

REMUS: A Rerouting and Multiplexing System for Grid Connectivity Across Firewalls

Jefferson Tan · David Abramson · Colin Enticott

Received: 29 June 2007 / Accepted: 21 May 2008
© Springer Science + Business Media B.V. 2008

Abstract The Grid provides unique opportunities for high-performance computing through distributed applications that execute over multiple remote resources. Participating institutions can form a virtual organization to maximize the utilization of collective resources as well as to facilitate collaborative projects. However, there are two design aspects in distributed environments like the Grid that can easily clash: security and resource sharing. It may be that resources are secure but are not entirely conducive to resource sharing, or networks are wide open for resource sharing but sacrifice security as a result. We developed REMUS, a rerouting and multiplexing system that provides a compromise through connection rerouting and wrappers. REMUS reroutes connections using proxies, ports and protocols that are already authorized across firewalls, avoiding the need to make new openings through the firewalls. We also encapsulate applications within wrappers, transparently rerouting the connections

among Grid applications without modifying their programs. In this paper, we describe REMUS and the tests we conducted across firewalls using two Grid middleware case studies: Globus Toolkit 2.4 and Nimrod/G 3.0.

Keywords Grids · Grid connectivity · Firewalls · Networks · Network security · Wrappers

Abbreviations

FTP	File Transfer Protocol
DMZ	Demilitarized Zone
GRAM	Grid Resource Access and Management
GSI	Grid Security Infrastructure
GSSAPI	Generic Security Services Application Programming Interface
IP	Internet Protocol
REMUS	Rerouting and Multiplexing System
SSH	Secure Shell
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VO	Virtual Organization
WS	Web Service(s)

1 Introduction

The Grid refers to “a distributed computing infrastructure for advanced science and engineering” [8]. Grid applications require advanced computations through distributed processing, with an

J. Tan (✉) · D. Abramson · C. Enticott
Faculty of Information Technology,
Monash University, 900 Dandenong Road,
Caulfield East, VIC 3145, Australia
e-mail: jefferson.tan@infotech.monash.edu.au

D. Abramson
e-mail: david.abramson@infotech.monash.edu.au

C. Enticott
e-mail: colin.enticott@infotech.monash.edu.au

explicit focus on supporting *virtual organizations* (VOs). By participating in a VO, institutions are able to share and coordinate computations, commonly with the Internet as the underlying communication infrastructure. Unfortunately, on the Internet, there is always the risk of unauthorized access to data, transactions or computational resources. For this reason, networks on the Internet are often protected with firewalls and other measures. There is particular concern that network devices between source and destination, referred to as *middleboxes* [19], render communications impossible. This presents a dilemma for establishing VOs since this reduces the level of collaboration that is possible. Furthermore, many Grid applications assume that network connectivity is mostly unrestricted. Accordingly, VOs may face failed collaborations or considerable application reprogramming because the member sites have security policies that clash, resulting in more restrictions than anticipated in the design of their applications. Moreover, since the participants in a VO are independent of one another, their network security measures are likewise quite varied. These issues may be taken together as a *security interoperability problem* [7]. Somewhere in the interconnected networks involved, a mismatch occurs between what is allowed on one end and what is not allowed on the other. The result is a connectivity problem, where “benign applications cannot traverse a firewall” [21].

We have studied the likely scenarios resulting from security interoperability and explored or developed methods that facilitate connectivity despite such restrictions. The result is REMUS [25], a project in which we built a *rerouting* and *multiplexing* system that enables cross-firewall connectivity while minimizing the need to modify or rebuild either the Grid applications or the firewalls. Grid-running software, including user applications and Grid infrastructure middleware, are encapsulated within a virtually connected environment with minimal or zero code modification. Likewise, rather than engage in firewall evasion and avoidance, REMUS complies with the security policies for a given network. Only authorized channels are used, and mechanisms are provided to minimize the impact on existing security measures. In addition, REMUS uses different

connection mechanisms, depending on whichever is appropriate for a given scenario, e.g., using proxies or tunneling. REMUS is intended to support a library of such mechanisms, given that no single solution will always work. REMUS achieves “connectivity virtualization”—leaving Grid software and firewalls mostly or entirely untouched—by intercepting outgoing connections on all hosts and rerouting them across networks.

REMUS is also intended to be as portable as possible. Hence, our solutions are built from a number of standard secure communication protocols such as Secure Shell (SSH) [30] and SOCKS [15, 16]. In this manner, it becomes more likely for REMUS to be readily supported across different platforms.

This paper describes REMUS, which we designed to be a general-purpose Grid connectivity middleware that supports different Grid software and platforms. For purposes of testing and illustration, we implemented REMUS as a virtual connectivity middleware underneath the Globus toolkit and the Nimrod/G parametric modeling suite. However, the efficacy of our solutions for other Grid applications in similar situations will also be discussed. Given that network performance is affected by rerouting, we present performance measurements for various Grid operations with and without REMUS intervention.

This paper is structured thus: Section 2 discusses how Grid networking works and what firewalls are. Section 3 describes our case studies: Globus and Nimrod/G. Section 4 lays out the problems caused by multiple firewalls across Grids. We illustrate by discussing specific problems posed to Globus and Nimrod/G. Section 5 presents REMUS as a solution, while Section 6 discusses technical details about tunneling and port forwarding strategies in general. Section 7 discusses the implementation of REMUS in its current form, using SSH tunneling and SOCKS rerouting. We illustrate by discussing how it assists Globus and Nimrod/G, as particularly representative types of Grid middleware/applications. We also discuss security issues and features pertaining to REMUS. Section 8 describes our performance tests across two campuses of Monash University. We discuss related works in Section 9 and then conclude the paper in Section 10.

2 Grid Networking and Firewalls

Software running on Grids, which may be part of the Grid infrastructure or of a user-level application, can use the network in a number of ways. Software and utilities at the infrastructure level may use the network for job or data management transactions as well as Grid management tasks. Such operations typically follow a client-server model, where the client initiates a connection to the server and makes the appropriate request which the server complies with if all requirements are met. User applications may also involve network activities, including message-passing and data transfers.

For example, it is possible that a Grid application involves database operations. One of the components, C_{DB} , might be a database server listening for transaction requests. The client component, C_{CL} , might need to periodically send data down to C_{DB} through standard database transactions. Likewise, a user application may involve two components which share data, where one component must send periodic output to another component for processing. A workflow application will require that the output of one component is sent down to a downstream computation as the latter's input. Regardless of what sort of Grid software is using the network, firewall issues may arise where data communications of any type becomes impossible. In order to resolve such issues, three things must be understood properly: the nature of Grid networking, the nature of firewalls, and how software components on the Grid communicate.

2.1 A Grid Network Model

Figure 1 shows to the right the Grid architecture as described in a Grid anatomy by Foster et al. [8]. The Internet layered protocol architecture consisting of the Transport, Internet and Link layers is shown against a Grid architecture, but the correspondence needs to be clarified. There is no distinction here between connectivity for the Grid and connectivity at the fabric layer for non-Grid networking. We therefore present a two-layered stack formed by the *fabric connectivity* layer and a *Grid connectivity* layer above it. Note that the latter corresponds with the *connectivity* layer as

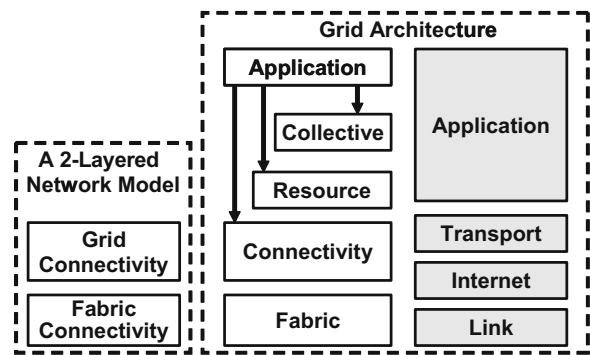


Fig. 1 The two blocks at the left form a two-layered model of Grid networking. The original Grid Architecture comes from “The Anatomy of the Grid” [8]

described in the Grid anatomy [8]. However, this two-layered network model highlights the difference between fabric-level networking and Grid-specific networking. Doing so allows us to distinguish between the nature of networking and firewall issues at each of those levels.

The bottommost fabric connectivity layer is the pre-existing communication infrastructure that interconnects the Grid fabric, typically designed and administered independently of the Grid infrastructure. Fabric connectivity is not managed on the Grid, although there is interest to merge the two levels of networking for purposes of optimization. The Grid *connectivity* layer involves Grid-specific functionality to support Grid protocols and standards, e.g., for authentication and authorization. For this reason, the illustration shows the Grid connectivity layer as overlapping the collective, resource and connectivity layers of the original Grid anatomy. We are not concerned at this point with a rigid mapping between network operations at the Grid connectivity layer and the corresponding operations at the fabric connectivity layer below. What matters is that we can separately analyse and deal with the firewall issues that come up on each layer.

Many organizations that operate Grid environments have underlying fabric networks that pre-date Grid computing, and Transmission Control Protocol (TCP)/Internet Protocol (IP) is probably the most preferred infrastructure [3]. TCP/IP is a protocol stack or suite that consists of the *transmission control protocol* and the *Internet protocol*

beneath it. Grid connectivity does not necessarily require TCP/IP implementations only, but its prevailing popularity for local area as well as wide area networking has made this the typical case. The Grid infrastructure is commonly a middleware layer on top of existing computing platforms, and TCP/IP is the common underlying network infrastructure. Our work is mostly confined within such implementations.

TCP/IP networking largely based on a *client/server* paradigm [3], with the basic notion of clients initiating the communications by sending a request for a service to a server, which, if appropriate, complies with the request. Much of the services offered over the Internet use TCP, which provides reliable, connection-oriented data transport between hosts that are identified by IP addresses. A different class of applications use another transport standard, the connectionless *user datagram protocol* (UDP), which does not guarantee delivery of data, but provides a simpler and faster mechanism [3]. Both TCP and UDP use the concept of *ports*, which are access channels, presumably available and accepting connections. A single Internet “host” machine may support many ports, and these ports may be bound to available services. By standardizing to a set of well-known port assignments to well-known services, it becomes easier to access services on the network.

We define a network domain D_i as being made up of nodes $n_1, n_2, \dots, n_y$, which are either service clients or service providers (or servers), interconnected over the network. A connection is made to each node using a certain protocol θ and a given destination port number $1 \leq p_k \leq 65535$. Assume for this discussion that we are only concerned with application-level protocols, e.g., SSH, TELNET, SMTP. We also assume that available ports are between 1 and 65,535. The parameters for a connection K are therefore the source address, the destination address, a source port number, a destination port number, and a protocol type. A connection over such a network may therefore be identified at any given time by a quintuple, $K = (n_i, n_j, p_k, p_l, \theta)$, which identifies both ends of a connection, by IP address and port number, as well as the protocol in use.

While Grid connectivity involves Grid-specific services, ports and protocols, they are still imple-

mented on top of TCP/IP. Basic Grid services will therefore involve connections that are still identified via a quintuple $K = (n_i, n_j, p_k, p_l, \theta)$, where the protocol θ is Grid-specific but nonetheless based on TCP/IP. There are Grid-specific considerations involved concerning when a service may be granted. This is primarily about verifying the identity of the user behind a client request and the authorizations associated with that user for the requested services.

We may therefore distinguish between network usage at the Grid connectivity layer and at the fabric connectivity layer by determining if the protocols and ports involved are Grid-specific or not. This distinction is relevant in later distinguishing between security policies specific to Grid-related communication and those that are not.

2.2 Firewalls

Security policies implemented by firewalls will be typically segmented into logical partitions. A subnet within a larger network may be reserved for privileged users, whose access rights translate to lenient firewall restrictions. Policies may reflect departmental or institutional concerns, and so there may be departmental as well as institutional firewalls operating independently of each other. In the same way, security policies may concern fabric connectivity issues, which exist independently of Grid connectivity concerns. Those concerns do not always harmonize with each other. In some cases, a global or institution-wide security policy may conflict with a local or departmental security policy. In the same way, since the Grid reflects a policy of sharing, there may be conflicts between an institutional concern to limit access to resources and a local policy to open up resources for sharing on a Grid.

For example, all database access may be intended for authorized use only within the institution, but certain databases may subsequently be intended for sharing with authorized partners on the Grid. It is likely in this situation that all database servers are protected by the institutional firewall, which screens all externally initiated connections. This security policy is therefore implemented at the fabric level, while Grid-specific means of controlling or influencing accessibility

may not be available, i.e., the firewall is not a Grid resource that can be managed as such.

Another example that illustrates lost potential caused by firewalls is when applications on the Grid are physically allocated across different institutions. Communication across those networks may be harder for user applications because they do not use standard port assignments. A port is an access channel across a network, and port numbers for well-known services, e.g., telnet, SSH, FTP, are practically standardized. System-level software are likely to use those ports, and firewalls may admit access to those ports for standard services, including Grid services, e.g., job services and data transfers. While having firewalls open certain ports at the fabric level is not always welcome, opening up a few ports is less risky than opening up several. However, defining a range of several possible ports may be the only solution for user applications, whose port assignments may not be known until runtime.

Working towards Grid connectivity need not require changes that may compromise security policies at the fabric level. We contend that there may be ways to provide Grid connectivity even in these situations. To engineer a solution requires that we first analyze specific cases of fabric-level firewalls that block Grid connectivity.

Typical firewall restrictions operate by checking connection destinations against a list of open, closed and filtered ports. This applies to inbound, outbound, and forwarded connections, relative to the protected domain. Such firewalls are among the easiest to set up and among the most common. In many, if not most, operating systems, a firewall will be installed on the installation machine by default. If such a firewall was blocking all incoming TCP connections to port 5,432 on the database machine from outside the network, there is no chance to make a connection from outside.

Let us consider the parameters for such problems. We earlier gave a simple definition of a connection as a quintuple, $K = (n_i, n_j, p_k, p_l, \theta)$, which identifies both ends of a connection, by IP address and port number, as well as the protocol in use. Security policies regarding connections to a given node or its network translate to a finite “list” of allowed and disallowed connections. Note that there are different and independent lists of

security restrictions prevailing for each firewall in a given VO. It is common to find several firewalls within the same organization, e.g., for each network or local firewalls for some nodes. Combining the various lists of the firewalls traversed makes for a potentially narrow window of accessibility. To simplify each situation between two nodes that need to connect, we focus on a single list which is a result of compounding the security policies across the two nodes and all networks between them. Our concern is in specific connections to port p_j on node n_B , i.e., $n_B.p_j$, from some source port on node n_A . This list includes ports that are simply open but not bound to any pre-existing service, or those that are through some specific protocol. Note that we use the term “connection” loosely in this paper. We do not necessarily imply a connection-oriented protocol such as TCP. This may also include UDP, which is connectionless. We say that a node n_A is connected to port p_i on node n_B via protocol θ when successful transport-level communication, e.g., TCP or UDP, begins and persists for the duration intended.

In a generic sense, whether at the fabric or Grid connectivity levels, successful connections require three things. First, there must not be any security restrictions in place that will block connections from n_A to n_B on port p_i . Second, the connection should have been initiated by a component c_A and accepted by component c_B . Finally, it is necessary that both c_A and c_B can speak the same language: protocol θ . The first requirement is generally understood to be a function of the network security policies, implemented by a firewall. The second requirement on component interaction is trivial: one component must initiate a connection that another component accepts since the point-to-point connection is existentially dependent on two peers. The last requirement is common sense: communication requires a communication protocol agreed upon by both parties.

2.3 Grid Software and Firewalls

In order to generalize the problem, we must be clear about the nature of Grid software communications and how they are affected by firewalls. Figure 2 shows a generalized model, where fire-

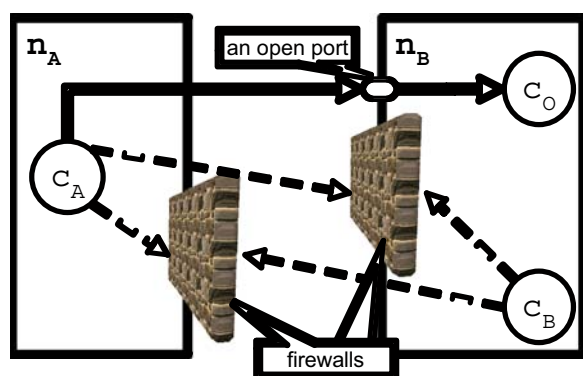


Fig. 2 The four dashed arrows show four different paths that may be blocked by firewalls when C_A and C_B try to communicate using sockets. In contrast, C_O is accessible via an open port which is never blocked by a firewall

walls situated on two sites may each block outbound and inbound connections between C_A and C_B . The exception is C_O , which listens behind an open port that is not blocked by a firewall. The situation involving the former two illustrates a worst-case scenario, where direct socket connections are blocked in both directions each time. REMUS was designed so that such situations can be dealt with in various ways. We illustrate this by investigating the case of Nimrod/G [1, 2] and Globus Toolkit [9] in such a scenario, where REMUS must assist in outbound connections as well as inbound connections for both sites. Nimrod/G is particularly apt since Nimrod agents must be pushed out from the Nimrod/G site (outbound) to computational nodes, which must accept the launch request (inbound). Once executing, the agents must then contact the Nimrod/G site from the computational node, and Nimrod services must be accessible (inbound) so that the agents may pull down experiments and experiment data. It is thus sufficient to illustrate REMUS using Nimrod/G since it provides us with the worst case to investigate. But what about UDP transports? The above scenarios do sound more applicable to TCP streams, which are connection-oriented. UDP traffic is connectionless, but they are treated in almost the same way by firewalls. Most Internet services are TCP-based, so that firewalls are more likely to restrict TCP connections. On the other hand, some services are accessible using UDP,

and these may also be restricted by firewalls. Such restrictions take the same approach, identifying ports and IP addresses that are to be either accessible or blocked. REMUS is designed to also redirect UDP traffic, since the rerouting mechanisms we use are able to tunnel UDP traffic. However, we did not include such tests in this paper since, in our experience, most Grid-based transactions are carried over TCP.

3 The Globus Toolkit and Nimrod/G

This section briefly introduces the Globus Toolkit [9] and the Nimrod/G suite of tools [1]. Both are used as case studies in order to demonstrate that typical Grid connectivity can be facilitated through REMUS. Note that, while much of what we will discuss would apply to several versions of the Globus Toolkit, we only used version 2.4 (GT2.4) for our experiments. We provide some discussion about newer versions of the Globus Toolkit in the next section. These versions seem at first glance to be less affected by firewall issues, but the situation is actually unchanged. As for Nimrod/G, much of what we present will be specific to Nimrod/G 3.0 agents over GT2.4 resources, but again, the situation is no different from the later versions of Nimrod/G, regardless of the Globus Toolkit version underneath.

3.1 Case Study 1: The Globus Toolkit

The Globus Toolkit [11] provides various client and server tools to manage Grid resources. It has become a de facto standard in building Grid testbeds, thus providing us a good case study opportunity. Figure 1 illustrates how the Globus toolkit provides the connectivity, collective and resource layers of the Grid. It facilitates operations over underlying resources, i.e., the Grid *fabric*, and tools for job management and data management. Globus typically deals with a pre-existing resource manager, which manages several physical resources as one resource on the Grid, e.g., a queue manager for an entire cluster. Connectivity with and among Grid resources is supported by several protocols in order to build connections, authenticate users, authorize

access and delegate tasks. Security-related connectivity is provided within a *Grid Security Infrastructure* (GSI) which is based on public-key cryptography. Grid resources, which may be abstractions over the fabric, may be accessed using a number of protocols, including the HTTP-based *Grid Resource Access and Management* (GRAM) protocol and GridFTP for file transfers. GRAM controls the allocation of resources' compute nodes and the monitoring of computations allocated to them. GridFTP is an extension over the standard File Transfer Protocol (FTP) with GSI mechanisms and enhancements that include the ability to use multiple data streams. The two server-side components that facilitate GRAM and GridFTP are the *Gatekeeper* and the *GridFTP server*. The Gatekeeper accepts submissions for computational jobs and allocates them to the appropriate *job managers*. In some cases, job managers will deal with autonomous resource managers, rather than directly with the resources themselves. For example, some job managers will deal with particular types of batch submission systems. The job manager will also return status data and the job output to the GRAM client. The GridFTP server provides data transfer mechanisms, facilitating the transport of files to or from some remote source. GridFTP supports multiple parallel streams to maximize throughput, and operates within the GSI framework for security purposes.

Figure 3 illustrates some of the Globus Toolkit components in a snapshot of a running system. In this simplified scenario, a Globus client may either transact via job or file transfer operations. The Gatekeeper receives job-related requests and spawns off an appropriate job manager for that operation. While the initial traffic between the client and the Gatekeeper is in one direction only, traffic between the client and the job manager may be initiated in both directions. Data transfer operations are delegated to a spawned FTP server, one for each transfer request. This FTP server may open several ports, depending on how many parallel streams are requested, then data streams proceed in one direction, always initiated by the Globus client. Collectively, all Globus transactions will require that connections can be initiated in both directions.

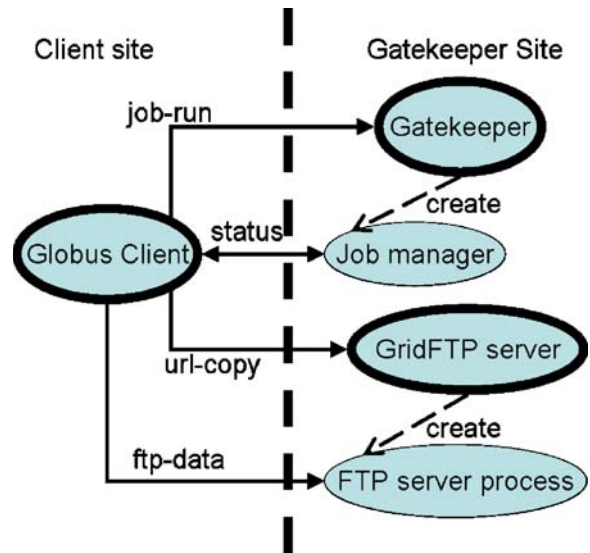


Fig. 3 This is a snapshot of Globus Toolkit components that communicate across site or network boundaries. A firewall that stands in the middle can easily block normal connectivity

The Globus toolkit firewall requirements document [28] defines certain access settings that are necessary to facilitate successful Globus transactions. Apart from the service ports for the Gatekeeper and GridFTP, several ephemeral ports (above port number 1024) must be accessible for incoming and outgoing TCP traffic. Note that the Gatekeeper and the job manager, from the Gatekeeper site, may initiate connections back to the client site. This *callback* functionality may require client sites to permit incoming traffic to up to 10 ports for each expected simultaneous user. At the server site, 20 ports are recommended [28]. For data transfers, multiple connections will also be initiated, from the GridFTP server, back to the client site. Having several new TCP streams that follow the initial one, proceeding from both directions, makes firewalls more difficult to handle.

3.2 Case Study 2: Nimrod/G

We look at Nimrod/G as a good case study of a Grid application. It is typical of applications that execute jobs on Grid resources using standard job and data transfer operations. **Nimrod** is middleware that sits above and harnesses the computa-

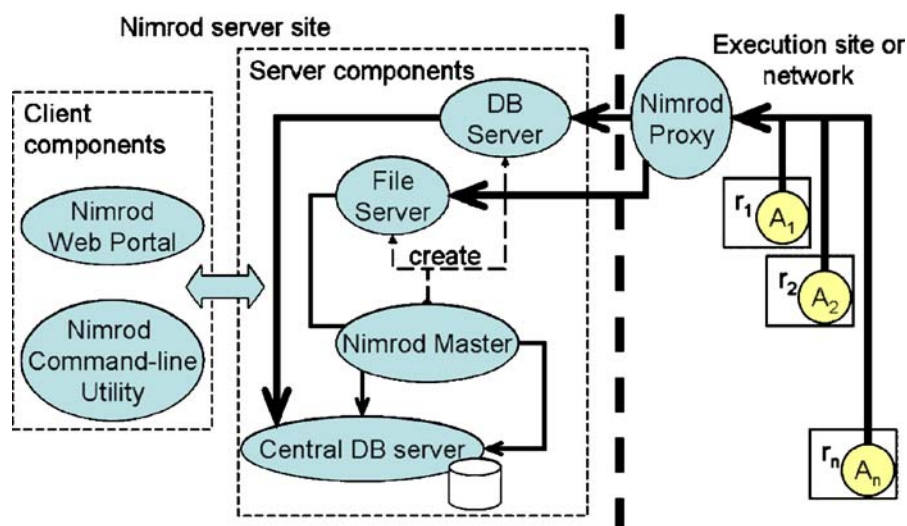
tional power of several loosely-coupled processors to execute experiments [1]. Nimrod experiments are *parameter sweep applications*, involving tasks that process a large range of parameter combinations. Areas where parametric applications are useful include design and engineering, as both areas benefit from the simulation of their target product over several different situations to study the effects of changing parameters, e.g., weather and environmental conditions, energy characteristics, load, etc. [1, 24].

Nimrod/G is the Grid-enabled version of Nimrod, consisting of client components, server components and computational components. Client components include command-line or web browser interfaces as well as the tools and libraries they need to define and manage experiments. Server components are at the core of managing parameter sweep applications, including the scheduler, a centralized database server and a *Nimrod master* to manage the other Nimrod components and the interaction with Grid resources. Computational components run on resources to manage the execution of jobs as required by Nimrod experiments.

When an experiment is started up, an *experiment master* is created, which will invoke a new *file server* as well as a *database server* if they don't already exist. The file server handles file transfers while the database server handles control data for executing jobs. The appropriate *ac-*

tuator for the chosen Grid resource must then submit the correct agent binary as a job to be executed. This is determined by the resource's architecture, kernel and operating environment version. If the binary already exists on the resource, it is simply reused. When it starts up, it will connect to the Nimrod central database to verify its "marching orders", e.g., jobs to execute, input files to download, and whether or not it runs as a *Nimrod Agent* or as a *Nimrod Proxy* on the head node. The task of an agent is to run the experiment using an assigned parameter set. By farming out several agents across Grid resources, Nimrod/G can parallelize the parameter sweep and reduce the overall execution time across the entire range of parameter sets, each processed as manageable tasks. The Nimrod Proxy is, as the name implies, a proxy utility to centralize the communication between the agents and the file and database servers. The agents, directly or through the Nimrod Proxy, talk to both the database server and the file server throughout the experiment until the results are sent back. This necessitates a connection from the agent to two distinct ports behind which the two Nimrod servers will listen [1]. Figure 4 illustrates a snapshot of Nimrod/G components at work as they manage an experiment. The dashed line running down the middle signifies network boundaries. There are client components and server components. Note that Nimrod job submissions and data transfers

Fig. 4 This is a snapshot of Nimrod/G running across two sites. The interconnection is necessary for Nimrod to manage the agents (A_1 through A_n) on remote compute resources (r_1 through r_n), which in turn manage computational jobs and data transfers that they might require



via Globus will be treated as transactions with a Globus client. For example, Nimrod submits experiments as jobs using `globusrun`, which is a Globus client utility. However, as far as Nimrod is concerned, the server components of Nimrod are service providers. To the left of these components in the diagram are client components. At present, we are actively using two: the portal that can be accessed via web browsers, or a command-line utility which provides the same client functions.

Communication between the Nimrod agents (on the right) and the file and database servers (on the left) go through a Nimrod Proxy in this particular scenario. Connections from the Nimrod Proxy, or from the agents if the proxy is not deployed, are all outbound, initiated by the agents. Interconnectivity among Nimrod components across a Grid testbed must be bidirectional. The initial connections to Grid resources to deploy or invoke the Nimrod agents and proxy require outgoing traffic initiated by Nimrod as Globus GRAM submissions. During the processing of an experiment, Nimrod agents must be able to make outbound connections back to the Nimrod database and file servers, for the effective management of experiment tasks.

4 The Problem with Firewalls

The primary purpose of firewalls is to protect resources or networks by blocking particular incoming or outgoing communication across the network. This presents us with a conflict between the barriers that firewalls must enforce and the accessibility which Grid middleware take for granted.

4.1 A Classification of Conflicts

In an earlier work, we considered three cases of hindered communications between two components communicating via the network [25]. A *node conflict* occurs when security restrictions block connections to the target node at any port. A *port conflict* occurs when security restrictions block connections to a particular target port only. We may also find that, while a port conflict exists, there is instead another vacant port available. In

such a situation, the connection still cannot be made to the intended component until it binds to that vacant port or use some bridging mechanism that will facilitate communication. We can also encounter a *protocol conflict*, when a successful connection can be made to the node at a given port but the service listening behind that port is unable to communicate using the protocol expected by the connecting component. Grid middleware like the Globus Toolkit and Nimrod/G are dependent on the network. Both assume that it is possible for various components running across remote machines to interact across the network using multiple ports and protocols. Any of the conflicts mentioned above can disrupt network activities of the Globus Toolkit, Nimrod/G, and Grid applications in general.

4.2 Firewalls and the Globus Toolkit

While it is possible, firewalls are usually not situated between Globus toolkit components and computational resources. However, firewalls can block connections in one or the other direction between the Globus client requesting for services and the Globus service components waiting for requests or processing them. Certain ports must be accessible through the firewall in order for the toolkit to function properly [28]. To begin with, ports to access Globus services must be accessible, including those for the Gatekeeper, the GridFTP server, and additional ports that are required at runtime, which can be set via the `GLOBUS_TCP_PORT_RANGE` variable. If connections to these ports are unsuccessful, GRAM and GridFTP operations cannot proceed. The GRAM service may also use *callbacks* to the client for status notification. Unless incoming connections to the clients are allowed within a certain port range, GRAM callbacks will fail. GridFTP has its own unique requirements. For the sake of performance, the Globus toolkit uses many ports for parallel streaming. However, in a secure environment, the number of open ports is deliberately kept at a minimum, which would rule out the port range configuration.

Even for the newer versions of the Globus Toolkit, which have been available for a few years now, the problem with firewalls has not

disappeared. In the newest configuration guides describing how to deal with firewalls, the same limitations prevail [28]. Much of the changes in the GT3 and GT4 pertaining to connectivity simply deal with adopting web service standards. Hence, job initiation and management use web services on new port assignments, but requirements for multiple ephemeral ports remain. For example, event notification and file staging still proceed from server-side ports to client-side ports. File transfers using GridFTP do not change at all. The bottom line is that a fair number of ports must still be directly accessible on both sides. Approximately 10 ports must be accessible at the client site for each simultaneous user, and up to 20 ports must be accessible at the server side for each simultaneous user. This remains undesirable where the client and server sites are required to maintain strict network security policies.

4.3 Firewalls and Nimrod/G

For Nimrod/G, firewalls can get in the way between the computational resources and the Nimrod server components (see Fig. 3). When Nimrod/G uses a Grid resource for the first time, it must first transport a copy of the Nimrod remote binary to the computational resource. This binary encapsulates both the Nimrod agent and the Nimrod Proxy. If the firewall blocks the transfer, then Nimrod may not proceed with using the resource at all. If the binary is present, submitting experiments is a matter of initializing the agents on a given computational node for a particular task over a specific parameter set. This invocation is usually through a normal request to execute a job, e.g., through a queued submission. The firewall can get in the way and Nimrod will not be able to use the resource if this request is not received by the intended job manager. Assuming it is initiated, the agent must connect back to Nimrod server components to obtain control and input data. The agent must talk to both a database and file server, so it must have access to two distinct server-side ports on the Nimrod site where the two servers accept requests. The firewall protecting the computational resources may block these connections, making it impossible for the agents to receive their tasks for processing. We may deploy

a Nimrod Proxy to centralize all connections from the agents to the Nimrod servers, but the firewall may likewise block connections from the Nimrod Proxy. It could be deployed elsewhere, but the agents must be able to reach it and it must be able to reach the Nimrod server components. Furthermore, during the processing of the task assigned to it, the agent sends “heartbeats” to Nimrod, “I’m alive” status updates which the Nimrod master must keep track of in order to manage the active agents. If the heartbeats are blocked by firewalls and are not received by the Nimrod server components, the agents are assumed to have failed, and their jobs must be restarted. Finally, assuming that the experiment is completed, the agent must be able to transmit its output back and perhaps receive additional tasks to process.

5 Rerouting Via REMUS Using Wrappers

We earlier mentioned our classification of firewall issues as port, node or protocol conflicts. We previously addressed some of these problems with ad hoc solutions based on what channels were allowed, using whatever mechanism was appropriate [25]. However, the philosophy behind REMUS is not tied down to any single firewall traversal mechanism. REMUS advocates a multi-pronged, unobtrusive, rerouting, middleware solution: In our design, Grid applications should have access to *rerouters*, wrappers that will intercept normal communications and get them across to the intended party using whatever means available. Several traversal mechanisms are appropriate at different situations, and REMUS should be capable of as many of them as possible. Figure 5 illustrates the abstract model by which rerouters bridge connections between components that seek to communicate despite the presence of firewalls between them. Two communicating components C_A and C_B are separated by one or more firewalls that may restrict direct communications between them. Rather than force the reprogramming of the components or the firewalls, we advocate a wrap-and-reroute approach. Two rerouters ρ_A and ρ_B are shown to be wrapped around each communicating component. All network connections will be intercepted and rerouted as appropriate. There

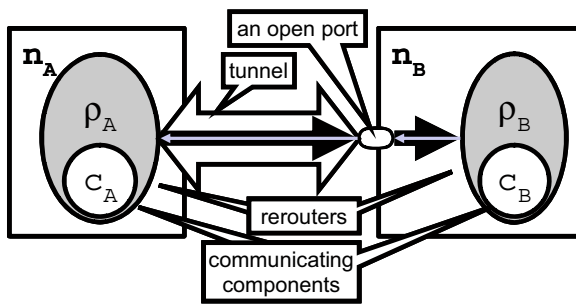


Fig. 5 The abstract model of REMUS illustrates the two central ideas of diverting connections unobtrusively via wrappers and rerouting

is a primary tunnel shown as a bi-directional block arrow between the two nodes. This tunnel will cross several nodes, networks and firewalls, if necessary, and may involve several proxy hops. The tunnel may also feature compression and security features, depending on what is required or available. Forward and reverse connections, shown as black and pale arrows in opposite directions, are connections between the components that are carried over the rerouting tunnel.

The basic REMUS architecture addresses the classification of conflicts earlier mentioned in a straightforward manner. In a vacant port situation, since direct connections to the vacant port are possible, then we position rerouters on the vacant ports as required. If there is a port conflict with an existing daemon, then rerouters will be deployed, if possible, as plug-in or extension components for the pre-existing daemon. For example, we may deploy a web service implementation of a rerouter. In a node conflict situation, where one or both nodes cannot be accessed via any port, we can use a proxy approach where rerouters must work with one or more rerouters sitting on mutually accessible third-party proxy nodes. If necessary, a proxy chain is formed with however many hops are required to establish the tunnel. Rerouting is not possible in cases where the firewall restricts connections to or from one of the nodes completely. This is true not only for REMUS but for any solution that can be devised. Tunneling is not possible unless one port is at least accessible for establishing the tunnel. In such a situation, the only recourse is to lift restrictions

on the firewalls or middleboxes where they are implemented, i.e., opening *pinholes* [21].

6 Tunneling and Port Forwarding Techniques

Firewalls and similar middleboxes have become commonplace appliances on the Internet and within organizational networks. The restrictions they impose have prompted people, for different reasons, to seek various means of circumventing the restrictions, either openly or covertly. While the motivation behind such activities is at times questionable, there can be legitimate reasons for resorting to them. It is possible that a group of users, with authorization, wishes to explore an application, middleware or operating environment which requires network access that is not otherwise permitted by the firewalls already in place. It may also be a preference to leave the firewall configured as it is, and institute an independent access mechanism that may be more controllable and flexible than the firewall. Security policies implemented by firewalls are considered to be part of the infrastructure, and applications seeking to use the network legitimately for authorized purposes may simply have to work with them. Here we mention some of the techniques and tools that have been devised to work around firewalls.

SSH is commonly used for tunneling, and has been recommended for Globus firewall issues [12, 28]. Many secure networks allow SSH traffic through because some form of login must be tolerated and SSH is highly secure. With pure SSH tunnels, the assumption is that the SSH service on the remote site is available and accessible. It is then possible to program the SSH client to use several pairs of mapped ports. The local port is set up to listen for connections, which are then forwarded to the counterpart remote port on the remote destination host.

SSH may also piggyback through another transport mechanism if direct SSH connections are not possible. One that we tried was using the HTTP proxy or web proxy service to deliver SSH connections, assuming that the service was available and accessible. Proxies allow traffic to go physically around firewalls, by rerouting connections through a third party that is accessible as well

as authorized to send network traffic out across the firewall.

Whether or not we use SSH directly or via a proxy, port forwarding can only assist in getting the connection across in one way. Problems are encountered as connections are made in the other direction over multiple ports. It is not entirely possible to predict which ports will or should be used by new connections. The `GLOBUS_TCP_PORT_RANGE` is an environment variable that allows the Globus toolkit to restrict whichever ports *can* be used within that range, but that does not provide enough control. We may designate 1-to-1 port maps between several local ports and several target remote ports, but there is no way to determine which of the ports (and thus, its counterpart) will be used by a particular instance of a Globus or Nimrod/G component.

For example, when the GRAM client executes via `globus-job-run`, it will listen on specific ports, e.g., 65502, 65503. We assume that, due to the firewall, those ports cannot be accessed directly. Instead, the job manager will have to rely on a tunnel. For example, we can run the SSH tunnel so that ports 14000 through 14535 on the job manager site will map to ports 65000 through 65535 on the GRAM client site. The job manager must therefore use the local ports 14402 and 14403 for callbacks to reach the right target ports, 65502 and 65503, on the GRAM client site. However, the job manager only doesn't know about which local ports map to the right ports on the GRAM client site. After all, it only knows about those ports (65502 and 65503) because the GRAM client told it so, and the latter does not know about the port forwarding at all. Some other mechanism must set up the right port to map for the job manager, e.g., map local port 35502 to remote port 65502, and so on. One way to get around this problem is by submitting batch transactions, and results are cached until claimed. In this manner, it becomes unnecessary for the job manager to connect back to the GRAM client to deliver results or status information. The port through which the results can be obtained is known via a GASS-cache URL, e.g., <https://globus.csse.monash.edu.au:4102/12357/1153210197/>, but the given port, 4102, cannot be

reached behind the firewall. The request to access it must go through port forwarding, with an explicit URL that reflects the local port assignment [12]. This involves having to determine that URL after the batch submission is made, since the ports used by Globus are dynamically allocated, as are the GASS-cache URL strings. This illustrates a need for a flexible tunneling mechanism to hide the details of port forwarding.

The situation is slightly different with Nimrod. We must assume that Nimrod agents are deployed successfully via job submissions that go in the resource-bound direction. The port forwarding allocation of local and remote ports may then be predetermined via statically assigning the port forwarding, e.g., the local ports X and Y forward to the database and file server ports of Nimrod, respectively. On the other hand, with several agents running on multiple resources, it would be inefficient to create tunnels for each of them. Centralizing their connections back to Nimrod makes sense, through a Nimrod Proxy, which can sit on the front node, but the hosting node dynamically allocates the ports that will be allocated to the proxy. Normally, the Nimrod Master only learns of those ports when the Nimrod Proxy reports to Nimrod after successfully starting up. The Nimrod Master may then invoke the agents and include the Nimrod Proxy contact details when they are started up. Setting up the Nimrod Proxy on a static pair of ports reduces its flexibility, which comes from its ability to run as a normal job allocated by the hosting resource manager. For example, it is possible to bridge the gap between remote agents and the Nimrod server components through multiple proxies in a chain. In that scenario, each Nimrod Proxy sets up listening ports. Agents can use these ports directly or through other Nimrod Proxies, which can be deployed for intermediary roles as the need arises. It is always possible to “hack” a solution to the firewall issues described by patching up mechanisms here and there, as the situation allows, with the intervention of systems administrators. Grid application components may be rewritten slightly and firewalls may be altered as the need arises. It is far more desirable, however, to engineer a solution that abstracts the complexities of firewalls, tunnels, ports and protocols. For this reason, we developed REMUS to

be unobtrusive and geared towards using standard communication mechanisms through channels that are authorized by existing security policies.

7 The Implementation of REMUS

Figure 5 illustrates the principles behind REMUS. For one, the mechanism that provides rerouted transport is not fixed. Tunnels, after all, can be constructed in several different ways, e.g., IP over TCP, TCP over TCP, or UDP over TCP. In cases where the tunnel is transported using TCP or UDP, the protocol carrying the tunnel may also vary, e.g., SOCKS, SSH, web proxy. Whichever one is actually in use will depend on what the situation allows. In this section, we will provide details concerning the implementation of REMUS that assumes the typical, restrictive scenario:

- We have no direct connectivity to the Grid service ports (2811/tcp, 2119/tcp, 8080/tcp or 8443/tcp for Globus),
- nor to the ephemeral ports that the Grid software requires,
- but SSH is available.

We therefore create tunnels using SSH, and use SOCKS in order to invisibly reroute and multiplex connections across. This is not intended to be the only implementation of REMUS, but we consider this to be practical in this likely scenario where tunneling is still possible through SSH. It has the added benefit of encrypted transport for authorized users with explicitly granted accounts.

As explained in Section 6, it is undesirable to rely on hard-coded maps between the local port and the remote destination port, since the remote destination port cannot be pre-set. We therefore prefer a dynamic port-forwarding tactic, which supports multiple ports that are not predetermined until the ports are accessed. We rely on proxy-based solutions because direct connections are often blocked, so we prepare for the worst case that remains workable. We expect to use a third party proxy node that is mutually accessible to the communicating hosts. We have tested proxy solutions that use web proxy and SOCKS

proxy methods [12, 16]. The web proxy creates the connection on behalf of our rerouter, which sets up an SSH tunnel via the CONNECT command. However, web proxy servers often accommodate connections to port 80 (HTTP) or 443 (HTTPS) only, whereas we must create the SSH tunnel via port 22/tcp, Globus service connections via port 2119 and 2811, plus the ephemeral ports within the Globus TCP port range. A separate SOCKS proxy is therefore better since it is not restricted to only one target port, as it is intended to be a more generic proxy protocol for TCP and UDP [15, 16].

However, there are limitations. Institutions do not always provide either a web or SOCKS proxy for *incoming* connections. It is more common to find that only the SSH port, 22/tcp, is accessible on secure networks. For this reason, we favor the use of SSH as the underlying tunnel transport. On top of that, we still needed a way to intercept and reroute network traffic to use the SSH tunnels. For rerouting, we decided to use the open source *Dante* implementation of SOCKS [4]. It allows connections to be wrapped transparently within an environment that puts SOCKS functions ahead of normal network system functions. This approach results in transparently rerouted connections via SOCKS tunneled under SSH, with minimal reconfiguration of firewalls and Grid applications, if at all. This implementation with SSH and SOCKS, presents a clear and straightforward illustration of the wrap-and-reroute approach.

7.1 Setup and Configuration

Setting up the SSH + SOCKS implementation of REMUS is straightforward, even in the absence of an automatic or wizard-like deployment and configuration interface:

1. There is an assumption that user accounts are available on both networks, by which an SSH session may be established in both directions. This may be through a normal SSH connection with passwords, or with public/private key-pairs, or with X.509 certificates using OpenSSH with GSSAPI support [14].
2. The SSH connection is established with the preferred encryption and compression

settings, but dynamic port forwarding must be enabled. For example:

```
$ ssh -D 1080 username@remotenet.org
```

The switch “-D 1080” enabled SOCKS proxy services via the local port 1080. Since encryption and compression settings were omitted, then this session will use default settings. Settings which we recommend are “-o ‘CompressionLevel 9’” for maximum compression and “-c blowfish” to use the fast blowfish encryption algorithm. Other settings are possible.

Note that another SSH connection is required from the opposite direction if connections are also not possible otherwise. The assumption is the same: we assume that SSH is allowed through this direction and the necessary user account is available.

3. Dante must be available on both ends to provide SOCKS client features in order to reroute outgoing connections. A SOCKS client configuration file (socks.conf) will determine which outgoing connections may use the SOCKS connections. The rules express, in a straight-forward fashion, whether an outgoing connection to a given IP addresses and port will be facilitated through the tunnel, using which SOCKS proxy service. The appropriate proxy service associated with each rule is identified also by IP address and port number. Here is an example of a simple rule to forward connections bound for the subnet 172.17.171.0/28 via a SOCKS proxy running locally (IP address 127.0.0.1) from port 1080:

```
route {
  from: 0.0.0.0/0 to: 172.16.171.0/28
  via: 127.0.0.1 port = 1080
  protocol: tcp udp
  proxyprotocol: socks_v4
  method: none
}
```

Note that there are several possible options in how the configuration may be given. For example, whether SOCKS v4 or v5 are supported by the SOCKS proxy indicated, or

whether connections are authenticated by username.

4. The SSH clients on both ends provide outgoing SOCKS proxy services via the dynamic port forwarding port specified at invocation. This is the simplest approach, but a more secure solution is possible. Instead of using the SSH client to provide proxy services, a remote SOCKS daemon via Dante may instead be used. The Dante SOCKS daemon has its own configuration file (sockd.conf) with rules that determine which incoming connections will be permitted. Using a full SOCKS daemon such as Dante has the advantage of effectively putting an inner firewall at the end of SOCKS tunnels. It is only necessary that the SOCKS client be configured to point to the Dante SOCKS daemon instead of the SSH client. Here is an example of configuration rules for the Dante SOCKS daemon:

```
logout: stderr
internal: eth0 port = 1080
external: eth0
method: username none #rfc931
user.notprivileged: nobody
user.libwrap: nobody
client pass {
  from: 127.0.0.1/32 port 1-65535
  to: 0.0.0.0/0
}
client pass {
  from: 127.0.0.0/8 port 1-65535
  to: 0.0.0.0/0
}
client block {
  from: 0.0.0.0/0 to: 0.0.0.0/0
  log: connect error
}
pass {
  from: 127.0.0.1/32 to: 172.16.171.0/28
  protocol: tcp udp
}
block {
  from: 0.0.0.0/0 to: 0.0.0.0/0
  log: connect error
}
```

This configuration allows local connections only, which makes sense if the SOCKS client connects through an SSH tunnel from a

remote site. This setup is ideal to give local administrative control over the SOCKS connections. Note that the final “pass” socks-rule permits connections intended for nodes in subnet 172.16.171.0/28, which we may assume to be a network of Grid resources. The SOCKS daemon must therefore have network access to those resources. The daemon itself may be situated wherever preferred, as long as the clients can access it either directly or indirectly.

5. Grid components must be invoked in such a way that the SOCKS libraries of Dante are preloaded. This is how network connections are implemented using SOCKS rather than as normal socket connections. In Linux implementations, this is done by appending the SOCKS library file, e.g., `/usr/lib/libsocks.so`, to the environment variable `LD_PRELOAD`.

For convenience, we used shell scripts that initiate the SSH tunnels and generate the SOCKS configuration file:

```
#!/bin/bash
LOCAL_SOCKS_PORT = $1
SSH_USER = $2
SSH_SERVER = $3
SSH_CLIENT = /usr/bin/ssh
#
## Loop will force a restoration
of the tunnel in case it drops
out
#
while [1]; do
#
## High compression, fash
cipher, and set up SOCKS proxy
at requested port
ssh -N -o 'CompressionLevel 9'
-c blowfish -D $LOCAL_SOCKS_PORT
\ SSH_USER@$SSH_SERVER
done
```

These scripts, which execute under the bourne-again shell environment (bash), are straightforward and quite trivial. The creation of an SSH tunnel simply requires the preferred SOCKS proxy port as input. This

same port number is input to the SOCKS configuration script, given that it must specify which local port to use for a specific destination. Typically, this is a local port, so that it is only accessible from within this same host. In other situations, it may be a port on another machine that is always accessible, e.g., an officially designated proxy server. In our tests, we only had one local destination for each SOCKS configuration, since we only involved two Globus hosts in our tests, one acting as a client, and the other as a server, and one proxy for all connections in that direction (client to server). In advanced configurations, the scripts would accept a range of port numbers, and would create several SSH tunnels and configure SOCKS to use several SOCKS proxies. The scripts are the same, either way, differing only in the range that we supply. Since we used OpenSSH without GSSAPI in our tests, it was necessary to first launch and authenticate the SSH agent, allowing us to make subsequent SSH connections after having only supplied the passphrase once. Certificate-based SSH access is not standard for all SSH-capable servers, as it requires specific installation/configuration of SSH services. If it were available, then the once-off authentication via the SSH agent is replaced by the generation of one proxy certificate which may be valid for several hours or days. In generating one proxy certificate, authentication via a passphrase is also required once.

7.2 Globus and REMUS

We explored the efficacy of REMUS via SOCKS on the Grid by applying it to the Globus Toolkit and Nimrod/G. At the core of Globus transactions are job management and data transfer operations. We focused on how to support these operations despite the restrictions imposed by firewalls. Nimrod/G needed a slightly different solution. Nimrod agents invoked by the job manager on the compute nodes are network-aware and Grid-aware components that are responsible for pulling in computational tasks and data to process as part of a parameterized experiment. As with Globus components, firewalls will hinder the

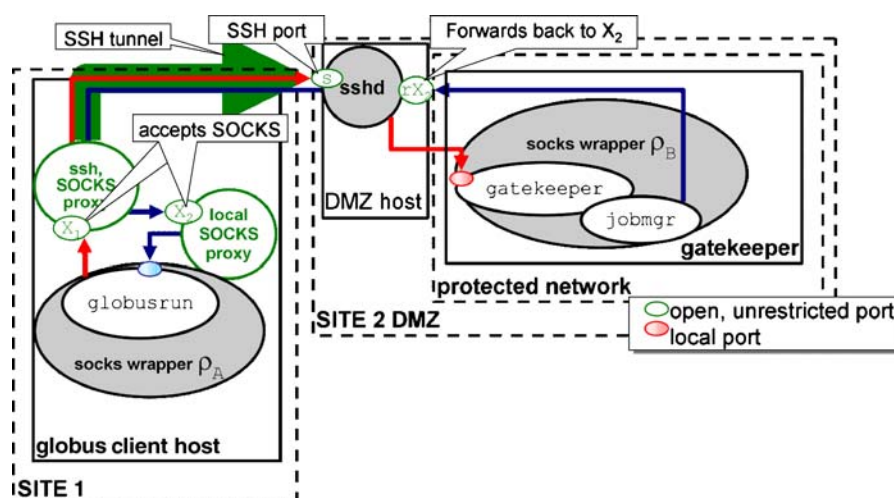
interconnections between Nimrod components, particularly where the agents must interact with Nimrod server processes. While it is possible to reprogram Grid applications in order to support rerouting across firewalls, there are considerable advantages in using the unobtrusive middleware approach.

Figure 6 gives an overview of how REMUS assists Globus Toolkit 2.4 (GT2). There are actually two levels of connectivity involved. First, there is the Globus connectivity level for Globus components. These connections are via TCP and typically use two service ports for connecting to the Gatekeeper and GridFTP services and several ephemeral ports within a certain port range. The latter ports are used on both the client and server hosts to restrict the ports that need to be open for incoming and outgoing connections. In order to facilitate the connections across firewalls, we must construct a second level of connectivity underneath, REMUS connectivity, which reroutes everything through a tunnel while remaining almost invisible to the Globus components. The rerouters invisibly come into effect through environment variables that need to be set and configuration files for the Gatekeeper and GridFTP servers that must include directives for those variables. The actual communication protocol to be used depends on the security policies in place, but we also considered which protocols can do flexible port rerouting. Our reference implementation of this technique was to use SOCKS wrappers

on top of (or within) SSH tunneling. Figure 6 illustrates the SSH tunnel as a side arrow that spans across from SITE 1 to the DMZ host on SITE 2. Note that connections to the Gatekeeper go through this SSH tunnel. The assumption is that SSH sessions are allowed between the sites involved. OpenSSH supports SOCKS as a proxy protocol, and SOCKS wrapping is implemented by “socksified” connections: all communication between components are transparently coursed through SOCKS instead of directly via normal TCP. The technique calls for components to load SOCKS libraries into memory, making the components operate within a socksified environment. This includes job managers and binaries invoked on behalf of the user by the job managers. The SOCKS-enabled behaviour is therefore inherited from the gatekeeper down to the bottom-most computation. At the client side, the same preparations make sure that Globus client binaries are likewise socksified. The result of this virtual environment is that all Globus components communicate via SOCKS without being explicitly invoked with hard-coded tunnel settings.

SOCKS connections are carried through the SSH tunnel connected to an authorized host, which is typically located at the demilitarized zone (DMZ) of one or both networks. The assumption is that the network security policy permits incoming SSH connections directly to the Globus server components or to the DMZ host that can access them. SOCKS proxy services are provided

Fig. 6 Globus connections are enabled across firewall restrictions via SOCKS wrappers. The SOCKS connections are in turn tunnelled through a single SSH session with port forwarding



by SSH, a feature included in OpenSSH [29]. However, since OpenSSH SOCKS connections only go in one direction, we needed two SOCKS proxy mechanisms. We created a second SSH session which tunnels SOCKS back the other way, although we can also use reverse port forwarding through the first tunnel to send connections back to where a local SOCKS proxy is running, as shown in Fig. 6 (see connection via port rX_2 to the local SOCKS proxy component on SITE 1).

7.3 Nimrod/G and REMUS

Nimrod/G behaves almost like a typical Globus client for Globus transactions, and uses Globus client binaries. Through SOCKS-wrapping, Globus transactions between Nimrod/G binaries and the resources running from the Globus resources are enabled across the SSH tunnel. However, instead of simply submitting computational jobs, Nimrod/G talks to networked components: Nimrod agents. These agents must be able to connect back to the Nimrod server components, perhaps past another set of firewalls. A proper configuration of the Globus job manager provides the SOCKS-enabled environment to the Nimrod agents when they are executed. Figure 7 shows how SOCKS wrappers around the Nimrod

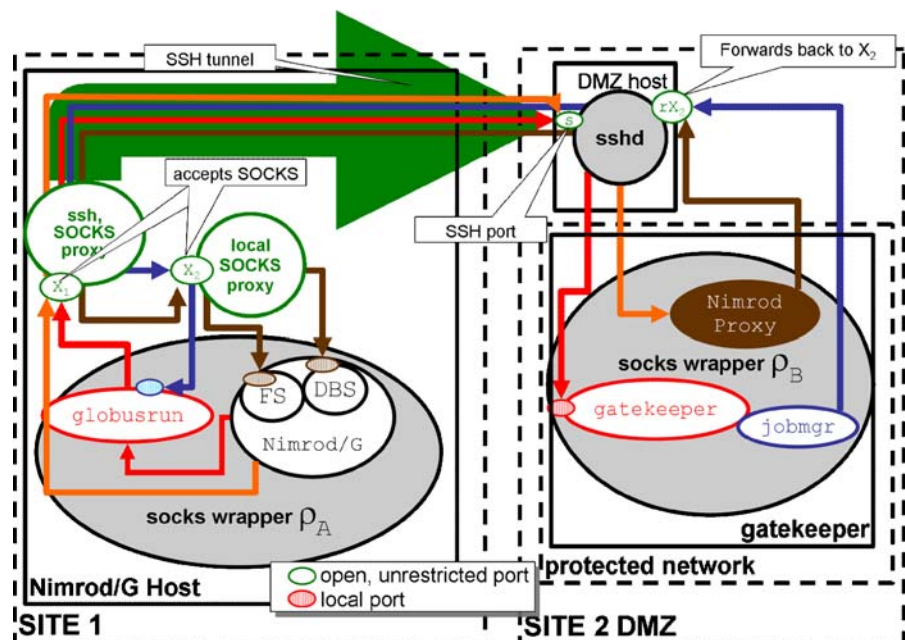
agents or proxy allow connections back to Nimrod/G File and DB servers (FS and DBS). The connection goes via reverse-direction port forwarding using port rX_2 on the DMZ host, through to a local SOCKS proxy on the Nimrod/G host.

The implementation and configurations described were enough to enable GRAM transactions, GridFTP copies and Nimrod/G experiments to successfully run over two networks separated by firewalls that restrict connections in both directions. There is a reasonable trade-off between total connectivity failure and optimal network performance. To illustrate, we ran some GridFTP experiments.

7.4 REMUS and Security

REMUS was designed in order to transport Grid data exchange across firewalls. Since firewalls are typically implementations of a given organization's security policies, then any ability to escape firewall restrictions must be considered carefully. It cannot be denied that REMUS presents a lowering of network security, but it is designed to do so only for purposes of Grid computing. REMUS is designed to be part of the Grid infrastructure. It must be administered as an application layer

Fig. 7 Nimrod/G server and agent components are connected via REMUS. The underlying connections are as in Fig. 5, except that Nimrod/G components are also wrapped around SOCKS



firewall, in the sense that Grid middleware and applications can be viewed as applications on top of the standard networking infrastructure.

In examining the security aspects of Grid applications using REMUS, we must consider three levels:

1. The Grid security level pertains to Grid-specific mechanisms and protocols that enforce authentication and authorization. Globus Grids, for example, validate all transactions using X.509 credentials that must be authorized by the VO's certificate authority. For example, there must be valid user, host, and proxy certificates.
2. The REMUS security level pertains to REMUS-specific mechanisms and protocols that we enforce in two ways. First, REMUS cannot be used arbitrarily to get into a secure network. Creating a rerouted path through a tunnel requires a valid, authorized and authenticated account in the remote site. In our SOCKS + SSH implementation, the tunnel is erected through an SSH session, itself requiring a valid account that must accept dynamic port forwarding. This implementation relies on strong authentication techniques used by SSH, such as the use of public key rather than symmetric key encryption. Port forwarding can be switched off for users who are allowed to remotely log in but are not authorized to perform port forwarding. Furthermore, while our current implementation uses SOCKS v4, future implementations will use SOCKS v5, itself requiring certificate-based authentication and authorization. The second way by which to enforce security in REMUS is by managing a "rerouting table". Like a normal IP routing table, the REMUS rerouting table controls the path to be taken by connections from X to Y. Unlike normal IP routing tables, however, apart from source and destination IP addresses, rerouting tables must also consider source and destination ports. The level of control is similar to that which is used for typical packet filtering, i.e., stateless IP firewalls.
3. Finally, security is still ultimately controlled at the network infrastructure level. We may consider REMUS to be a middleware that is

inserted between the Grid middleware and the underlying network infrastructure. This typically involves TCP/IP networking, and REMUS will not be able to enter a network if its firewall does not authorize entry. For example, using the SOCKS + SSH implementation, an SSH session must first be created in order to create the tunnel. This requires access to the SSH service on port 22, as well as an explicitly granted user account. REMUS is not intended to break firewalls by entering networks without authorization. It must be configured to use only authorized means, such as through a valid SSH session for the user.

There is no doubt that REMUS is lowering security to some extent. It may be subjected to abuse, perhaps by unauthorized users. However, the alternatives can be worse. As mentioned previously, Globus toolkit developers recommend opening ten ephemeral ports on the client site and 20 on the server site—for *each* concurrent user. We believe that REMUS presents a better alternative.

8 Performance Tests

While rerouted networking through REMUS is expected to be slower than direct networking, we decided to test our throughput anyway, to see how far behind REMUS performance might be. Note that these tests were conducted in an environment where SSH is the only channel open between the two networks. Therefore, while our performance might normally be considered poor, our results should be seen in the proper perspective. We are not claiming that REMUS improves throughput, but we do show that REMUS succeeds in providing connectivity where there was previously none possible (without removing some firewall restrictions).

8.1 Test Environment

Note that the details below are indeed for a specific implementation of REMUS. We noted earlier in the philosophy of REMUS that it is meant to be capable of different modes of connectivity.

These tests are for a specific mode that is suitable for typical secure environments:

- SSH is the only accessible service across, so we use SSH to create the tunnel.
- In order to multiplex several connections across the tunnel, we use a SOCKS proxy.

Therefore, when we refer to test details below in terms of REMUS, the details are actually specific to the SOCKS over SSH implementation—as far as the transport is concerned. The rerouting layer of REMUS, which is implemented.

The two hosts are about 11 km apart, connected via a combination of copper and fiber. The average round-trip time is 0.43 ms. Both hosts use network interfaces that operate in full-duplex at a theoretical throughput of 1,000 Mbps (Gigabit Ethernet). They are about two routers apart from each other, the average bandwidth between them is 94.4 Mbps, the average round-trip time is at 0.475 ms, and the average jitter at 0.1685 ms. On the other hand, our graphs are not meant to prove any sort of performance improvement. We are aware of how tunneling will introduce performance degradation, due to the aggregate delays introduced by encapsulating TCP (the stream we are tunneling) within TCP (the tunnel stream). The graphs are provided to compare stream-oriented (as against burst-oriented traffic) throughput using REMUS rerouting and tunneling against the theoretical ideal, in which connections are made directly. The point is simply that REMUS does introduce throughput degradation, but only by as little as 25% from the best case we've encountered. It must also be pointed out that the tests were conducted on a live network in the presence of typical heterogeneous traffic (as can be found in a multi-campus university).

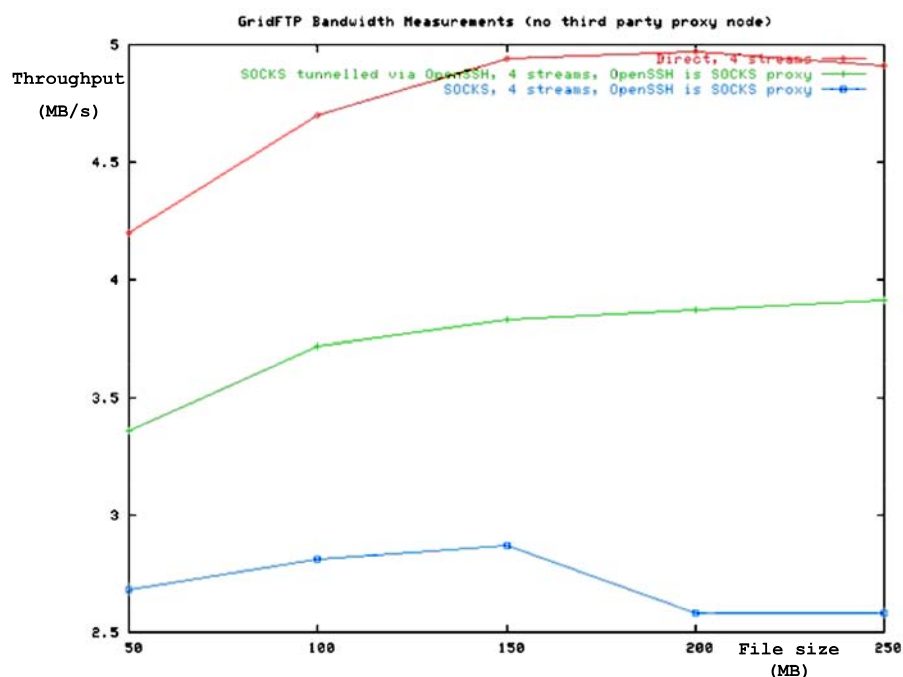
8.2 Results

While the two hosts are within the same organizational domain, they are connected over a wide-area network. Figure 8 and Table 1 show how data transfers will slow down due to the rerouting that REMUS uses to divert connections across the

firewall. Measurements are in Megabytes per second, computed from the size of the file transferred and the best time taken from several transfers. We performed tests to transfer files with sizes of 50, 100, 150, 200 and 250 Mb, all containing binary data. This is significant when considering that it may reduce the effectiveness of compression in the transport.

1. *Top Graph, Mode1: Direct.* The topmost graph represents direct GridFTP transfers. Four parallel GridFTP streams were used. This mode of transfer, with the firewall temporarily relaxed for purposes of this study, presents the best performance, with almost 5 Mb transferred per second.
2. *Middle Graph, Mode 2: REMUS Rerouted and Tunneled.* As with the previous graph, there are four parallel GridFTP streams for each file transfer at the application layer. At the transport layer, only one TCP stream is actually used across the network by the tunnel. SSH internally uses individual “channels” for fcs presents our best case so far, nevertheless suffering a 25% reduction in network performance compared to the best case (direct). Due to the multiplexing, plus the latency introduced by intercepting connections, rerouting them, and encrypting the streams, transfer rates shrink to about 3.8 Mbps.
3. *Lowest Graph, Mode 3: REMUS Rerouted Only.* As with the first case of direct transfers, there are four parallel GridFTP streams at the application layer. There are also four actual TCP streams at the transport layer. Here performance is at its worst among the three, peaking at only 2.8 Mb transferred per second. That Mode 2 performs better than Mode 3 can be attributed to OpenSSH. Both modes use SOCKS, equally suffering from the same overhead of SOCKS encapsulation, i.e., headers, SOCKS request/response dynamics. On the other hand, whereas Mode 3 uses a SOCKS proxy on the remote site, Mode 2 situates the SOCKS proxy locally. In effect, the SOCKS encapsulation overhead bytes are not transported in Mode 2. This can be gleaned from the source code of OpenSSH (we examined 4.7p1). Accordingly, the application

Fig. 8 Throughput graphs for Direct, REMUS (SOCKS + SSH tunneling) and pure SOCKS, in that order from the *top* to the *bottom* graphs, in terms of megabytes per second of data transferred using GridFTP



data encapsulation efficiency of SOCKS is at about 84% [17]. However, for each of the four GridFTP streams, the actual efficiency of what the tunnel will transport in Mode 2 goes up closer to 100%, since OpenSSH does not need to send the headers down the SSH stream. Additionally, our tunnel was configured for encryption.

Using SOCKS through OpenSSH thus presents our best case so far, nevertheless suffering a 25% reduction in network performance compared to the best case (direct). There may be other ways to reduce the degradation. We have already tuned the SSH parameters in our experiments with a faster cipher, blowfish, but there may be faster cipher implementations that we haven't tried. SOCKS v5 may yield better performance at the rerouting stage compared to SOCKS v4. Other methods to optimize the connections might also minimize the degradation, and they can be the subject of further work. For example, we can use multiple tunnels. While a port bottleneck remains on port 22, which is the standard SSH port, there will be multiple SSH server threads running behind it for each tunnel stream. Such measures can offer foreseeable performance improvements up

to a certain extent, but will typically continue being slower compared to direct connections. On the other hand, when secure networks are involved, direct connections may simply be impossible, as we encountered in our case studies. That REMUS would perform as well as direct connections was never our claim. Throughput degradation is expected, but unobtrusive cross-firewall communication between Grid applications becomes possible. The experience also taught us the following:

- REMUS connections can be optimized using a combination of compression and cipher configuration choices. The implementation

Table 1 Throughput figures for GridFTP tests using the SOCKS implementation of REMUS show rerouting degradation can be as low as 25% of Direct transfers

Transport	Transfer rate
Direct GridFTP	Up to 4.97 Mbps
REMUS rerouted and tunneled (local SOCKS proxy)	Up to 3.91 Mbps
REMUS rerouted only (remote SOCKS proxy)	2.8 Mbps
REMUS rerouted only (remote SOCKS proxy, alternative implementation)	1.8 Mbps

also has an effect. We use Dante to “socksify” connections in both modes 2 and 3 as described above, but using a remote Dante SOCKS proxy brought performance down to less than 1 Mbps. The Mode 3 figures in the previous table come from the use of OpenSSH as the remote SOCKS proxy, with slightly better performance at about 1.8 Mbps (last entry in Table 1). Other implementations of SOCKS may provide even better results, but finding or building the best, with the most optimized configuration, will require some experimentation.

- It is best to situate the SOCKS proxy at the host from which the traffic originates, rather than at the destination host. As described earlier, we tried using a SOCKS proxy at the remote host in Mode 3, connected to directly (port 1080). We also tried a Dante SOCKS proxy accessible via a straight port-forwarding tunnel from the source host, availing therefore of the compression in the tunnel. Either way, there was throughput degradation, as the bottleneck appears to be the SOCKS dynamics and encapsulation between the SOCKS client and the SOCKS proxy. We yielded better results in Mode 2, where the SOCKS proxy was in the same host as the component sending data, and the encapsulation overhead can be excluded from the actual transport between networks.

9 Related Work

Many have studied the problem caused by firewalls that hinder successful Grid operations [4–6, 12, 13, 19, 21, 22, 26], particularly for Globus-based Grids. The Globus toolkit firewall requirements document [28] defines the access settings necessary to facilitate successful Globus transactions. A large range of ports must be accessible in both directions, whereas security policies tend to generally restrict the number of open ports. In many cases, it is simpler to use tunnels through those few open ports, if possible, rather than to open several ports. An early and common example of using tunneling is through SSH port forwarding [12]. SSH is commonly accepted

through firewalls to some hosts in the secure network because SSH is itself hardened against unauthorized access. Luckily, SSH provides port forwarding as a means to route several channels of traffic through a single connection. However, tunneling Globus transactions this way is somewhat unwieldy. It is necessary, for example, to explicitly include port numbers in Globus toolkit command line arguments. It is also necessary to sacrifice callbacks from Globus components, which won’t work if direct connections are blocked by firewalls protecting those remote sites. This can be mitigated to a certain degree, e.g., using batch submissions, but giving up callback functionalities should not be necessary, as we illustrate with REMUS.

Other customizations of Globus, such as the *Nexus Proxy* that can sit outside the firewall [26], are undesirable because of those customizations. Likewise, users may also resort to reprogramming Globus through hard-coded settings, such as one case involving GridFTP running on a private network that was subject to network address translation (NAT) to remote hosts on the Internet [27]. One Grid programming environment we surveyed, *NetIbis* [4], uses a specialized network infrastructure underneath that uses *simultaneous initiation* or *TCP splicing*, a feature included in the TCP standard in which both parties establish connections with each other. Such reprogramming and customizations should not be necessary. Legacy or pre-existing applications and Grid implementations need not be reprogrammed. *Generic connection brokering* (GCB) uses a broker, accessible to both parties, which will make the decision as to the direction for the connection between parties [23]. This allows a connection to proceed between two components in either direction, even if firewall restrictions would normally block connections in one direction or the other. The use of the GCB that is accessible to outgoing connections from connecting hosts is similar to how REMUS makes use of rerouters in proxy nodes. However, compared to the SOCKS wrapper implementation of REMUS, GCB is not as unobtrusive, as it does not use the notion of wrappers to completely hide the presence of a third party relay. *XRAY* [22] is another facility that uses the *GCB broker* in order to facilitate communications between client and server applications.

XRAY uses agents on behalf of clients and servers to minimize customizations to applications and middleboxes. *Owner tests* determine benign and legitimate connections while also using the defensive features of available middleboxes, e.g., firewalls. XRAY manages rules to allow or deny connections to or from sockets of client and server applications after successful authentication and authorization tests. To facilitate connections to servers, the agent for a successfully registered server application obtains an official IP address through a GCB *inagent* which leases the required sockets on the GCB broker. Client applications are informed of the officially bound addresses so that connections are made to the GCB broker instead. Client and server certificates are also required for successful connections between client and server. REMUS is architecturally compatible with XRAY, with similar recourse to agents and relaying via mutually accessible proxies. On the other hand, REMUS ignores middleboxes and our network model assumes that middleboxes are relevant only at the fabric level.

There is a particular class of solutions to the firewall traversal problem which views the firewall and other middleboxes as a controllable resource, sometimes referred to as a *dynamic firewall* [6]. Intermediary devices that sit between communicating components in a network are called middleboxes, and the notion of *middle-box communications* [19] standardizes them as controllable resources. There is on-going effort pushing for an appropriate IETF standard for “midcom”. However, there are also non-standard efforts along the lines of controllable firewalls. Cooperative on-demand opening, developed by the people behind the XRAY project, controls firewalls directly by altering the in-kernel firewall configuration in order to admit connections if verified to be legitimate [21]. Tests are conducted to authenticate and authorize applications before lifting the restrictions as required. Owner tests and application registration is essential given that the firewall itself is being controlled through agents. The dynamic firewall (*Dyna-Fire*) project within the Advanced Computational Data Center Grid (ACDC-Grid) also makes the firewall reconfigurable at [13]. Connections are only established after a successful “knocking” phase when clients

authenticate themselves through an authorized secret “knock”. The philosophy is good but is not without disadvantages. A number of ports on the firewall must be reserved for port knocking communication, and cost of monitoring knocks is non-trivial. There is also concern about the port knocking implementation and certain security exploits, i.e., message replay and man-in-the-middle attacks [6]. However, Dyna-Fire will only trust a knock from a trusted IP address, but this makes client authentication dependent on IP addressing, rather than encrypted and/or signed credentials.

Other forms of tunneling mechanisms have been built but for other purposes, such as glogin [20], which erects a tunnel through a job submitted as a normal Globus job. This is intended to provide interactivity to users, giving them remote login access to Grid nodes. It will not work if the firewall blocks connections between the user hosts and the glogin job on the compute node, or if jobs are constrained by maximum runtimes.

An impressive connectivity infrastructure that we may compare against is WOW [10]. Real world hosts execute virtual machines which behave as virtual Grid workstations. WOW provides the virtual addressing and location facilities in order to form a virtual network of Grid resources. These are able to interconnect across firewalls through a peer-to-peer (P2P) overlay network sitting outside secure domains. However, as with any solution, the firewalls must allow at least outgoing connectivity to the P2P network, which facilitates tunnels in order to provide Grid networking across secure domains of different institutions. This would not have worked with the Royal Melbourne Institute of Technology (RMIT) student lab scenario, which we briefly describe below in the conclusion section. The P2P overlay network therefore performs the functions of a proxy facility. WOW provides several important facilities, including a robustness that allows virtual workstations to migrate from one real world network to other real world networks. WOW is also mostly unobtrusive, since virtual workstations exist in a totally sandboxed guest environment on the host that executes it. However, these features come with a price. The virtual machine approach is costly, since virtual workstations are executing as a complete guest operating system. WOW

virtual workstations are complete Linux systems. They only require a minimal Linux installation, which excludes desktop utilities but includes a full kernel, a basic set of device drivers, and Grid middleware. Disk storage requirements are small, but memory requirements are as would be required of a Grid resource. More importantly, each virtual workstation executes as many heavyweight and lightweight processes as one would expect in a standalone host running Linux. This has considerable CPU cycle implications for the machine hosting the virtual workstation. In contrast, REMUS uses the middleware approach. On a given Grid resource, REMUS introduces a middleware tier underneath Grid middleware such as REMUS. Both memory and processor impact are minimal. As far as the viability of firewall traversal is concerned, REMUS and WOW are comparable. They both require a reasonable firewall situation, in which connections are possible with proxies (in the case of REMUS) or P2P overlay networks (in the case of WOW). In the data transfer measurements of WOW, throughput was between 1.36 and 1.83 Mbps, using SSH (secure) copy. REMUS throughput, via GridFTP transfers using SSH tunnels, reached more than 3.75 Mbps. However, we cannot accurately compare throughput since data concerning network distances between hosts and background (unrelated) loads across networks between hosts are not available. While we can measure the degradation of throughput between direct and REMUS-facilitated data transfers, no such measurement was given for the hosts involved in the WOW data transfer tests.

REMUS prefers to leave firewalls alone. Middlebox devices are viewed as fabric resources which need not be visible at the Grid connectivity level. Implementing REMUS rerouting via SOCKS and SSH provides an adequate degree of security since SSH connections require legitimate user accounts that are authenticated with asymmetric key cryptography. REMUS SOCKS proxies run at the user level and are configured individually for each user. In essence, connections are legitimized by the fact that the users establishing those connections are legitimate users with accounts and credentials authenticated at the system level, i.e., via SSH logon. Note that it is also possible to use X.509 certificates for SSH

logons using implementations of SSH that support GSSAPI [14]. We must also note that controllable middleboxes within a network are not yet typical, apart from direct super-user commands to control the firewall or router. Middlebox recognition of Grid-based credentials is largely unsupported at this time, except in cases where firewall and routing mechanisms are incorporated into firewall traversal systems, e.g., [13, 21, 22]. Firewall reprogramming at runtime must be approached carefully, given the potential risks involved, even for short-duration tampering. It also presents a blurring of fabric and connectivity components, where middleboxes like firewalls and NAT gateways become visible at the Grid level, i.e., they become Grid resources. Network policies will require substantial rethinking in order to make such blurring satisfactory across the entire organization. There are also disadvantages when relying on proxy X.509 certificates for a temporary lifting of firewall restrictions. Some proxy certificates may be valid for weeks, such as those used in long-running scientific computations. While REMUS does not disagree with the notion of firewall controls at the Grid-level, we are working with what are currently typical situations.

10 Conclusion

Figure 9 illustrates what we have achieved with REMUS. As a “lower middleware”, REMUS implements a virtual connectivity layer that enables Grid communications on top of the Fabric Layer’s TCP/IP stack, without breaking the restrictions imposed by firewalls. We implemented a rerouting architecture that uses wrappers to facilitate connections across firewall restrictions, and tested it across a VO setting with a firewall between two participating institutions’ networks. One institution hosts Nimrod/G and accepts experiments, while the other institution hosts Globus computational resources. Our latest implementation of REMUS provided basic connectivity using SSH to tunnel connections through, with SSH being the open channel authorized by both institutions. In order to handle multi-port communications, we used SOCKS proxy connections tunneled through the SSH session. To provide components with

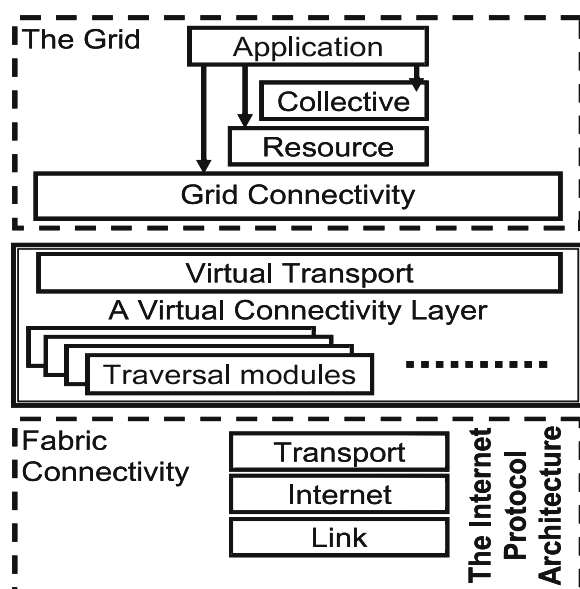


Fig. 9 REMUS is an implementation of a virtual connectivity layer that supports general-purpose Grid communications across firewalls

SOCKS capabilities quickly, we used SOCKS wrappers that automatically reroute connections via SOCKS across the SSH tunnel. Importantly, *there was no need to modify any piece of code for both Globus and Nimrod/G*, nor was there a need to modify the firewall settings. The libraries used for SOCKS communication were dynamically linked and simply preloaded using environment variable settings. This version of REMUS also demonstrates how GSI authentication succeeds despite the actual port dynamics involved in SOCKS-wrapping and tunneling.

To date, we have had two Nimrod/G and Globus installations outside of Monash University that were assisted by REMUS. A normal student lab at RMIT University in Melbourne, Australia, was configured as a Globus resource running Condor for job management. Being inaccessible from outside, we created two SSH + SSH tunnels between Monash and RMIT using a DMZ host as a proxy node, as shown in Fig. 7. We used library preloading to place all Nimrod and Globus processes within a SOCKS-enabled environment. The second case involved the South Australian Partnership for Advanced Computing,

which also ran a cluster within a protected network. Globus connections were accepted through a gateway node that was outside of these restrictions. However, Nimrod agents executing on the cluster nodes were unable to connect back to the Nimrod servers outside. They implemented REMUS with pure SOCKS (no SSH) so that the Nimrod agents are rerouted to access Nimrod servers outside.

In the near future, we will further investigate the performance implications of REMUS. We do not have access to performance data from the two installations just mentioned. They have discontinued use of these facilities for reasons unrelated to Nimrod/G, Globus and REMUS. At present we only have access to REMUS installed across two campuses of Monash University. We are looking to install REMUS on other live sites, where we can test the efficacy of parallelizing connections through multiple proxies and multiple tunnels. We will also explore and implement other means of traversing firewalls, since different networks configure different sets of open ports. REMUS is envisioned to provide a multi-pronged approach, with multiple rerouting capabilities to suit different scenarios. We must also investigate the applicability of REMUS in facilitating UDP connections, something which is likely, using SOCKS v5, but has yet to be attempted in REMUS. Finally, we will improve the internal architecture of REMUS rerouters to make it easier for them to adapt to whatever deployment scenario they encounter, through semi-automation features, policy translation [18], or through easy configuration directives.

Acknowledgements The Australian Partnership for Advanced Computing supported this work. Prototype experiments with REMUS were made possible with the kind assistance of the South Australian Partnership for Advanced Computing and Dr. Savitri Bevinakoppa of Royal Melbourne Institute of Technology University, Melbourne.

References

1. Abramson, D., Sasic, R., Giddy, J., Hall, B.: Nimrod: a tool for performing parametrised simulations using distributed workstations. In: Proceedings of the 4th

- IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995
2. Buyya, R., Abramson, D., Giddy, J.: Nimrod/G: an architecture of a resource management and scheduling system in a global computational Grid. In: Proceedings of HPC Asia 2000, Beijing, China, 14–17 May 2000
3. Comer, D., Stevens, D.: Internetworking with TCP/IP volume 3: client-server programming and applications. Prentice-Hall, Upper Saddle River (2001)
4. Inferno Nettverk A/S: Dante—a free SOCKS implementation (Online). <http://www.inet.no/dante/>. Accessed 2005
5. Denis, A., Aumage, O., Hofman, R., Verstoep, K., Kielmann, T., Bal, H.E.: Wide-area communication for grids: an integrated solution to connectivity, performance and security problems. In: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13 '04), pp. 97–106 (2004)
6. Dynamic Firewalls Configuration, Ver. 1.0, D-Grid Integrationsprojekt (DGI), Technical Report, (Online). https://www.d-grid.de/fileadmin/user_upload/documents/DGI-FG3-5/FG3-5_Dynamic_Firewalls.pdf. Accessed Feb. 2006
7. Foster, I., Kesselman, C., Tsudik, G., Tuecke, S.: A security architecture for computational Grids. In: Proceedings of the 5th ACM Conference on Computer and Communications Security Conference, pp. 83–92 (1998)
8. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the Grid: enabling scalable virtual organizations. *Int. J. Supercomput. Appl.* **15**(3), 200–222 (2001)
9. Foster, I., Kesselman, C.: Globus: a metacomputing infrastructure toolkit. *Int. J. Supercomput. Appl. High Perform. Comput.* **11**(2), 115–128 (1997), Summer
10. Ganguly, A., Agrawal, A., Boykin, P.O., Figueiredo, R.J.: WOW: self-organizing wide area overlay networks of virtual workstations. *J. Grid Comput.* **5**(2), 151–172 (2007)
11. The Globus Alliance: Globus Toolkit. <http://www.globus.org>. Accessed 2008
12. Graupner, S., Reimann, C.: Globus grid and firewalls: issues and solutions in a utility data center environment, Technical report HPL-2002-278. Hewlett-Packard, Palo Alto, 2 October 2002
13. Green, M.L., Gallo, S.M., Miller, R.: Grid-enabled virtual organization based dynamic firewall. In: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID'04), pp. 208–216 (2004)
14. Hutzelman, J., Salowey, J., Galbraith, J., Welch, V.: Generic Security Service Application Program Interface (GSS-API) Authentication and Key Exchange for the Secure Shell (SSH) Protocol, RFC document, IETF (Internet Engineering Task Force). <http://www.ietf.org/rfc/rfc4462.txt>. Accessed May 2006
15. Lee, Y.: SOCKS: a protocol for TCP proxy across firewalls, NEC Systems Laboratory (Online). <http://archive.socks.permeo.com/protocol/socks4.protocol>. Accessed 2005
16. Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., Jones, L.: SOCKS protocol version 5. IETF RFC 1928. <http://tools.ietf.org/html/rfc1928>. Accessed March 1996
17. Mills, W.N. III, Krueger, L., Chiu, W., Halim, N., Hellerstein, J.L., Squillante, M.S.: Metrics for performance tuning of web-based applications. Computer Measurement Group, Turnersville (2000)
18. Neisse, R., Bosquiroli Almeida, M.J., Granville, L.Z., Rockenbach Tarouco, L.M.: Policies translation for integrated management of grids and networks. In: Liebrock, L.M. (ed.) Proceedings of the 2005 ACM Symposium on Applied Computing, pp. 724–729, March 13–17, Santa Fe, New Mexico. SAC '05. ACM, New York (2005)
19. Quittek, J., Stiernerling, M., Srisuresh, P.: Definitions of Managed Objects for Middlebox Communication, Internet-Draft, IETF (Internet Engineering Task Force), October 2006
20. Rosmanith, H., Kranzlmüller, D.: Glogin - a multifunctional, interactive tunnel into the grid. In: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID'04), pp. 266–272 (2004)
21. Son, S., Allcock, B., Livny, M.: CODO: firewall traversal by cooperative on-demand opening. In: Proceedings of the 14th IEEE Symposium on High Performance Distributed Computing, July 2005
22. Son, S., Farrellee, M., Livny, M.: a generic proxy mechanism for secure middlebox traversal. In: Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2005). Boston, Massachusetts, USA, 27–30 September 2005
23. Son, S., Livny, M.: Recovering internet symmetry in distributed computing. In: Proc. 3rd IEEE/ACM International Symposium Cluster Computing and the Grid (CCGrid 2003), pp. 542–549, 12–15 May 2003
24. Sudholt, W., Baldrige, K., Abramson, D., Enticott, C., Garic, S.: Application of Grid computing to parameter sweeps and optimizations in molecular modelling. In: Zomaya, A. (ed.) Parallel Computing for Bioinformatics, pp. 27–35. Wiley, Hoboken (2006). Also appeared in The International Conference on Computational Sciences (ICCS04), Krakow Poland, 6–9 June 2004
25. Tan, J., Abramson, D., Enticott, C.: Bridging organizational network boundaries on the Grid. In: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid 2005). Seattle, Washington, 13–14 November 2005
26. Tanaka, Y., Hirano, M., Nakada, H., Sekiguchi, S.: Performance evaluation of a firewall-compliant Globus-based wide-area cluster system. In: Proceedings of the IEEE International Symposium on High-Performance Distributed Computing 2000 (HPDC 2000), pp. 121–128, 1–4 August 2000
27. Thorpe, S.: Online, an email sent to the globus-discuss mailing list about running a GridFTP server behind a NAT firewall. http://www-unix.globus.org/mail_archive/discuss/2005/01/msg00216.html. Accessed January 2005

28. Welch, V.: Globus Toolkit Firewall Requirements, version 9. The Globus Alliance, (Online). <http://www.globus.org/toolkit/security/firewalls/Globus-Firewall-Requirements-9.pdf>. Accessed 31 October 2006
29. Welch, V., Foster, I., Kesselman, C., Mulmo, O., Pearlman, L., Tuecke, S., Gawor, J., Meder, S., Siebenlist, F.: X.509 proxy certificates for dynamic delegation. In: Proceedings of the 3rd Annual PKI R&D Workshop (2004)
30. Ylonen, T.: SSH—Secure login connections over the internet. In: Proc. 6th USENIX Security Symposium, San Jose, CA, pp. 37–42, July 1996