodes in a cluster communicate with each other to achieve clusterwide process migration. The messages have different names, contents and these are used for specific purposes in the protocol. There are three type of messages: *request*, *reply*, and *asynchronous*.
A node sends a request message to another when it wants the latter to do some operation for it. Receiving a request, a node carries out a well defined operation. The receiver then sends back a reply message containing the result of the operation. Asynchronous messages are sent to another node to notify an event of interest to the latter. These messages are not replied to. The operation carried out by a node after receiving a request or notification could be idempotent, meaning that effects of doing this operation once is equivalent to doing it more that once. Idempotency has major impact in the fault tolerance operations. The general description of a message also includes the potential receiver and sender of the message, instances when a sender sends the message, the receiver action after receiving the message, behavior of the receiver and sender in case of failure.

- *Request and reply packet formats*

The formats of the request and reply packets are shown in Figure 1. The first five fields of the packages constitute the header.
The *Node ID* identifies the sending node. The server at the receiving host validates the Node Id by looking it up in the access list. The *Age Number* is used

[1]Even in the first Unix-like operating systems this problem was common, so at the end of the '70s the system call `setuid()` was introduced in System V (version 7).

[2]The port 300 was picked simply because it is not associated to any standard service.

Total Length	Node Id	Age Number	Sequence Number	Msg. Id	... Msg. Data

Header of the request messages.

Total Length	Node Id	Age Number	Sequence Number	Msg. Id	Error Code	... Msg. Data

Header of the reply messages.

Fig. 1. Header of the request and reply messages.

to preserve the uniqueness of the process ids in the face of a fault. The *Sequence Number* is used for error checking, e.g. detection of duplicated messages. The *Msg Id* identifies the particular request. In order to assembly the reply the server copies the header of the request package into the reply header. The client, after receiving the reply, validates it by comparing the *Node Id*, *Age Number* and *Sequence Number* in it with those sent in the request packet. The result of the requested operation is placed in the *Error Code* field of the reply packet by the server.
In the reply packet, if the error code is `-PROCESS_MIGRATED`, then the next bytes contain the Node Id of the new node of the remote process involved in the operation. If, while serving some request involving a remote process, the server does not find the process on its node, then it checks whether it is the originating node of the process. If this is the case, then it searches the process's entry in the *process reference table* and if it finds the entry, then it reads the Node Id of the current node of the process from it and returns the `-PROCESS_MIGRATED` error along with this Node Id back to the client. The client, on receiving this error, redirects the remote call to the new Node Id of the process. This new Node Id field is not shown in the message specific reply format, but implicitly exists if the error code is `-PROCESS_MIGRATED`.

3.2. Improving CLEX security

The first topic we have to understand when trying to address the security problems in an SSI system is trust relationships. An SSI blurs the traditional boundaries between ”inside and ”outside. Inside ends up being the relationship between all nodes in the CLEX kernels list, and outside is everything else. If it is considered to extend the system to include nodes geographically dispersed, then the inside becomes a complex set of trust relationships that must be established in an unsafe environment.

3.2.1. Authentication

The authentication scheme presented above is far from being secure, everyone having root credentials is able to break the system. It can be considered secure enough only under very well controlled environments.
Conventional technologies, for instance the technologies employed for e-commerce, are designed to guarantee security in both ends, client and server. Though, in a typical cluster, this mechanism necessarily becomes complex, since a node is usually client and server at the same time (*peer*). The authentication process of our migration process protocol could be extended using a more powerful authentication scheme, e.g. Kerberos [13, 14] or Public Key Infrastructure (PKI/X509.3 certificates). Even though certainly both alternatives would improve the system security, it would introduce a single point of failures. Now the system depends on the proper functioning of this new component, the *authentication server*.

3.2.2. File system

Even though NFS is very simple regarding to structure, it assumes a very strong confidence model: the client trusts the remote file server. This is a dangerous assumption in real world, since the server (or anybody with root access) can obtain the files locally, or by sniffing in a LAN[3]. Another problem with NFS is user id spoofing because most of the permission validation actions are made by the kernel at the client side. So it's clear the necessity of a secure distributed file system replacing NFS. For instance we can rely on SFS (*The Self-Certifying File System*) [15] or CryptosFS [16], both introduce cryptographic techniques to the distributed file system. Other reasonable alternatives are AFS (or OpenAFS) and Coda. This way, using the authentication, encryption, and access control mechanisms provided by the distributed file systems the in-transit information is safe from interception, modification, and message injection attacks.
On the other hand if simplicity and availability of NFS is a good enough reason to preserve it we can configure NFS over ssh (sacrificing speed). This has the extra benefit of working with NAT, unlike IPSec.

3.2.3. Nodes database

Certainly the only way to obtain a scalable and dynamic system is through a service able to add new nodes to the cluster in real time. To accomplish this task we need to implement a dynamic database (*access list*). This decision carries the need of improving security, since we need to avoid hostile nodes to be integrated to the cluster.
This new scheme can be implemented dividing the database into two parts. A static part keeps a list of IP addresses just like before, being these nodes considered *implicitly trusted*. On the other hand we have a dynamic list in order to allow the addition of nodes dynamically. This list can be implemented by a *daemon* that employing well known cryptographic and digital signature algorithms[4] permit secure authen-

[3]For example using Ethereal, a GPL-licensed software available for Unix-like and Microsoft Windows platforms.

[4]For example Diffie-Hellman or MQV (Menezes-Qu-Vanstone) [17, 18] for key exchange.

tication of peer nodes, avoiding man in the middle (MiM) attacks.
Depending on the security of the surrounding environment the nodes will compose the static or the dynamic part of the list. In the extreme case of a secure and controlled environment every node will integrate the static database (identical to the original protocol), while in a not secure one every node will be added to the system exchanging keys during the INIC message[5]. Obviously there are hybrid compositions in cases not so extreme.

3.2.4. Communication confidentiality and integrity

The original protocol sends messages without any kind of protection facing potential confidentiality and integrity breaks, meaning it is possible to intercept, modify, and inject messages.
Possibly the easiest way to solve these problems is by adding security to the IP level. IPSec [19] supports various mechanisms of authentication and encryption, and using the 'Encapsulated Security Payload' (ESP) mode it can encrypt all the traffic in layer 3. In this case we can implement a VPN to include every participating node, ensuring the boundaries between "inside and "outside of the system.

4. EXTENDING THE CLUSTER

A cluster designed to follow the above characteristics will provide an increased computer power (exploiting idle processors) and will be able to balance the load of the system, smoothing the activity peaks.
Other interesting possibilities of the process migration in a cluster are related to a better exploitation of the storage space, fault tolerance, access locality to the data, better system administration, and mobile computing [20].
The system described in [21] provides nonpreemptive process migration for heterogeneous clusters Unix-like. This heterogeneity means not only different configurations, it also includes different architectures and operating systems. These capabilities provided by our system can be extended to a *grid computing* environment.
Various authors [22, 23, 24] have tried to precise a grid definition. In fact, the concept of grid computing is still evolving and most attempts to define it precisely end up excluding implementations that many would consider to be grids. In this sense we won't try to precise this concept, we will use this term to refer to a geographically distributed system where every component is a cluster instead of a node.
Figure 2 shows a possible example of use of this system. Let's suppose Cluster A presents an overloaded status index, while the rest of the clusters are idle. In this case A will migrate jobs in order to balance the global resources, absorbing load peaks in some region of the system. Even when this example is very simple it is clear that complexity growths considerably when we scale our initial cluster to a grid environment.

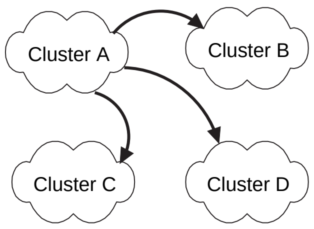

Fig. 2. Jobs are migrated to less busy parts of the system to balance resource loads.

At first sight this technologies can be seen as able to take any application and run it a 1000 times faster without the need for buying any more machines or software. This is not the case since not every application is suitable or enabled to be parallelized. Moreover, some kinds of applications can take a large amount of work to modify them to achieve faster throughput. The configuration of the system can greatly affect the performance, reliability, and security of an organizations computing infrastructure.

4.1. Requirements

In order to extend such a cluster with geographic distribution certain design characteristics are needed:

Administrative hierarchy. An administrative hierarchy able to manage this new and complex environment is needed. It determines how administrative information flows through the nodes of the system.

Communications. In the CLEX original protocol UDP was the chosen protocol for communication among processes distributed within the cluster. This model is not valid now, it will be mandatory to include QoS parameters such as latency, bandwidth, reliability, fault-tolerance, and jitter control.

Information services. A grid is a dynamic environment where the location and types of services available are constantly changing. A major goal is to make all resources accessible to any process in the system, without regard to the relative location of the resource needed. It is necessary to provide mechanisms to enable a rich environment in which information is readily obtained by requesting services. The grid information (registration and directory) services components provide the mechanisms for registering and obtaining information about the grid structure, resources, services, and status.

[5]The first action made by a node after booting is to send an INIC message to the rest of the nodes in the list. This operation is part of the fault tolerance mechanism.

Naming service. In a grid, like in any distributed system, names are used to refer to a wide variety of objects such as computers, services, or data objects. The naming service provides a uniform name space across the complete grid environment. Typical naming services are provided by the international X.500 naming scheme or DNS.

Distributed file systems. Most distributed applications require access to files distributed among many servers. A distributed file system is therefore a key component in a distributed system. From an applications point of view it is important that a distributed file system can provide a uniform global namespace, support a range of file I/O protocols, require little or no program modification, and provide means that enable performance optimizations to be implemented, such as the usage of caches.

Security. Any distributed system involves all four aspects of security: confidentiality, integrity, authentication, and accountability. Security within a grid environment is a complex issue requiring diverse resources autonomously administered to interact in a manner that does not impact the usability of the resources or introduces security holes/lapses in individual systems or the environments as a whole. Security infrastructure constitutes the focus of this work.

Fault tolerance. To provide a reliable and robust environment it is important that a means of monitoring resources and applications is provided. To accomplish this task, tools that monitor resources and application need to be deployed.

Resource management. The management of processor time, memory, network, storage, and other components in this new environment is clearly very important. The overall aim is to efficiently and effectively schedule the applications that need to utilize the available resources in the grid computing environment. From a users point of view, resource management and scheduling should be transparent; their interaction with it being confined to a manipulating mechanism for submitting their application. It is important in a grid that a resource management and scheduling service can interact with those that may be installed locally.

Computational economy. As a grid is constructed by coupling resources distributed across various organizations and administrative domains that may be owned by different organizations, it is essential to support mechanisms and policies that help in regulate resource supply and demand.

User and administrative GUI. The interfaces to the services and resources available should be intuitive and easy to use[6]. In addition, they should work on a range of different platforms and operating systems. They also need to take advantage of Web technologies to offer a view of portal supercomputing. The Web-centric approach to access supercomputing resources should enable users to access any resource from anywhere over any platform at any time.

5. CONCLUSIONS AND FUTURE WORK

The cluster environment is usually considered to be immerse in a safe environment. If this is not the case it should be needed to implement a secure system over the hostile environment. In order to accomplish this goal it is important to introduce secure underlaying protocols, like the ones presented in Subsection 3.2. So a logical approach is to study different alternatives able to isolate the impact on the performance and heterogeneity of the system, since the use of cryptography methods imply additional processing. Another problem is portability, perhaps not every protocol is available in every employed platform.

After the verification and optimization process of the load sharing protocol[7] we shall analyze the modifications and extensions needed for a grid environment. For instance, we'll consider the scheduling and load sharing, network latencies, security, and a new fault tolerance model.

Clearly *grid computing* technologies are still walking their first steps but according to the web growth, this computing paradigm goes in evident ascent.

Finally, without question one of the most promising lines of work in process migration is the use of mobile agents on the web. In this sense, instead of the workstation and the pool processors models, the computers are connected as interfaces to the model "the network is the computer", in this case transparency losing relevance. Performance now dependes on the network latency and thus the transfer of the process state is not so important, compared to the LAN case. If the execution environment is safe, users will gradually allow the execution of remote process on their computers. Another important topic to consider is heterogeneity, automatically supported at the language level. In this sense, mobile agents on the web solve impediments analogous to those in process migration, allowing the massive use of both technologies, since they share similar mechanisms.

[6]In this sense our research group is working on developing graphic tools jointly with the "Laboratorio de Visualización y Computación Gráfica" at our University.

[7]Besides a formal verification of the protocol it will be important to have at least two implementations on different operating systems in order to check correctness and heterogeneity.

6. REFERENCES

[1] S. J. Mullender, C. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Stavern, *Amoeba: a distributed operating system for the 1990s.*, IEEE Computer, May 1990.

[2] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software Practice & Experience*, 1991.

[3] Amnon Barak and Oren La'adan, "The MOSIX multicomputer operating system for high performance cluster computing," *Future Generation Computer Systems*, vol. 13, no. 4–5, pp. 361–372, 1998.

[4] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The condor experience," 2004.

[5] Javier Echaiz and Jorge Ardenghi, "Single System Image: Pilar de los Sistemas de Clustering," *V Workshop de Investigadores en Ciencias de la Computación, WICC 2003*, pp. 210–214, May 2003.

[6] Inc. Sun Microsystems, "RFC 1094: NFS: Network File System Protocol specification," Mar. 1989.

[7] James H. Morris, Mahadev Satyanarayanan, Conner Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith, "Andrew: A distributed personal computing environment," *Communications of the ACM*, vol. 29, no. 3, pp. 184–201, Mar. 1986.

[8] Peter Smith and Norman C. Hutchinson, "Heterogeneous Process Migration: The Tui System," Tech. Rep. TR-96-04, Department of Computer Science, University of British Columbia, Feb. 1996.

[9] Derek L. Eager, Edward D. Lazowska, and John Zahorjan, "The Limited Performance Benefits of Migrating Active Processes for Load Sharing," *Conf. on Measurement & Modelling of Comp. Syst., ACM SIGMETRICS*, pp. 63–72, May 1988.

[10] John K. Ousterhout, "Scheduling Techniques for Concurrent Systems," in *Third International Conference on Distributed Computing Systems*, May 1982, pp. 22–30.

[11] Ken Shirriff, "Building Distributed Process Management on an Object-Oriented Framework," in *Proceedings of the USENIX Annual Technical Conference (USENIX-97)*, Berkeley, Jan. 1997, pp. 119–132, Usenix Association.

[12] Javier Echaiz, "Migración de Procesos en Sistemas Heterogéneos," Magister en Ciencias de la Computación, Universidad Nacional del Sur, Apr 2005.

[13] B. Clifford Neuman, J. G. Steiner, and J. I. Schiller, "Kerberos: An authentication service for open network systems," in *Winter 1988 USENIX Conference*, Dallas, TX, 1988, USENIX Association, pp. 191–201.

[14] J. Kohl and C. Neuman, "RFC 1510: The Kerberos Network Authentication Service (V5)," Sept. 1993.

[15] Frans Kaashoek, "Self-certifying File System," Dec. 2000.

[16] Declan Patrick O'shanahan, "CryptosFS: Fast cryptographic secure NFS," Nov. 2000.

[17] Burton S. Kaliski Jr., "IEEE P1363: A Standard for RSA, Diffie-Hellman, and Elliptic-Curve Cryptography (Abstract)," in *Security Protocols Workshop*, 1996, pp. 117–118.

[18] T. Mark A. Lomas, Ed., *Security Protocols, International Workshop, Cambridge, United Kingdom, April 10-12, 1996, Proceedings*, vol. 1189 of *Lecture Notes in Computer Science*. Springer, 1997.

[19] J. Lee, "A survey on IPSec Key Management Protocols," 1997.

[20] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou, "Process migration," *ACM Computing Surveys*, vol. 32, no. 3, pp. 241–299, 2000.

[21] Javier Echaiz and Jorge Ardenghi, "Carga Compartida en Sistemas Distribuidos Heterogéneos," *V Workshop de Investigadores en Ciencias de la Computación, WICC 2004*, pp. 549–553, May 2004.

[22] Geoffrey Fox and Dennis Gannon, "Grid computing: Computational grids," *Computing in Science and Engineering*, vol. 3, no. 4, pp. 74–86, July/Aug. 2001.

[23] Ewa Deelman and Carl Kesselman, "Grid computing," *Scientific Programming*, vol. 10, no. 2, pp. 101–102, 2002.

[24] And Domenico Laforenza, Mark Baker, and Rajkumar Buyya, "Grids and grid technologies for wide-area distributed computing," July 2002.

[25] Mary Thompson, "Security implications of typical grid computing scenarios," Dec. 2000.