Mesh: Secure, Lightweight Grid Middleware Using Existing SSH Infrastructure*

Paul Z. Kolano
NASA Advanced Supercomputing Division, NASA Ames Research Center
M/S 258-6, Moffett Field, CA 94035 U.S.A.
kolano@nas.nasa.gov

ABSTRACT

Grid computing promises gains in effective computational power, resource utilization, and resource accessibility, but in order to achieve these gains, organizations must deploy grid middleware that, in most cases, does not adhere to fundamental security principles. This paper introduces a new lightweight grid middleware called Mesh, which is based on the addition of a single sign-on capability to the built-in public key authentication mechanism of SSH using system call interposition. The initial Mesh implementation is compatible with approximately 90% of the world's SSH servers and any SSH client that supports public key authentication. Resources may be added to a Mesh-based grid in a matter of minutes using just five small files and two environment variable settings. Mesh adheres to fundamental security principles and was designed to be compatible with strong security mechanisms including two-factor authentication, SSH bastions, and restrictive firewalls. Mesh uses a remote command model, which is based on the syntax and commands already understood by users, thus requires no additional knowledge to utilize effectively. Several existing services have been integrated with Mesh to provide resource discovery and query, high performance file transfer, and job management.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Privacy Protection–*access controls, authentication*; C.2.4 [**Communication/Networking and Information Technology**]: Distributed Systems–client/server, distributed applications

General Terms

Security

Keywords

SSH, access control, authentication, authorization, delegation, distributed systems, grids, middleware, security, single sign-on

*This work is supported by the NASA Advanced Supercomputing Division under Task Order NNA05AC20T (Contract GS-09F-00282) with Advanced Management Technology Inc.

SACMAT'07, June 20-22, 2007, Sophia Antipolis, France.

1. INTRODUCTION

Grid computing [7] aims to connect large numbers of geographically and organizationally distributed resources to increase computational power, resource utilization, and resource accessibility. Two fundamental capabilities that grids must provide [8] are *single sign-on*, where the same user identity and credentials may be used for authentication across all grid resources, and *delegation*, where users may empower systems and services to authenticate and perform operations on their behalf across the grid. These capabilities support advanced models such as services that automatically select the most desirable resources for user jobs and then submit the jobs to those resources without user intervention. Single sign-on and delegation are implemented within grid middleware that runs as a network daemon on all grid resources to provide common services for the grid.

The same capabilities that allow users to easily access many different resources also provide attackers with the means for rapid, widespread compromise. With single sign-on, compromised user credentials give attackers instant access to every grid resource. With delegation, credentials become more vulnerable to misuse as they are accessed by third party systems and services to perform unsupervised operations. Finally, the grid middleware that provides these capabilities is itself potentially subject to remote exploits on every grid resource. Thus, before any grid middleware can be deployed in security-conscious organizations, it must be backed with assurance that it has been designed and implemented to minimize both the opportunity for compromise as well as the damage possible from compromise.

One basis for such assurance is adherence to Saltzer and Schroeder's fundamental security design principles [28]. Unfortunately, existing grid middleware implementations do not adhere to these principles, putting organizations that deploy them at risk. For example, three of the most popular middleware implementations are Globus [6], Condor [18], and UNICORE [5]. Globus is the most widely deployed grid middleware and is the reference implementation for many proposed grid standards. The Grid Security Infrastructure (GSI), which is the authentication component of Globus, consists of over 100 MB of source code. Each Globus service handles its own authorization, with the relevant code for each scattered throughout the remaining 450 MB of the source distribution. This volume of code makes detailed inspection impossible and breaks *economy of mechanism*, in which designs should be kept as simple and small as possible. The sheer complexity of the system and its voluminous documentation create a steep learning curve for all aspects of installation, configuration, maintenance, and usage, thereby breaking *psychological acceptability*, in which the human interface should be designed for ease of use.

Condor's source code is not publicly available, thus lacks an

open design that is subject to inspection. Its default security model uses host-based authentication, thereby breaking *separation of privilege* as the compromise of one host allows the compromise of all accounts on all other hosts without any additional user credentials. Neither Condor nor UNICORE offer any fine-grained authorization mechanism, thus lack the ability to specify a *least privilege* policy in which only the exact operations required are allowed. None of the three provides any *separation of privilege* between the ability to perform grid operations and the ability to modify files that affect non-grid logins such as shell startup files.

This paper presents a new secure, lightweight grid middleware called Mesh, **M**iddleware using **E**xisting **S**SH **H**osts. Mesh was designed to provide simple and easy to use grid capabilities within a strong security architecture based on fundamental security principles. The foundation of Mesh is SSH, which has many desirable properties as the basis for grid middleware. It is the reference remote login mechanism on all Unix systems, providing a vast infrastructure on which grids may be built. It has client support on most platforms and APIs in most languages. It has built-in support for scripted operations through the use of public key authentication and for credential delegation and renewal through SSH agents. All communication is secured by encryption with both protocols and implementations subject to rigorous scrutiny. The additional goals of Mesh were to provide:

- *Single sign-on*. Users must have the capability to use the same identity and credentials to access any Mesh resource for which they are authorized.
- *Minimal installation and configuration on existing hosts*. Enabling Mesh capabilities on any host already running an SSH server should be a trivial procedure.
- *Compatibility with existing security models and infrastructures*. Mesh should not impose any constraints on users when they are not using Mesh services. Mesh should be compatible with strong authentication mechanisms such as two-factor authentication.
- *Time and scope limited credentials*. Operations performed with Mesh credentials should be limited to those authorized by site, host, and user policies for a specific time frame.

All services in Mesh are accessed using SSH remote commands authenticated based on standard SSH private keys loaded into an SSH agent. Remote commands are based on the syntax, commands, and models already understood by users, thus require no additional knowledge to utilize effectively. Several existing services have been integrated with Mesh to provide a complete grid environment with support for resource discovery and query, high performance file transfer, and job management. Mesh is currently deployed across multiple NASA supercomputing centers to provide secure unattended remote operations in a single sign-on environment.

This paper is organized as follows. Section 2 describes related work. Section 3 gives an overview of the Mesh architecture and its usage. Section 4 discusses the addition of single sign-on to SSH. Section 5 presents Mesh authorization. Section 6 describes the services that have been integrated into Mesh. Section 7 presents Mesh performance results. Finally, Section 8 discusses conclusions and future work.

2. RELATED WORK

There are a variety of efforts related to the problem addressed by this paper. A single sign-on capability has been integrated into SSH using various authentication methods including Kerberos, the Grid Security Infrastructure (GSI), and the UNICORE X.509 infrastructure. In Kerberos [14], users authenticate to the Key Distribution Center (KDC) for a given realm using a password to obtain a ticket for authenticating to a Ticket-Granting Server (TGS). This ticket is then used to obtain additional limited-lifetime tickets for authenticating to remote servers. Users can obtain tickets to servers in other realms through trust relationships between KDCs. GSI-Enabled SSH [10] relies on certificate-based authentication provided by the GSI component of Globus and is enabled using a specially-patched version of OpenSSH. UNICORE's SSH functionality [25] is based on manipulation of the authorized_keys file returned to the SSH server. The authorized_keys file is generated dynamically by the UNICORE client when an SSH connection is requested by the user and then written to the user's ~/.ssh directory using the UNICORE server on the target host, which does not allow users to maintain their own authorized_keys settings.

All of these approaches have similar drawbacks. They all require special clients beyond stock SSH to retrieve tickets/certificates and use them for authentication. They all have a significant code base and rely on secondary protocols in addition to SSH for security, thus increasing the potential for security vulnerabilities. They all have single points of attack (i.e. the KDC of Kerberos and the certificate authorities for GSI and UNICORE), which, if compromised, allow access to all user accounts within the realm/VO. Finally, none of them restrict user capabilities after authentication to the SSH server, thus compromised user credentials allow an attacker to arbitrarily modify the user's account on every system they have access to.

Several projects provide restrictive environments for SSH commands in which only file transfers are allowed. McCullough describes an approach [19] based on SSH forced commands in which file transfers are only allowed to a specific file path on the destination system. The Rssh [26] and Scponly [30] projects restrict SSH functionality by providing a custom user shell that only permits SCP commands. In addition, both shells use chroot() to limit the user's access to the file system. These projects are all geared towards file transfers, however, thus do not allow the user to be limited to an arbitrary set of commands.

There are a number of sandboxing approaches to limit the operations that an application may perform. Identity boxing [33] extends traditional Unix discretionary access control by providing a mechanism to label subjects and objects in the system with arbitrary strings instead of the finite space of numerical UIDs. Processes run inside an identity box with a given label and can only access files permitted by ACL for that label even though the actual UID for the process may allow such access. Systrace [24] provides a constrained execution environment where an application is only allowed to invoke system calls permitted by a configurable policy. Systrace provides an interactive tool to assist in generating policies for specific applications.

A variety of grid middleware projects exist. Globus [6] is based on a web service architecture and uses a certificate-based approach for single sign-on. While Globus is full-featured, it has significant software requirements on both clients and Globus-enabled hosts. Globus supports fine-grained authorization through the Community Authorization Service (CAS) [22], which issues non-standard proxy certificates that define the permissions that users have on grid resources. The only service in which CAS support has been implemented, however, is the GridFTP file transfer service. Globus also supports the Virtual Organization Membership System (VOMS) [1], which issues standard proxy certificates that associate a set of attributes with the user, which can then be used by individual grid

services as they see fit. While users can request proxies with limited subsets of attributes, to do so they must understand which attributes each service supports, how the service uses those attributes, and which attributes are required for a given operation.

The Uniform Interface to Computing Resources (UNICORE) [5] is middleware that can be installed on hosts using only Perl, but still requires a certificate infrastructure and a specialized client. Condor [18] is a grid environment whose goal is to increase utilization of existing resources by farming out jobs to idle workstations. Native Condor security is based on weak host-based authentication, but can be strengthened using compatible external security frameworks such as Kerberos and GSI. Increasing its security in this manner, however, requires significant software installations and specialized clients.

GROWL [11] hides the details of grid access behind an abstraction layer. Applications use the GROWL client library with APIs in several languages to request operations from a GROWL server. The GROWL server then processes these requests using the appropriate requests to the underlying grid. Grid client software is only needed on the GROWL server while GROWL clients only require the lightweight GROWL client library. This approach, however, does not decrease the complexity of grid server deployment.

WSRF::Lite [20] is a lightweight Perl implementation of the Web Services Resource Framework (WSRF) that provides a simple, easy to install hosting environment for WSRF-compliant web services. WSRF::Lite provides mutual authentication using X.509 certificates, but requires the installation of a special client. GridSite [4] allows users to authenticate to websites using X.509 certificates within a standard browser, after which they are allowed edit and upload web content according to a site security policy. This model is not suitable, however, for arbitrary grid applications. In M-grid [37], each computational resource keeps a standard java-enabled browser continuously running that executes applets downloaded from a central job management website. M-grid relies on built-in applet sandboxing to protect resources from malicious code, but the severe restrictions imposed by the default Java sandbox means that M-grid can only be used for very limited classes of applications.

3. MESH OVERVIEW

3.1 Mesh Architecture

Each Mesh installation is dedicated to serving a particular *virtual organization* (VO), which is defined to be a "dynamic collection of individuals, institutions, and resources" [8]. Mesh allows individuals in one VO to execute SSH remote commands on Mesh-accessible resources within other cooperating VOs using a single Mesh SSH private key (hereafter abbreviated to *Mesh key*) generated at the local VO. A full Mesh deployment consists of two dedicated hosts and three primary software components (besides SSH itself). The first host, called the *Mesh Proxy* (MP), is responsible for mediating all SSH remote commands that are to execute on Mesh-accessible resources within the VO. Users that attempt to bypass the MP and contact VO resources directly will be unable to utilize Mesh authentication, thus ensuring complete mediation.

Command mediation on the MP is carried out by a software component called the *Mesh Authorization Shell* (Mash). Mash is a highly flexible and customizable login shell replacement that parses remote commands and authorizes them against a site security policy. As part of the authorization process, commands can optionally be rewritten to force compliance with specific site policies or to provide enhanced usability. Authorized commands are passed on to the appropriate VO resource for execution using a second SSH remote command.

VO resources are made Mesh-accessible by injecting a software component called the *Mesh Interposition Agent* (MIA) into the resource's SSH server using library preloading, which dynamically modifies its behavior during public key authentication. Instead of authenticating against the authorized_keys file stored locally in the user's home directory, the MIA causes the server to authenticate against an authorized_keys file retrieved at run-time from a second dedicated host within each VO called the *Mesh Authentication Point* (MAP).

Single sign-on is achieved during key retrieval from the MAP and authentication to the MP. In Mesh, each individual is assumed to have a *home VO*, which is the VO with which they are most frequently associated (e.g. the institution for which they work). When a key is retrieved by the MIA from a MAP that is not the user's home MAP, the key retrieval is propagated to the home MAP. Likewise, when authenticating to an MP that is not the user's home MP, that MP will initiate a key retrieval from the home MP. Thus, a Mesh key generated at the home VO is valid at all VOs. Figure 1 shows the basic steps of this process, which will be discussed in detail throughout the remainder of the paper.

Once the user has authenticated successfully on the Mesh-accessible resource, the MIA ignores the user's login shell and instead executes the remote command issued by the MP using a software component called the *Mesh Exec Security Shell* (Mess). Mess is a constrained execution shell that ignores metacharacters and only executes programs authorized by the administrator and not disallowed by the user. While the command is executing, it is subject to read, write, and execution controls enforced by the MIA. Once the command terminates, the SSH sessions will terminate as the final step.

The authentication and authorization components of Mesh may be deployed independently, thus allowing VOs to choose between a full deployment, a deployment of only the single sign-on features without proxying or additional authorizations, or a deployment of only authorization components without single sign-on. To simplify the discussion, the remainder of the paper assumes a full deployment. Although originally intended for grid-like operations, organizations without an interest in grid computing can still deploy some or all of Mesh to take advantage of the additional security features it adds to stock SSH installations.

3.2 Mesh Usage

To use a Mesh-based grid, users must obtain a Mesh key by invoking the *mesh-keygen* command on their home MP (via an SSH remote command), as shown in steps (1) and (2) of Figure 2. This command uses ssh-keygen to create a public/private key pair. The public key is then stored on the MP and copied to the MAP using an SSH remote command. The user must authenticate to both the MP and the MAP during this process to prevent an attacker that compromises the MP from arbitrarily placing keys on the MAP, thereby compromising all accounts. Once the user has successfully authenticated twice, the private key is returned for their use. Keys may also be obtained from foreign (i.e. non-home) MPs, but those keys may only be used within the associated VO. This provides resiliency in case the home MP is not accessible. Users may revoke one or all keys by invoking the *mesh-keykill* command on the MP on which they were generated.

The Mesh key generation process was explicitly designed to accommodate VOs that require two-factor authentication to access resources. Since grids are used for processing complex workflows that require unattended remote operations, they are fundamentally incompatible with two-factor authentication since the user is not always there to provide the second factor, such as a physical token

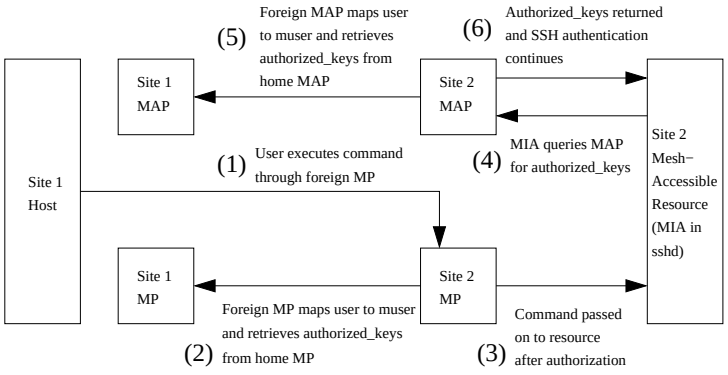

Figure 1: Mesh remote command processing

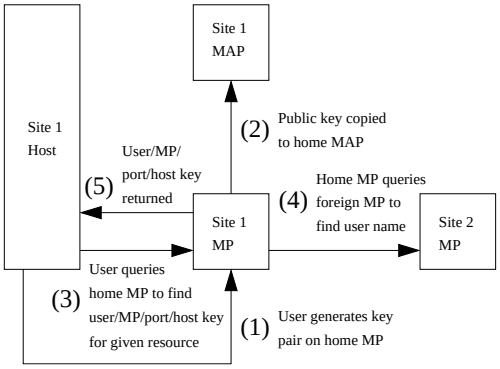

Figure 2: Mesh user preparation

or biometric property. For such VOs, Mesh can be configured to require two-factor authentication to obtain a Mesh key and to limit key lifetime to the extent desired. Thus, while it is still not possible for the user to be present during unattended operations, it can be guaranteed that all Mesh keys used for such operations are directly bound to a successful two-factor authentication within some configurable time into the past. This allows Mesh to coexist with strong authentication to the extent possible.

In the Mesh design, the user is only required to know the host name and public SSH host key (or fingerprint) of their home MP. When a user wishes to invoke a remote command on a Mesh-accessible resource outside of their home VO for the first time, they must retrieve the information needed to communicate with the MP protecting that resource. This includes the host name of the MP, the port number and public host key of the SSH server on that MP, and the user name of that individual on the MP. This information can be retrieved by invoking the *mesh-getmp* command on their home MP, as shown in step (3) of Figure 2. The MP host name and port are derived based on a mapping from IP addresses/netmasks to MP host names, called the *mps file* that is exchanged between VOs along with the MP/MAP public SSH host keys when they first decide to cooperate. The appropriate user name is retrieved from the foreign MP and then returned to the user with the rest of the information, as shown in steps (4) and (5) of Figure 2.

4. MESH AUTHENTICATION

The authentication component of Mesh is responsible for adding single sign-on to the standard SSH public key authentication mechanism. In the standard mechanism, each user has an *authorized_keys file* on each host (traditionally in their ~/.ssh directory), that dictates which private keys can be used to authenticate to the SSH server. A private key can be used for authentication if the corresponding public key is listed in the authorized_keys file. The standard mechanism is unsuitable for single sign-on as it would require copying every user's authorized_keys file to every host, which does not scale to large numbers of hosts across multiple VOs.

Instead of distributing authorized_keys files across all grid resources, Mesh uses *system call interposition* [13] to dynamically modify the behavior of system calls within stock SSH servers to retrieve public keys on-demand from a centralized source within each VO when needed. This approach is preferable to modifying SSH itself since it does not require source code to be kept up-to-date with the latest patches and revisions nor is there the possibility for introducing bugs into the code. It is also preferable to using a customized file system such as FUSE [9] to affect access to the authorized_keys file as changes are isolated to a single process on the system, thus keys are not subject to tampering by other processes. The traditional approach to adding authentication mechanisms to Unix systems through Pluggable Authentication Modules (PAM) [29] is not suitable for Mesh because no PAM module exists to perform SSH public key authentication. Finally, system call interposition allows Mesh to enforce additional access controls by intercepting appropriate system calls, as will be discussed in Section 5.4, which would not be feasible with other approaches.

4.1 Mesh Interposition Agent

During authentication, the Mesh Interposition Agent (MIA) intercepts SSH server system calls related to authorized_keys file access. The MIA is written in Bypass [34], which is a minimal syntactic wrapper around C/C++ code that isolates the user from slight differences in system call interfaces and implementations between Unix operating systems. The MIA has been implemented for OpenSSH [21], which has an almost 90% market share [32], thus will work with the vast majority of SSH installations. A similar approach would most likely work with other SSH implementations (in particular, the SSH.COM Tectia Server [31], which shares its ancestry with OpenSSH), but will not be discussed. The MIA is injected into an OpenSSH server using library preloading by setting the appropriate environment variable (e.g. LD_PRELOAD or _RLD_LIST) to the location of the MIA shared library.

In OpenSSH, an authorized_keys file is read into memory using two basic steps. First, the server invokes the stat() system call on the key file to obtain information such as ownership, permissions, size, etc. For the server to accept the key file, it must exist with a size greater than zero and have appropriately restrictive permissions (i.e. not group or world writable). If these conditions hold, the server reads each line (i.e. key) in the file using the fgets() standard C library function and performs the appropriate steps to test if the user has the corresponding private key. Note that this behavior is identical across all versions of OpenSSH portable releases from the initial version 1.0pre1 of Oct. 1999 up to the latest version 4.5p1 of Nov. 2006. Thus, the MIA is likely to work with all past releases as well as new releases for the foreseeable future.

The MIA intercepts the stat() call to guarantee that whenever the stat() of a key file is requested, that file will exist with the correct permissions and a non-zero size. When the file does not already exist, it is created and padded to a non-zero size. The intercepted stat() then uses the standard stat() to return the file's information. The fgets() call is intercepted to provide an appropriate authorized_keys file to the SSH server. In this case, an fgets() call to the user's au-

thorized_keys file produces a callout to an external program called *mesh-getkey-hook*. This program is responsible for returning an authorized_keys file to a given file stream that can be read a line at a time by the intercepted fgets() call. Since mesh-getkey-hook may need to access sensitive data not accessible by ordinary users such as private keys, it is executed with elevated (i.e. root) privilege. After it returns, privileges are dropped back to the user level. In the default Mesh implementation, mesh-getkey-hook is a simple script that retrieves an authorized_keys file via SSH from the local MAP, which is described in the next section.

4.2 Mesh Authentication Point

The Mesh Authentication Point (MAP) provides an authorized_keys file for each user of a particular VO to MIAs running on Mesh-accessible resources within the VO. These files are retrieved from the MAP during the mesh-getkey-hook callout of each MIA, which performs an SSH to the local MAP and invokes the *mesh-getkey* command, as shown in step (4) of Figure 1. This command takes a user name and returns that user's authorized_keys file with the SSH connection ensuring the integrity of the file.

In order to invoke mesh-getkey on the MAP, each Mesh-accessible resource must have a private key that can be used to issue a remote command to the MAP as a designated MAP user. Using the mechanisms that will be discussed in Section 5.2, each user/key pair is restricted such that the one and only operation it may perform is retrieving authorized_keys files via mesh-getkey. Thus, even if a Mesh-accessible resource is compromised and the MAP key read, the attacker can only obtain user public keys, which cannot be used to compromise the private keys, thus has negligible security impact.

For resiliency, Mesh-accessible resources of the local VO accept both locally generated Mesh keys as well as keys generated at the user's home VO. Thus, the authorized_keys file returned by mesh-getkey must contain both the public key stored on the local MAP as well as the public key stored on the user's home MAP. Since the user names of the same individual may differ across VOs or across organizations within the same VO, the local MAP cannot simply use the same user name it was given by the MIA to request the appropriate public key from the home MAP. Otherwise, the key returned by the home MAP may be for a completely different individual or may not exist at all. To ensure that MAPs refer to the same individual, mesh-getkey first maps the local user name into a globally unique Mesh user name in X.509 Distinguished Name [36] format, called the *muser name*. This is done using a file of three-tuples called the *musers file*. Tuples consist of a user name a muser name and an organization within the local VO to which this mapping applies. Note that VOs must only define tuples for their own user base. As with any grid solution, however, VOs may have policies that require all users of their systems to have a local account (with associated paperwork), in which case the local user base by necessity also contains individuals from other VOs. For VOs with more flexibility, such as permitting dynamically-generated accounts, the administrative overhead of the musers file can be avoided in favor of a dynamic user/muser mapping function.

A second configuration file, called the *maps file*, containing mappings from muser-based regular expressions to MAP host names, is then used to find the home MAP of the muser. To retrieve the public key from the home MAP, the local MAP must have a user/key pair for the home MAP just as the local MIA has a user/key pair for the local MAP. In general, each MAP must have a user/key pair for all other MAPs with which it cooperates. The local MAP uses the appropriate user/key pair to invoke mesh-getkey on the home MAP, as shown in step (5) of Figure 1. The home VO key is then concatenated with the local VO key and returned to the requesting MIA, as shown in step (6) of Figure 1. The resulting authorized_keys file is redirected by the calling mesh-getkey-hook into a file stream, which is read by the MIA and returned as the result of fgets(). Finally, the SSH server verifies that the user has the corresponding private key and authentication either succeeds or fails.

5. MESH AUTHORIZATION

The authentication described in the previous section also provides simple authorization. Namely, if a given user does not possess an appropriate private key with which they can authenticate, they will be denied access to all Mesh-accessible resources. If desired, Mesh can be deployed with no further authorizations. Basic system authorization, however, is not sufficient for strong security in grid environments, thus a full deployment is recommended. First, for a resource added to a grid to be useful, that resource must be accessible from throughout that grid, which also provides a channel of attack. Many organizations block access to internal resources using a firewall and/or bastion for just this reason, thus exposing these resources to the outside world is not compatible with existing security models and infrastructures. Second, grids support delegated credentials that may be used by systems/services to perform operations on the user's behalf. Thus, compromises or bugs associated with such systems/services may lead to those credentials being used for unintended or unexpected purposes. For this reason, the operations permitted by grid credentials should be limited according to site security policy as well as a policy acceptable to the user.

Besides basic system authorization, Mesh implements four additional layers of authorization. First, remote commands may only be invoked on Mesh-accessible resources by passing through the MP. Second, remote commands are only allowed to pass through the MP after authorization and rewriting by Mash. Third, commands on Mesh-accessible resources are subject to additional host- and user-specific authorizations enforced by Mess. Finally, commands are restricted in what they can do on each Mesh-accessible resource by the MIA.

5.1 Mesh Proxy

The MP is responsible for limiting the exposure of Mesh-accessible resources within a particular VO to attack by mediating all remote commands before they are executed on those resources. In general, any command "ssh host command" that the user wishes to execute on a given Mesh-accessible resource, must be prepended with "ssh MP" to that VO's MP. If the user tries to bypass the MP, even if the remote host is accessible from the network, authentication to that host using their Mesh key will fail.

This protection is implemented by the interception of the accept() system call by the MIA. In OpenSSH, the server accepts one waiting network connection using accept(), then forks a process to handle the connection. As part of an accept() call, the IP address of the connecting host is filled into an appropriate data structure. The MIA uses this address to modify its behavior accordingly. If the host corresponds to the IP address of an MP defined in the MESH_MP_IPS environment variable, the MIA will authenticate using the MAP. If the host does not correspond to an MP, the MIA does not load itself into the forked process, thus forcing SSH back to built-in authentication methods. This approach provides compatibility with existing security models and infrastructure since users not connecting through the MP can use whatever credentials and authentication methods the VO normally supports (e.g. their normal authorized_keys file) and allows minimal installation and configuration on existing hosts since only a single SSH server is required (although a second SSH server can still be run if desired).

In order for the MP of one VO to accept a key generated at another VO, MPs must support remote key retrieval in the same manner as Mesh-accessible resources. This is enabled in the same fashion by preloading the MIA into the MP SSH server. Instead of invoking mesh-getkey on another host, however, the mesh-getkey-hook of the MP invokes mesh-getkey on itself, which retrieves both the locally stored public key as well as the public key from the user's home MP, as shown in step (2) of Figure 1. Note that the home VO key is retrieved from the home MP instead of the home MAP to maintain the separation of privilege between site and host authentication carried out by the MPs and MAPs, respectively. Thus, each MP must have a user/key pair with which to retrieve public keys from other MPs, just as each MAP has similar access to other MAPs. Each MP must also have its own musers and maps files to derive the home MP.

5.2 Mesh Authorization Shell

Since all remote commands to Mesh-accessible resources must pass through the MP, this also provides a point at which those commands can be examined for conformance to site security policy. For example, a policy might specify which commands each user is allowed to execute and with what arguments. SSH provides a built-in mechanism for enforcing such policies using *forced commands* in the authorized_keys file. Forced commands are only applied during public key authentication, however, thus cannot offer any protection during Mesh key generation, when non-public-key authentication mechanisms are used. Instead, the approach used by Mesh is to set the default shell for all MP users (also used for all MAP users) to a special restricted shell called Mash. SSH invokes the user's shell for both remote commands as well as interactive sessions, thus Mash is always invoked and has access to the user's command before it is actually executed.

The set of security policies that Mash enforces is defined in an XML configuration file called the *mashrc file*. Each policy in this file specifies the restrictions that the remote command must satisfy before it is executed. Mash distinguishes between commands allowed to execute locally on the MP (or MAP) and commands proxied by a second SSH connection to a Mesh-accessible resource. In the latter case, the arguments and options to the second SSH session are parsed and stripped away before authorizing the remainder of the command. Each command or set of commands can be associated with a set of rules that define the conditions under which it is authorized. Mash currently implements eight types of rules based on key generation time, SSH connection properties, command options, command arguments, invoking user, invoking group, home VO of invoking user, and environment settings. Delegation of the SSH agent connection to the target host can also be enabled or disabled on a per command basis.

Figure 3 shows a sample fragment of a mashrc file for the "bbftpd" command, which invokes the server portion of the BbFTP protocol [3] discussed later in Section 6.2. The *parsers* section defines how command options should be parsed (based on Perl Getopt::Std and Getopt::Long syntax) as well as the basic set of rules that hold for all instances of a particular command. The bbftpd command accepts options without values in the set {b, c, f, p, s, u, v} as well as options that take values in the set {e, l, m, w, R}. Additionally, it does not take any non-option arguments besides the command name, it should require the "-s" option, and it should disallow options in the set {b, c, p, u, v, w, R}. The *proxies* section defines on which hosts commands may execute along with additional restrictions and host-specific configuration. In the example, bbftpd is allowed to execute on the host "some_host", but only by user "some_user" and only when authenticating using a Mesh key. The

```
<mashrc>
 <parsers>
  <bbftpd>
   <getopt>bcfpsuve:l:m:w:R:</getopt>
   <argument><count>1</count></argument>
   <option>
    <regex>^(?=.*(^|\n)s)(?!.*(^|\n)[bcpuvwR])</regex>
   </option>
  </bbftpd>
 </parsers>
 <proxies>
  <ssh>
   <directory>/usr/bin</directory>
   <port>22</port>
   <argument>
    <value index="1">some_host</value>
   </argument>
   <environment>
    <set name="MESH_PUBKEY"/>
   </environment>
   <user><some_user/></user>
   <commands>
    <bbftpd>
     <directory>/usr/bin</directory>
    </bbftpd>
   </commands>
  </ssh>
 </proxies>
</mashrc>
```

Figure 3: Sample mashrc fragment for BbFTP

instance of bbftpd allowed to execute on some_host is the one installed in the "/usr/bin" directory.

In addition to disallowing specific arguments and/or options to remote commands within the configuration file, Mash also performs command rewriting to force compliance with site security policy and to enhance usability. Security-related rewriting includes operations such as stripping undesirable options and/or arguments and replacing relative paths with absolute paths to ensure that users execute only the authorized instances of commands. Since paths may differ across grid resources, path rewriting also enhances usability since the knowledge of where each authorized command resides is stored on the MP. Thus, a user may just use the base command name and the correct path will be automatically prepended. This is similar to the UNICORE's instantiation of abstract job objects when passing through its Network Job Supervisor [5]. Once Mash has authorized and rewritten a command, it is then passed on to the Mesh-accessible resource for execution, as shown in step (3) of Figure 1.

5.3 Mesh Exec Security Shell

After the SSH server on a Mesh-accessible resource authenticates the user based on the authorized_keys file returned from the MAP, it then executes the user's command. Normally, this command is executed with the user's login shell. In Mesh, however, the MIA dynamically maps the user's login shell to a restricted shell called Mess, which provides protection against many of the attacks to which traditional shells are susceptible. First, Mess will only execute those commands authorized by the administrator in a global configuration file called the *meshrc file*. While this protection is provided by Mash on the MP, Mess provides defense-in-depth such that even if the MP itself is compromised, an attacker still cannot execute arbitrary commands. This increases administrative overhead on each Mesh-accessible resource, but the simple format of the meshrc file allows each command to be authorized using a single line "+x /path/to/command", thus overhead is fairly minimal.

For additional protection, Mess does not support relative paths, thus attackers cannot execute alternative commands through path variable manipulation. Usability is unaffected since Mash rewrit-

ing on the MP always provides the absolute path to all commands. Mess also does not support metacharacters, which prevents hiding additional commands in original command's arguments using metacharacters such as backtick. Finally, Mess supports user-level policies that allow users themselves to specify which operations are permitted on each Mesh-accessible resource. Users specify these restrictions in a meshrc file in their home directory on each Mesh-accessible resource. By default, if no such file exists for a given user, no Mesh operations can be performed with that account. If the file does exist, all operations authorized by the global meshrc are allowed, but users can disable any or all of those commands using "-x /path/to/command".

5.4 Mesh Interposition Agent

Enforcing a given site security policy is complicated by the fact that the user must not be restricted in any way on Mesh-accessible resources when they authenticate through a site's normal authentication procedures or else Mesh would not be compatible with existing security infrastructure. Since the user may have another unrestricted means to access these resources, they are able to modify their configurations on those systems at will. Thus, a primary concern is ensuring that configuration changes made through normal access channels cannot be used to interfere with remote commands issued by the MP.

Commands issued to an SSH server are susceptible to several forms of interference, some of which cannot be prevented within current OpenSSH implementations. These include setting environment variables in ~/.ssh/environment that affect command execution, such as LD_PRELOAD, adding arbitrary commands to ~/.ssh/rc or to shell startup files such as ~/.cshrc, ~/.tcshrc, or ~/.zshenv, and adding forced commands to ~/.ssh/authorized_keys. Although many of these forms of interference are prevented by the use of Mess, additional code within the MIA provides its own protection for defense-in-depth. All but forced commands are blocked by disallowing access to ~/.* files with the exception of ~/.ssh/authorized_keys, which must be accessible by the SSH server even if it is not the file actually used as described in Section 4.1. This is done by intercepting the system calls relevant to file access such as open() and stat(). The code also protects against indirect access via manipulation of hard and symbolic links. It is not possible for users to add forced commands within Mesh since the authorized_keys file used for authentication is retrieved from the MAP. The MIA still provides protection against forced commands in the retrieved file by ignoring key lines containing the term "command" in case the MAP itself has been compromised.

Because grid credentials may be delegated for use by a variety of systems and services, it is desirable to place additional restrictions on the actions those credentials may perform. Besides the time and scope limitations provided by Mash and Mess, the MIA provides one final layer of authorization. By the time a remote command is ready for execution on a Mesh-accessible resource, it has been severely vetted by other Mesh components. A remaining concern, however, is that a legitimate operation permitted through the MP can be used to compromise a later unrestricted non-Mesh session. For example, if file transfer commands such as SCP are allowed through the MP, then the attacker could copy their own authorized_keys file to be used through a non-Mesh session or plant a trojan in the user's ~/bin directory. Since, in general, an attacker cannot interfere with later sessions if they cannot modify files, by default, the MIA does not allow any writes to the file system unless previously authorized. This protection is implemented by intercepting additional file-related system calls such as chmod(), rename(), unlink(), etc. Users can authorize writes to specific directories by listing them in their meshrc file, which must be edited through a non-Mesh session. Administrators can authorize writes for all users using the global meshrc file. Writes to any file or directory beginning with "." in the user's home directory are always forbidden, thus preventing most attacks even if the user permits writes in their home directory.

6. MESH SERVICES

The Mesh security core of Sections 4 and 5 provides the basic grid foundation of single sign-on, delegation, and fine-grained authorization. Using this foundation, VOs must then decide which services they wish to make available and under what conditions. Any service to be incorporated into Mesh must be invokable from the command line. This includes any application with a command line interface (CLI) as well as X11 applications that may utilized through the built-in X11 forwarding functionality of SSH. TCP services may be utilized similarly using the SSH port forwarding feature, which allows the use of the native client for the duration of the MP connection. Since the associated TCP server is a separate process invoked independently of the SSH server, however, Mesh cannot enforce the MIA authorizations in this case unless the MIA is also loaded into that server at startup.

An SSH remote command service model offers several advantages. Perhaps the most important advantage is usability. Users do not need any specialized knowledge to take advantage of a Mesh-based grid and can utilize the same interfaces they know and use every day from their interactive prompt. The second advantage is ubiquity. SSH is available on a wide range of devices from PDAs to routers to game consoles with APIs for all of the most commonly used programming languages. Any device with SSH and network access can be used to access a Mesh-based grid without the need for any additional software.

After authentication and authorization, the most basic grid services [8] are those for resource discovery and query, high performance file transfer, and job management. Together, these services support the traditional grid computing model where users (1) determine what resources are available and what they are capable of, (2) transfer data and/or software resources to the storage resource(s) associated with a given compute resource, and (3) execute that software using that data on that compute resource. Mesh has been integrated with several existing projects to provide these basic services. The following sections briefly describe these services and the security considerations for each.

6.1 Resource Discovery and Query

Grid information services collect information about the resources on a grid, which can be queried by users and/or other services. Grid resource brokers use this information to automatically select and rank resources based on user constraints and preferences. For example, to execute properly, a user's application may require a compute resource with Java JRE 1.5.0 installed. The same interfaces that allow users to find and select the most desirable resources for running their applications, however, also allow attackers to find and select the resources most vulnerable to attack such as compute resources running versions of software with known remote exploits. Thus, at the minimum, such services must only be exposed to authenticated users. Information services have need of additional authorizations as they must accept information updates from systems and, in some cases, users.

Mesh has been integrated with the Surfer resource broker [15] and the Pour information service [16], both of which were designed to be lightweight and accessible from the command line. Surfer CLI access is restricted using basic Mesh authentication and autho-

rization, while delegation is used by Surfer to query Pour instances across VOs. Pour is unique in that it can generate certain types of information on-demand when not found in the local database using remote commands constructed from the contents of user queries. The executables used for these queries are fixed for each Pour instance, however, thus can be authorized individually using Mash and Mess configuration. Pour implements its own security to restrict access to its database since it is beyond the control of Mesh file system authorizations. Users are allowed to add their own information, but all such information is tagged with their grid identity, which in the case of Mesh is their muser name. Users can only remove information they added and can limit their queries to ignore information added by others.

6.2 High Performance File Transfer

While initial access to resources may occur through grid mechanisms, the applications that eventually execute on compute resources operate in terms of traditional file systems and the associated access control models. File transfer services for grids must respect existing permissions and must support authorization at the file/directory level in order to protect the sensitive files in user directories that can affect current and future logins such as shell startup files, authorized_keys files, executables and libraries in the user's path, etc. Because Mesh operates at the system call level and restricts access to individual directories as discussed in Section 5.4, Mesh provides strong security for any file transfer service that can be invoked directly by the user over an SSH connection. Mesh supports built-in SCP and SFTP transfers and has been integrated with Bbcp [2], BbFTP [3], and Rsync [27] to provide high performance capabilities.

6.3 Job Management

Unlike traditional compute resources where users can directly execute programs as desired, high-end compute resources are under the control of job managers that allocate chunks of the resource for exclusive use by individual users for limited periods of time. In the grid paradigm, high-level grid services may submit jobs on the user's behalf. For example, in a complex workflow, less compute-intensive tasks may be performed on low-end resources with the resulting data transferred to a high-end resource and a job automatically submitted to perform the next stage of the computation. Since jobs execute arbitrary code of unknown effect completely beyond the control of the submitting grid software once passed to the job manager, delegated job submission represents a significant security risk. Given the high cost, large number of processors, and possibly one-of-a-kind nature of high-end compute resources, however, imposing additional security controls, which would result in a loss of precious CPU cycles, is not an acceptable option.

Mesh has been integrated with the Portable Batch System (PBS) [23], where the PBS "qsub", "qdel", and "qstat" commands may be used to submit, revoke, and monitor jobs, respectively. The approach taken by Mesh to secure job submission is based on fail-safe defaults and least privilege. By default, users cannot submit any jobs through Mesh even if authorized by the administrator. To enable job submission, the application path given to the qsub command must match a path that has been authorized individually by the user in their meshrc file using a line "+qsub /path/to/application". Thus, users must accept the risks of job submission only for the applications they specifically need. This protection takes advantage of extended rules within Mess that allow any command authorized with "+x /path/to/command" in the global meshrc file to have an extended rule "+command <constraints>", which can be tested against the contents of the user's meshrc file. Since this authorization takes place on Mesh-accessible resources, job submission capabilities are also protected against a compromise of the MP.

7. MESH PERFORMANCE

Table 1 shows the overhead imposed by Mesh during the various authentication and authorization stages of a command's execution. Measurements were gathered in a testbed consisting of four 2.4 GHz Pentium 4 machines running Linux with 512 MB of memory connected by 100 Mb/s ethernet. Each command was run with the target in the home VO and with the target in a foreign VO. In the foreign VO tests, one host had dual roles as the home MAP and MP, respectively. The overhead for the mesh-getkey command on the MAP was less than that of the /bin/true command on the MP because the MAP's Mash policy is smaller, thus requires less processing. Foreign VO operations incurred an additional overhead of two mesh-getkey commands.

From the user's perspective, the Mesh overhead incurred for a single command is fairly insignificant. This overhead becomes noticeable, however, when running many commands in quick succession as seen in the "10 * ssh MP ssh target /bin/true" case. To overcome sequential overhead, Mash and Mess support batched commands using the standard shell semicolon operator. Each batch is authenticated, authorized, and executed as a group, thus incurring Mesh overhead only once per batch, which significantly reduces the total execution time as seen in the "ssh MP ssh target 10 * /bin/true" case. The low-level authorizations of the MIA also incur minimal overhead as shown by the SCP transfer of 100 files. The performance of the applications integrated with Mesh is reported in respective papers for Surfer [15], Pour [16], Bbcp [12], BbFTP [12], and Rsync [35].

The main scalability concern in the Mesh architecture is the MP since it is responsible for proxying all remote commands issued to a particular VO. Each MP can only support a finite number of concurrent SSH sessions due to hardware and/or operating system limitations such as the amount of physical/virtual memory, the maximum number of concurrent processes, and the maximum number of open file descriptors. These limits may be reached when supporting a large number of users or through frequent use of commands with potentially long running times, such as large file transfers. Typical server configurations can easily support thousands of simultaneous connections, but for VOs in which these limits become a problem, the MP can be scaled linearly using multiple physical hosts with an appropriate server load balancing mechanism such as Linux Virtual Server (LVS) [17]. In this configuration, it is necessary for the MPs to share the same SSH host key and have access to the same set of user public keys. Keys can be shared using an appropriately secured shared file system or a secure copy to all other MPs of the cluster during key generation. Alternatively, these VOs can deploy Mesh in its less secure, but more easily scalable configuration that allows direct access to Mesh-accessible resources. The MAP only supports short-lived commands by default so is less likely to be affected by these limits.

8. CONCLUSIONS AND FUTURE WORK

This paper has described a new lightweight grid middleware called Mesh, **M**iddleware using **E**xisting **S**SH **H**osts. Mesh provides a single sign-on grid environment based on SSH public key authentication with facilities for resource discovery and query, high performance file transfer, and job management. The Mesh security architecture limits potential damage from compromise to the greatest extent possible and adheres to fundamental security principles [28].

	Target Location	Home VO			Foreign VO		
Command	Measure	Non-Mesh	Mesh	Total	Non-Mesh	Mesh	Total
ssh MAP mesh-getkey user (from target)		0.17	0.24	0.41	0.17	0.60	0.77
ssh MP /bin/true		0.17	0.47	0.64	0.17	0.89	1.06
ssh target /bin/true (from MP)		0.18	0.46	0.64	0.18	0.85	1.03
ssh MP ssh target /bin/true		0.35	1.00	1.35	0.35	1.81	2.16
10 * ssh MP ssh target /bin/true		3.50	10.0	13.5	3.50	18.3	21.8
ssh MP ssh target 10 * /bin/true		0.35	1.57	1.92	0.35	2.38	2.73
scp -S 'ssh MP ssh' -r /dir target: (100 1MB files)		14.4	1.10	15.5	14.4	1.90	16.3

Table 1: Mesh overhead (secs)

Users access remote services based on the commands and syntax with which they are already familiar, thereby promoting *psychological acceptability*. All access to Mesh services must pass through the Mesh Proxy (MP), which provides *complete mediation* of remote commands. Only those operations authorized by site policy are allowed to pass through the MP and only for a finite time frame, thus limiting users to *least privilege*. A *separation of privilege* between site and host authentication is provided by the MP and Mesh Authentication Point, respectively, which prevents a full site compromise when only one is breached. A separation of privilege is also provided between the user's ability to execute a remote command and the ability of that command to write to the remote file system by requiring the user to login interactively to explicitly authorize writes beyond the *fail-safe default* of no write access. A similar separation and default is provided for job submission, where users must authorize each application individually. The entire code base of the Mesh security core is less than 2500 lines of code, thereby achieving *economy of mechanism*, and will be released as open source to provide an *open design* subject to scrutiny by all.

There are several directions for future research. Additional restrictions may be added to the Mesh Interposition Agent such as allowing the user to restrict what directories can be read beyond normal file system permissions when they invoke operations through Mesh. For example, a directory with sensitive data can be excluded in case the user's Mesh key is compromised. Support for other SSH server implementations will also be investigated. Alternatives to library preloading will be studied to support protection within static binaries. In general, more experience is necessary with Mesh in a production environment to determine the capabilities that are missing or that need to be enhanced.

9. REFERENCES

[1] Alfieri, R., Cecchini, R. et al.: From gridmap-file to VOMS: Managing Authorization in a Grid Environment. Future Generation Computer Systems, vol. 21, num. 4, 2005.

[2] Bbcp. `http://www.slac.stanford.edu/~abh/bbcp`.

[3] BbFTP. `http://doc.in2p3.fr/bbftp`.

[4] Doyle, A.T., Lloyd, S.L., McNab, A.: GridSite, GACL and SlashGrid: Giving Grid Security to Web and File Applications. UK e-Science All Hands Meeting, Sep. 2002.

[5] Erwin, D.W., Snelling, D.F.: UNICORE: A Grid Computing Environment. 7th Intl. Euro-Par Conf., Aug. 2001.

[6] Foster, I., Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit. Intl. J. Supercomputer Applications, vol. 11, num. 2, 1997.

[7] Foster, I., Kesselman, C. (eds.): The GRID: Blueprint for a New Computing Infrastructure. Morgan-Kaufmann Publishers, Nov. 1998.

[8] Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. Intl. J. Supercomputer Applications, vol. 15, num. 3, 2001.

[9] FUSE. `http://fuse.sourceforge.net`.

[10] Globus Project: GT 4.0: GSI-OpenSSH. Dec. 2005. Available at `http://www.globus.org/toolkit/docs/4.0/security/openssh`.

[11] Hayes, M., Morris, L. et al.: GROWL: A Lightweight Grid Services Toolkit and Applications. UK e-Science All Hands Meeting, Sep. 2005.

[12] Hughes-Jones, R., Dallison, S.: Investigating the Interaction Between High-Performance Network and Disk Sub-Systems. 3rd Intl. Wkshp. on Protocols for Fast Long-Distance Networks, Feb. 2005.

[13] Jones, M.B.: Interposition Agents: Transparently Interposing User Code at the System Interface. 14th ACM Symp. on Operating System Principles, Dec. 1993.

[14] Kohl, J.T., Neuman, B.C., Ts'o, T.Y.: The Evolution of the Kerberos Authentication Service. Spring 1991 EurOpen Conf., May 1991.

[15] Kolano, P.Z.: Surfer: An Extensible Pull-Based Framework for Resource Selection and Ranking. 4th IEEE/ACM Intl. Symp. on Cluster Computing and the Grid, Apr. 2004.

[16] Kolano, P.Z.: A Unified Framework for Periodic, On-Demand, and User-Specified Software Information. 5th IEEE/ACM Intl. Wkshp. on Grid Computing, Nov. 2004.

[17] Linux Virtual Server. `http://linuxvirtualserver.org`.

[18] Litzkow, M., Livny, M., Mutka, M.: Condor - A Hunter of Idle Workstations. 8th IEEE Intl. Conf. of Distributed Computing Systems, Jun. 1988.

[19] McCullough, M.: Secure Automated File Transfer. ;Login:, 30(4), Aug. 2005.

[20] McKeown, M.: Build WS-Resources with WSRF::Lite. Jan. 2005. Available at `http://www-106.ibm.com/developerworks/edu/gr-dw-gr-wsrflite-i.html`.

[21] OpenSSH. `http://openssh.org`.

[22] Pearlman, L., Welch, V., Foster, I., Kesselman, C., Tuecke, S.: The Community Authorization Service: Status and Future. Conf. for Computing in High Energy and Nuclear Physics, Mar. 2003.

[23] Portable Batch System. `http:`

//www.altair.com/software/pbspro.htm.

[24] Provos, N.: Improving Host Security with System Call Policies. 12th USENIX Security Symp., Aug. 2004.

[25] Riedel, M.: UNICORE Secure Shell Plugin Guide. Oct. 2005. Available at http://prdownloads.sourceforge.net/unicore/sshpluginguide_1_0_1.pdf.

[26] Rssh. http://www.pizzashack.org/rssh.

[27] Rsync. http://samba.anu.edu.au/rsync.

[28] Saltzer, J.H., Schroeder, M.D.: The Protection of Information in Computer Systems. Proc. of the IEEE, vol. 63, num. 9, 1975.

[29] Samar, V.: Unified Login with Pluggable Authentication Modules. 3rd ACM Conf. on Computer and Communications Security, Mar. 1996.

[30] Scponly. http://www.sublimation.org/scponly.

[31] SSH Tectia Server. http://www.ssh.com/products/client-server.

[32] SSH Usage Profiling. http://openssh.org/usage.

[33] Thain, D.: Identity Boxing: A New Technique for Consistent Global Identity. ACM/IEEE Supercomputing 2005 Conf., Nov. 2005.

[34] Thain, D., Livny, M.: Multiple Bypass: Interposition Agents for Distributed Computing. J. Cluster Computing, vol. 4, num. 1, 2001.

[35] Tridgell, A.: Efficient Algorithms for Sorting and Synchronization. Ph.D. Thesis, Australian National Univ., Feb. 1999.

[36] Wahl, M., Kille, S., Howes, T.: Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names. IETF RFC 2253, Dec. 1997.

[37] Walters, R.J., Crouch, S.: M-grid: Using Ubiquitous Web Technologies to Create a Computational Grid. European Grid Conf., Feb. 2005.