Three Design Patterns for Secure Distributed Systems

Alan H. Karp, Kevin Smathers
Intelligent Enterprise Technologies Laboratory
HP Laboratories Palo Alto

HPL-2003-40
February 25th , 2003*

distributed computing, security

The computers we use are not secure, and they are even less so when connected to the Internet. A lot of blame has been put on lazy sysadmins for not applying patches promptly, but the fault is not entirely theirs. We believe that distributed systems should be designed to make attacks harder and to limit the damage done when attacks succeed. We propose three components of the system architecture that address these goals and make distributed systems easier to monitor and manage, while simplifying the task of writing secure applications. Following these guidelines won't make the system secure, but doing so will make it easier to build systems that are.

Three Design Patterns for Secure Distributed Systems

Alan H. Karp, Kevin Smathers

Hewlett-Packard Laboratories

Palo Alto, CA

Abstract

The computers we use are not secure, and they are even less so when connected to the Internet. A lot of blame has been put on lazy sysadmins for not applying patches promptly, but the fault is not entirely theirs. We believe that distributed systems should be designed to make attacks harder and to limit the damage done when attacks succeed. We propose three components of the system architecture that address these goals and make distributed systems easier to monitor and manage, while simplifying the task of writing secure applications. Following these guidelines won't make the system secure, but doing so will make it easier to build systems that are.

Introduction

Much of our thinking on the subject of computer security originated in the 1960s and 1970s [1, 2], a time when computers were rarely networked, those that were connected largely trusted each other, and those connections rarely changed. That is not the situation today. Virtually all computers and an increasing number of other devices are connected to the Internet. Many of the machines out there are running software that attempts to harm others, either because their owners are malicious or because their owners are careless and have allowed malicious people take control of their machines. To make matters worse, the environment is constantly changing, with machines joining and leaving the system and even changing owners.

In this paper we'll examine three ways to structure distributed systems to make them less susceptible to attack. The patterns we describe address certain aspects of anonymity, auditing, and access control. They can limit the damage done by some denial of service attacks and can make some attacks against applications more difficult.

There are many aspects of security that we don't address in this paper. We exclude from the discussion physical security, including tamper-proof devices, methods for authentication, specification of authorization policy, most aspects of privacy, and protocols for non-repudiation. Nothing we propose can mitigate attacks against the underlying operating system, including the network stack. Also, problems of misplaced trust, social engineering, and careless users are beyond the scope of this work.

Assumptions and Implications

The changes in the environment brought about by widespread connectivity mean that builders of distributed systems need to re-examine their assumptions when designing infrastructures for the Internet. The work presented here is based on the following assumptions and implications.

Large number of machines and users: Because of the large number of machines, we can't rely on a centralized repository for identity management. Because of the time it takes changes to propagate through such a large system, we can't rely on global data consistency to manage privileges, particularly revocation of privileges. The problems of identity management are exacerbated because there are so many users.

Dynamic: Applications are very difficult to write if the underlying system is always changing. Protocols change faster than applications are upgraded, and mismatches can be exploited to attack machines. Connections to services are frequently lost. Relying on application code to reconnect to a service, or find an equivalent service is dangerous. Applications can be run by users with limited privileges, but security decisions often need to be made in a context with more rights. Careless users can expose the entire machine, and even other machines, to attack. To the greatest possible extent we need to shield the applications from changes in its environment.

Heterogeneous: The environment is heterogeneous in device capabilities as well as in machine type and operating system. Some devices may have limited computation ability, communications bandwidth, or storage, which means we need to choose our security mechanisms carefully. An Internet wrist watch may not have sufficient storage to maintain an extensive set of certificates; an Internet microwave oven may not have the bandwidth to complete a key exchange in a timely manner. Also, different applications will be based on different interpretations of the protocols. A security upgrade should not be delayed pending a comparable upgrade in all the clients.

Hostile: We know that malicious people are trying to circumvent security, and we know that our mechanisms are not perfect. That means some attacks will succeed, and we need to design systems that will limit the damage that can be done when they do. Relying on applications to protect themselves from hackers multiplies the work and the opportunities for error. The system needs to shield applications from hackers as much as possible.

Different Environments: We need to recognize that there are different ways to achieve an end. Security mechanisms that work in the enterprise may be inappropriate in a small business or in a home. Our system architecture must be flexible enough to support a wide variety of requirements. It must also allow collaboration when the security policies and mechanisms of the parties differ.

In what follows we'll present three patterns for connecting the components of distributed systems and explain how they can be used to enhance security. We'll next describe how these ideas were used in a commercial system.

Separate Granting of Rights from Access Control

Most systems today use coordinated mechanisms for granting rights to processes and deciding which requests to honor. For example, most services accessed via web pages authenticate by user name and password. The service then uses an access control list (ACL) to decide which user requests to honor. As we all know, the problem for the user is the handling of all the user names and passwords. There is a corresponding problem on the server side dealing with updates to the ACLs as well as forgotten passwords and user IDs.

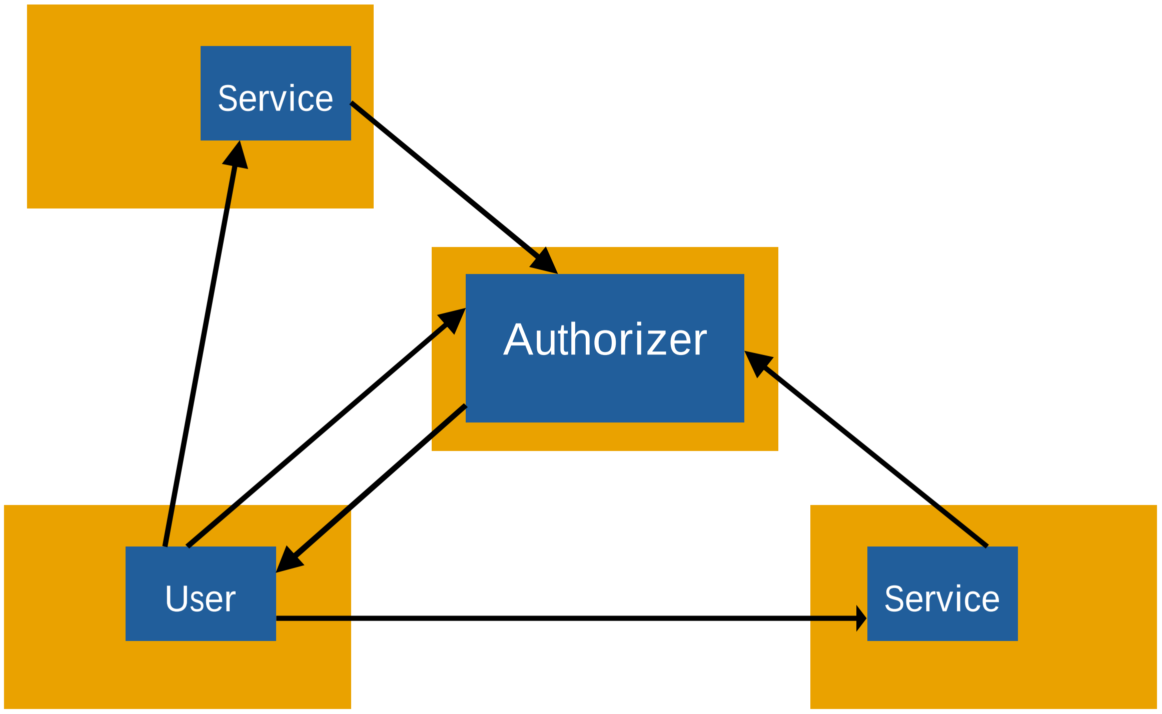

Figure 1. Separating authorization from access control.

There is no reason why the mechanism used to grant privileges to a person or a process should be related to the mechanism used to decide whether a particular request should be honored. In our first pattern, illustrated in Figure 1, we assume there is an “Authorizer” component. The authorizer can be an external service, as shown, or another application running with the service.

The service registers a set of rules with the authorizer, each associated with some credential denoting the allowed permissions. For example, a rule might be “Anyone submitting an employee certificate signed by HP may read this file.” The credential could be as simple as the string “Read”, digitally signed by the service to prevent forgery. Another, independently developed service could have the same rule, but could use a SPKI capability certificate as a credential. A third could use an identity representing the role “HP employee.”

Someone wishing to use these services would first authenticate with the authorizer and receive a set of credentials. In our example, a user presenting a properly signed certificate would receive a signed document containing the string “Read.” The service now only needs to whether the credential submitted corresponds to the request. It doesn’t need to know who the user is or how the user got the credential. For example, HP could change its policy to grant employee status to contractors without requiring any changes to the services. In addition, the authorizer has no need to interpret the permission, allowing it to handle any kind of service, even one invented after the authorizer started running. Note that the user's identity need not be revealed to the service, providing a measure of anonymity.

Using this pattern simplifies the code needed to grant the rights, because it doesn't need to deal with the interface of the service. The code that decides whether to honor the request is also simpler, because it can be independent of any authentication or authorization decision. This approach also addresses the issue of a single sign on for a family of services, but it does not solve the problem of dealing with several authorization services.

This approach is not entirely new. When a Kerberos [3] ticket is issued to a user who has presented certain credentials, that ticket can be used to access some resource. The access control decision is based on the ticket, not the authentication used to get the ticket. However, in Kerberos and similar systems the form of the ticket is part of the architecture. We propose that the ticket be specific to the service. After all, the access decision may depend on some context not considered when the ticket granting mechanism was designed. Kerberos ties the permission to an authentication; we propose making this feature a matter of policy, not architecture.

Mediate between Application and User

Jini [4] is representative of the way most networked services work. A user performs a look up in a JavaSpace and receives a handle to the service. The user then uses this handle to access the service. Most IETF defined Internet services, such as FTP, NTP, and NFS, work the same way. The problem is where to put authentication, authorization, and access control and where to generate the audit trail. For example, Jini services are expected to be devices, such as printers; we certainly don't want to put that much of a burden on them. Jini uses access control lists in the JavaSpace to control which groups of services can be discovered by a user, but there is no way to revoke access without involving the service.

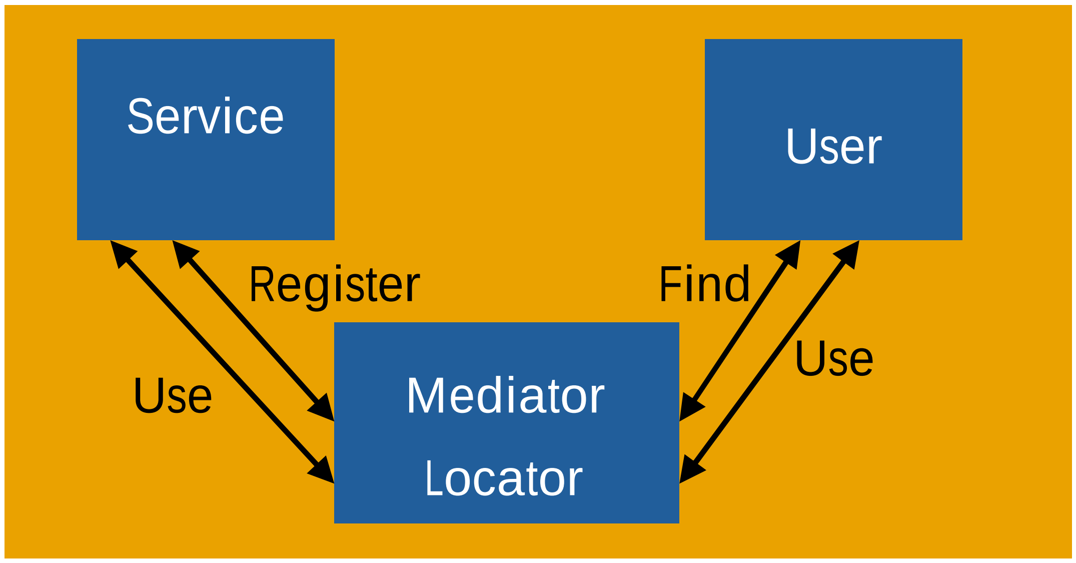

Figure 2. Mediating between client and service.

Figure 2 shows our second pattern, adding a level of indirection between the user and the application. (Note that the mediator and locator can be separate processes and need not even be on the same machine.) The analogy is with an operating system. The user of a file doesn't talk directly to the file system drivers; the requests pass through the operating system kernel. This mediator provides several useful functions. It can produce audit logs or generate usage events needed to manage the system effectively. The mediator can also translate between high-level policies, perhaps expressed in an access control list, into low-level mechanisms, such as allowing access to a particular file. The mediator also acts as a trusted third party, supplying a verifiable identity for both parties. If there is an inconsistency in the versions of the protocols, the mediator can translate where possible. Sending requests through an intermediary provides a measure of anonymity for both the service and the user.

The alternative of putting these functions into libraries to be linked with the application executable doesn't work as well. First of all, there is no way to be sure that every application will use the libraries properly. A seemingly trivial error in using a cryptographic library can make the system vulnerable. Also, using libraries adds the problem of security upgrades and bug fixes to the list of patches that need to be managed. Another problem is that there is no way to prevent the applications from bypassing the code that generates the necessary log entries and management events. The absence of a mediator also makes dispute handling and identity management issues that need to be addressed in every application.

There are existing systems that provide intermediaries. We've already mentioned operating systems, but application servers act as intermediaries for back-end services. They provide a dual role here. They shield the backend services from the open Internet and deal with large numbers of users, something most applications don't do well. However, the application server is not involved when internal users access the services, which means separate mechanisms must be used for the two classes of users. The CORBA ORB [5] is logically an intermediary, but implementations usually make the ORB a library in the client and service address spaces.

Use a Proxy for Remote Users

Most services accessed over the web today talk directly to the user. That's not a problem as long as the user is well behaved, but there is a problem when the user is malicious. In the worst case, the user can induce the service to run code injected by the user. Substantial harm can result since the service almost certainly has more privileges than a remote user. Cutting off attacks and revoking privileges must be handled separately by each application, tasks that some of them will get wrong.

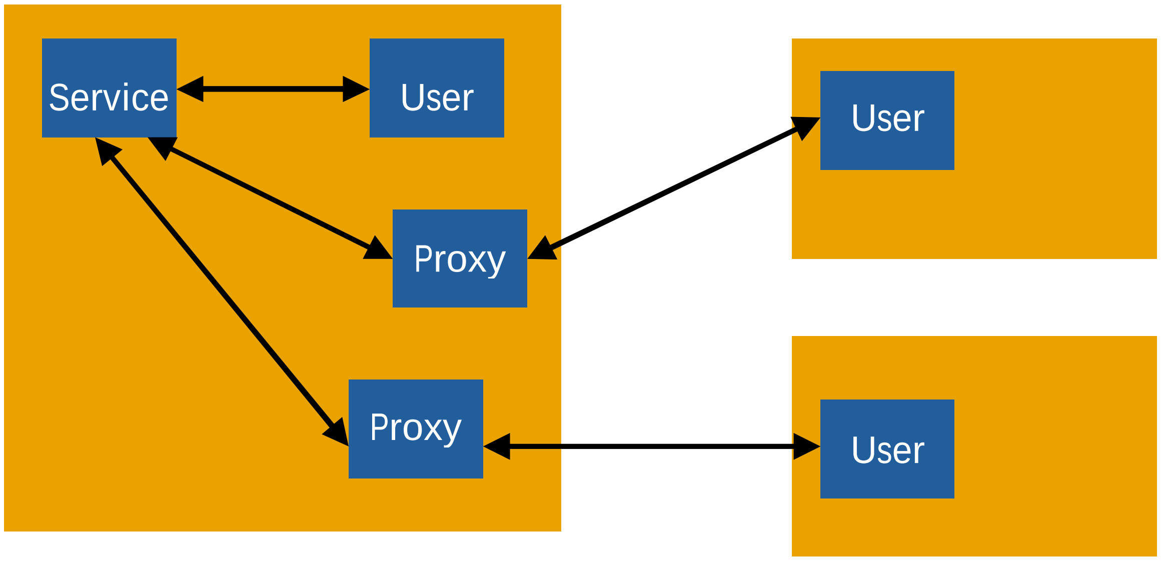

Figure 3. Using proxies for remote users.

When your service needs to talk to a user on another machine, do it through separate process that acts as a proxy, as shown in Figure 3. There are a number of benefits. First of all, the proxy can enforce any policies set up for the machine. After all, you wouldn't want a careless user to inadvertently send confidential material to an insecure machine. In addition, the proxy can mitigate the damage done by some attacks. If the attacker crashes the process in which the proxy is running, the only effect is to cut off the attacker's access to your machine. If you detect an attack against or through the proxy, you only need to kill the process running the proxy to cut off the attacker.

It may not be practical to have a separate process for each remote user. However, using a single proxy for several users means that one of them can deny service to the others or gain their privileges by subverting the proxy. Whether to share a proxy among users and which groups of users should share a single proxy is a question of balancing risk against cost. This design pattern makes such decisions one of policy, not architecture.

In some operating systems, the proxy can run with a set of privileges appropriate for the remote user instead of the local one. For example, in Unix we can `chroot` the proxy to a directory with a limited set of files. In this case, the proxy can be extremely accommodating, trying to do everything it is asked to do. Since its permissions are controlled by the machine on which it is running, it won't be able to do anything that the remote user hasn't been granted permission to do. Privileges can be revoked by removing them from the proxy's set of permissions.

In other situations, the proxy can be made smarter, filtering requests that might cause problems if forwarded to the application. The advantage of putting these controls in the proxy is that they can protect applications written with no consideration for remote users, and updates to the proxy add protections to all applications on the machine.

Other systems have such proxies. For example, SMTP [6] specifies an intermediary. Unfortunately, the SMTP service typically runs as root, which loses much of the security benefit of a proxy that runs with the remote user's privileges. A web server acting as a front-end for an application, such as a corporate database, can also be thought of as a proxy. However, this proxy acts on behalf of all remote users as a matter of architecture, not policy. Not only can a single malicious user deny service to all others by crashing the server, but the server necessarily has at least the union of the privileges of all the users and has even more.

There is one place where proxies are used for their security properties, proxy servers frequently used on corporate networks. The intent here is to shield the client from the service. We believe that proxies should be used on both ends.

Example of Use

These three patterns were used in e-speak [7,8], an open source product offered by Hewlett-Packard from 1999 until 2002. An e-speak environment consists of a set of *logical machines*. Each logical machine has an active component, called the *core*, and a passive component called the *repository*. Users and providers of e-speak services are called *clients* of the core.

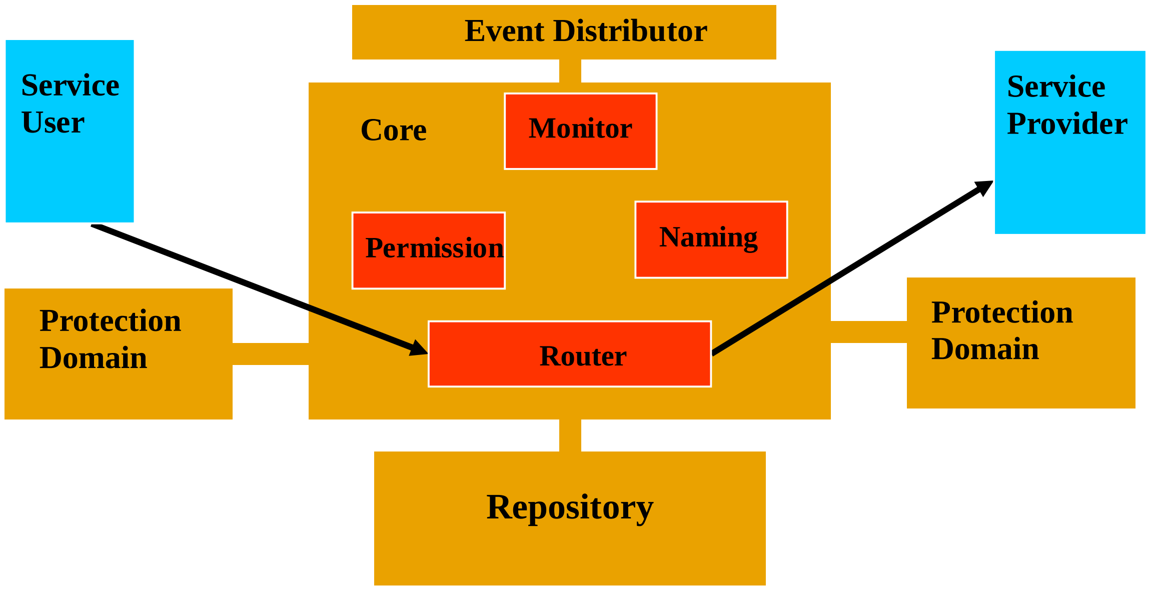

Figure 4. E-speak Logical Machine with clients.

Clients interact by sending messages naming resources to the core. The core uses information in the sending client's protection domain to resolve the name, determine the handler for the named resource, extract the permissions corresponding to the request, and forwards the request to the resource handler. The core also generates the appropriate usage events.

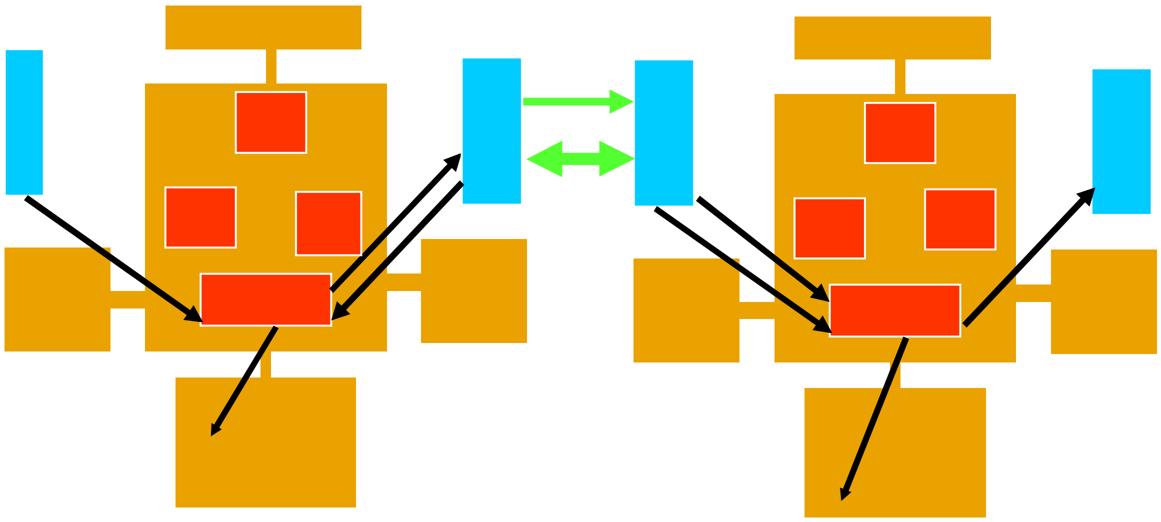

Figure 5. Using a remote service in e-speak.

Figure 5 shows the interaction between two logical machines. When two machines want to communicate, they each start a proxy to act on behalf of users on the other machine. The proxies handle all issues of mutual authentication, protocol negotiation, and link encryption. A local policy is used to determine which resources are to be made available to users on the other machine, and these resources are registered on the other machine listing the proxy as the handler.

When a client names a remote resource in a request, the core forwards the request and permissions to the handler, the proxy in this case. The proxy sends the request to its counterpart on the machine that exported the resource, which repeats the request. This request is then forwarded to the handler. Note that both cores, the requester, and the handler do exactly what they do for local requests. Only the proxies know anything about another machine.

Although no one is aware of a serious assault on the security of the system, it is also true that no one has reported a security breach in any of the existing applications. Since these include applications involving pride (HiTel), money, (mTicka), and proprietary information (SpinCircuit), situations where there is an incentive to find a flaw, there is a chance that the patterns described here helped make the system more secure.

References

1. J. H. Saltzer, "Protection and the Control of Information Sharing in Multics", *Proc. IEEE* **63**, #9, September 1974
2. J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations", *Comm. ACM* **9**, #3, March 1966
3. B. C. Neuman, and T. Ts'o, "Kerberos: An Authentication Service for Computer Networks", *IEEE Communications,* **32**, #9, September 1994
4. K. Arnold, B. O'Sullivan, R. W. Sheifler, J. Waldo, and A. Wollrath, *The Jini Specfication*, Addison-Wesley, 1999
5. R. Hoque, *CORBA 3*, IDG Books, 1998
6. J. Postel, "Simple Mail Transfer Protocol", IETF RFC 0821, 1982
7. N. Apte and T. Mehta, *Web Services: A Java Develoer's Guide Using E-speak,* Prentice-Hall PTR 2002
8. A. Karp, G. Rozas, A. Banerji, R. Gupta, "The Client Utility Architecture: The Precursor to E-speak", HP Labs Technical Report HPL-2001-136, June 2001