SmartSockets: Solving the Connectivity Problems in Grid Computing

Jason Maassen and Henri E. Bal
Dept. of Computer Science, Vrije Universiteit
Amsterdam, The Netherlands
jason@cs.vu.nl, bal@cs.vu.nl

ABSTRACT

Tightly coupled parallel applications are increasingly run in Grid environments. Unfortunately, on many Grid sites the ability of machines to create or accept network connections is severely limited by firewalls, network address translation (NAT) or non-routed networks. Multi homing further complicates connection setup and machine identification. Although ad-hoc solutions exist for some of these problems, it is usually up to the application's user to discover the cause of the connectivity problems and find a solution. In this paper we describe *SmartSockets*,[1] a communication library that lifts this burden by automatically discovering the connectivity problems and solving them with as little support from the user as possible.

Categories and Subject Descriptors: C.2.4 [Distributed Systems]: Distributed applications

General Terms: Algorithms, Design, Reliability

Keywords: Connectivity Problems, Grids, Networking, Parallel Applications

1. INTRODUCTION

Parallel applications are increasingly run in Grid environments. Unfortunately, on many Grid sites the ability of machines to create or accept network connections is severely limited by network address translation (NAT) [14, 26] or firewalls [15]. There are even sites that completely disallow any direct communication between the compute nodes and the rest of the world (e.g., the French Grid5000 system [3]). In addition, multi homing (machines with multiple network addresses) can further complicate connection setup.

For parallel applications that require direct communication between their components, these limitations have hampered the transition from traditional multi processor or cluster systems to Grids. When a combination of Grid sites is used, serious connectivity problems are often encountered.

[1]SmartSockets is part of the Ibis project, and can be found at http://www.cs.vu.nl/ibis

HPDC'07, June 25–29, 2007, Monterey, California, USA.

Unfortunately, it is often up to the user to discover the cause of these connectivity problems, a non-trivial task at best. Once the problems are identified, it may be possible to circumvent some of them by using ad-hoc solutions, such as opening a port range in a firewall, explicitly specifying which address to use on a multi-homed machine, or using SSH tunneling. Many problems, however, can only be solved by adapting the application or the communication library it uses. To make matters worse, as soon as the set of Grid systems being used changes, a large part of this process needs to be repeated. As a result, running a parallel application on multiple Grid sites can be a strenuous task [34].

In this paper we will describe a solution to this problem: the SmartSockets communication library. The primary focus of SmartSockets is on ease of use. It automatically discovers a wide range of connectivity problems and attempts to solve them with little or no support from the user. SmartSockets combines many known solutions, such as port forwarding, TCP splicing and SSH tunneling, and introduces several new ones that resolve problems with multi homing and machine identification. In 30 connection setup experiments, using 6 different sites worldwide, SmartSockets was always able to establish a connection, while conventional sockets only worked in 6 experiments. Using heuristics and caching, SmartSockets is able to significantly improve the connection setup performance.

SmartSockets offers a single integrated solution that hides the complexity of connection setup in Grids behind a simple interface that closely resembles sockets. We will show that it is relatively straightforward to port an existing application to SmartSockets, provided that certain programming guidelines are followed.

SmartSockets is not specifically intended for use in parallel applications or Grids. It can also be applied to other distributed applications, such as visualization, cooperative environments, or even consumer applications such as instant messaging, file sharing, or online gaming. However, many of these applications only require a very limited degree of connectivity. Often, clients simply connect to a server in a well-known location, making it relatively easy to apply an ad-hoc solution when a connectivity problem occurs.

Parallel applications, however, can be much more challenging. They often require a large number of connections between the participating machines, and each machine must be capable of both initiating outgoing and accepting incoming connections. Running such applications in a Grid environment with limited connectivity is difficult. Therefore, this paper will focus on this domain.

In Section 2 we describe the connectivity related problems encountered while running applications on multiple Grid sites. Section 3 describes how these problems are solved in SmartSockets and briefly looks at the programming interface. Section 4 evaluates the performance of SmartSockets, Section 5 describes related work, and Section 6 concludes.

2. CONNECTIVITY PROBLEMS

In this section we will give a description of the network related problems that can occur when running a single parallel or distributed application on multiple Grid sites.

2.1 Firewalls

As described in [15], "A firewall is an agent which screens network traffic in some way, blocking traffic it believes to be inappropriate, dangerous, or both.". Many sites use firewalls to protect their network from unauthorized access. Firewalls usually allow outbound connections, but block incoming connections, often with the exception of a few well-known ports (e.g., port 22 for SSH).

It is obvious that this connectivity restriction can cause severe problems when running a parallel application on multiple sites. When only a single participating site uses firewall, the connectivity problems can sometimes be solved by ensuring that the connections setups are 'in the right direction', i.e., that all required connections between open and firewalled machines are initiated at the firewalled site. This solution may require changes to the applications or communication libraries, however. Also, if both sites use a firewall, this approach can no longer be used. In this case, a firewall will always be encountered regardless of the connection setup direction.

One way to solve the problems is to request an *open port range* in the firewall. Connectivity can then be restored by adapting the application to only use ports in this range. Besides requiring reconfiguration of the firewall, open ports are also seen as a threat to site security.

When both machines are behind a firewall it may still be possible to establish a direct connection using a mechanism called TCP splicing [6, 10, 13, 20]. Simply put, this mechanism works by simultaneously performing a connection setup from both sides. Since this approach requires explicit cooperation between the machines, some alternative communication channel must be available.

2.2 Network Address Translation

As described in [21], "Network Address Translation is a method by which IP addresses are mapped from one address realm to another, providing transparent routing to end hosts.". NAT was introduced in [12] as a temporary solution to the problem of IPv4 address depletion. Although the intended solution for this problem, IPv6, has been available for some time, NAT is still widely used today.

Many different flavors of NAT exist, but the *Network Address Port Translation* is most frequently used [21, 29]. This type of NAT allows outbound connections from sites using private addresses, but does not allow incoming connections. Both the IP address (and related fields) and the transport identifier (e.g., TCP and UDP port numbers) of packets are translated, thereby preventing port number collisions when a set of hosts share a single external address.

As mentioned above, NAT only allows outbound network connections. Incoming connections are rejected, since the connection request does not contain enough information to find the destination machine (i.e., only the external IP address is provided, but that may be shared by many machines). This restriction leads to connectivity problems that are very similar to those caused by firewalls. Therefore, the solution described in Section 2.1 (connecting 'in the right direction') also applies to a NAT setup, and fails in a similar way when multiple NAT sites try to interconnect.

Although the TCP splicing mechanism can also be used to connect two NAT sites, a more complex algorithm is required to compensate for the port translation performed by NAT [6, 20].

Some NAT implementations have support for *port forwarding*, where all incoming connections on a certain port can be automatically forwarded to a certain host inside the NAT site. Using mechanisms such as UPnP [5], DPF [28], or MIDCOM [30], applications can contact the NAT implementation and change the port forwarding rules on demand. Port forwarding lifts many of the restrictions on incoming connections. Unfortunately, UPnP is mostly found in consumer devices, MIDCOM is still under development, and DPF only supports NAT (and firewall) implementations based on NetFilter [1]. As a result, these mechanisms are not (yet) generally usable in Grid applications. Currently, SmartSockets only supports UPnP.

In addition to causing connection setup problems, NAT also complicates machine identification. Machines in a NAT site generally use IP addresses in the private range [26]. These addresses are only usable within a local network and are not globally unique. Unfortunately, parallel applications often use a machine's IP address to create a unique identifier for that machine. When multiple NAT sites participate in a single parallel run, however, this approach can not be used, since the machine addresses are no longer guaranteed to be unique.

2.3 Non-routed networks

On some sites no direct communication between the compute nodes and the outside world is possible due to a strict separation between the internal and external networks. No routing is performed between the two. Only a front-end machine is accessible, and the connectivity of this machine may be limited by a firewall or NAT. Two of the sites used in Section 4 use such a setup.

It is clear that this is a major limitation when the site is used in a parallel application. The only possibility for the compute nodes to communicate with other sites is to use the front-end machine as a bridge to the outside world, using, for example, an SSH tunnel or a SOCKS [24] proxy. These are non-trivial to set up, however.

2.4 Multi Homing

When multi-homed machines (i.e., machines with multiple network addresses) participate in a parallel application, another interesting problem occurs. When creating a connection to such a machine, a choice must be made on which of the possible target addresses to use. The outcome of this choice may depend on the location of the machine that initiates the connection.

For example, the front-end machine of a site has two addresses, a public one, reachable over the internet, and a private one used to communicate with the site's compute nodes. As a result, a different address must be used to reach this

machine depending on whether the connection originates inside or outside of the site.

In [34] we called this the *Reverse Routing Problem*. Normally, when a multi-homed machine is trying to connect to a single IP address, a routing table on the machine decides which network is used for the outgoing connection. In the example described above the reverse problem is encountered. Instead of having to decide how to 'exit' a multi-homed machine, we must decide on how to 'enter' it. This problem is non-trivial, since the source machine generally does not have enough information available to select the correct target address. As a result, several connection attempts to different addresses of the target may be necessary before a connection can be established. In Section 3.2 we will describe heuristics that can be used to speed up this process.

Multi homing can have a major effect on the implementation of parallel programming libraries. The example above shows that it is not sufficient to use a single address to represent a multi-homed machine. Instead, all addresses must be made available to the other participants of the parallel application. In addition, some of the addresses may be in a private range and refer to a different machine when used in a different site. Therefore, it is also essential to check if a connection was established to the correct machine.

3. SMARTSOCKETS

In this section we will give an overview of the design, implementation and programming interface of the SmartSockets library, and describe how it solves the problems described in the previous section.

3.1 Overview

Currently, SmartSockets offers four different connection setup mechanisms, *Direct, Reverse, Splicing, and Routed*. They will be described in more detail below. Table 1 shows an overview of how these mechanisms solve the connectivity problems described in Section 2. As the table shows, each problem is solved by at least one mechanism.

Table 1: Overview of connectivity problems and their solutions.

	Connection Setup Mechanism			
Problems	Direct	Reverse	Splicing	Routed
Identification	X			
Multi Homing	X			
Single FW/NAT	(X)	X	X	X
Dual FW/NAT	(X)		X	X
No Routing				X

The machine identification and multi-homing problems are solved by the direct connection setup. As will be explained below, this approach also has limited firewall traversal capabilities (using SSH tunneling), so in certain situations it may succeed in establishing a connection in a single or even a dual firewall setting. In the table these entries are shown between brackets.

A reverse connection setup is only capable of creating a connection when a single firewall or NAT limits the connectivity. Splicing is capable of handling both single and dual firewall/NAT configurations. However, this approach is significantly more complex than a reverse connection setup (especially with dual NAT) and may not always succeed. Therefore, reverse connection setup is preferred for single firewall/NAT configurations.

A routed connection setup can be used in any situation where the connectivity is limited. Unlike the previous two approaches it does not result in a direct connection. Instead all network traffic is routed via external processes called hubs (explained in Section 3.3), which may degrade both latency and throughput of the connection. Therefore, the previous mechanisms are preferred. When connecting to or from a machine on a non-routed network, however, a routed connection is the only choice.

The SmartSockets implementation is divided into two layers, a low-level *Direct Connection Layer*, responsible for all actions that can be initiated on a single machine, and a high-level *Virtual Connection Layer* that uses *side-channel* communication to implement actions that require cooperation of multiple machines. The direct connection layer is implemented using the standard socket library. The virtual connection layer is implemented using the direct connection layer. Both layers will be explained in more detail below. Currently, SmartSockets is implemented using Java [2].

3.2 Direct Connection Layer

The direct connection layer implements all actions that do not require explicit cooperation between machines, such as determining the local addresses or creating a direct connection. It also supports a limited form of SSH tunneling.

3.2.1 Machine Identification

During initialization, the direct connection layer starts by scanning all available network interfaces to determine which IP addresses are available to the machine. It then generates a unique *machine identifier* that contains these addresses, and that can be used to contact the machine.

This identifier will automatically be unique if it contains at least one public address. If all addresses are private, however, additional work must be done. A machine that only has private addresses is either in a NAT site or uses a non-routed network. In the first case, a unique identifier can still be generated for the machine by acquiring the external address of the NAT. Provided that this address is public, the combination of external and machine addresses should also be unique, since other machines in the same NAT site should have a different set of private addresses, and all other NAT sites should have a different external address.

The SmartSockets library will use UPnP to discover the external address of the NAT site. If this discovery fails, or if the returned address is not public, a *Universally Unique Identifier* (UUID) [23], will be generated and included in the machine identifier, thereby making it unique.

3.2.2 Connection Setup

Once initialized, the direct connection layer can be used to set up connections to other machines. The identifier of the target machine may contain multiple network addresses, some of which may not be reachable from the current location. The private addresses in the identifier may even refer to a completely different machine, so it is important that the identity of the machine is checked during connection setup. As a result, several connection attempts may be necessary before the correct connection can be established.

When multiple target addresses are available, a choice must be made in which order the connection attempts will be performed. Although simply using the addresses in an arbitrary order should always result in a connection (pro-

vided that a direct connection is possible), this may not be the most efficient approach. Many Grid sites offer high-performance networks such as Myrinet [7] or Infiniband [4] in addition to a regular Ethernet network. Using such a network for inter-site communication may significantly improve the application's performance. In general, these fast networks are not routed and use addresses in the private range, while the regular Ethernet networks (often) use public addresses. Therefore, by sorting the target addresses and trying all private ones first, the fast local networks will automatically be selected in sites with such a setup.

The drawback of this approach is that the private addresses of a target will always be tried first, even if the connection originates on a different site. This may cause a significant overhead. Therefore, SmartSockets uses a heuristic that sorts the target addresses in relation to the addresses that are available locally. For example, if only a public address is available on the local machine, it is unlikely that it will be able to create a direct connection to a private address of a target. As a result, the connection order *public before private* is used. This order is also used if both machines have public and private addresses, but the private addresses refer to a different network (e.g., *10.0.0.10* vs. *192.168.1.20*). The order *private before public* is only used if both machines have private addresses in the same range. Section 4 will illustrate the performance benefits of this heuristic.

Unfortunately, it is impossible to make a distinction between addresses of the same class. For example, if a target has multiple private addresses, we can not automatically determine which address is best. Therefore, if a certain network is preferred, the user must specify this explicitly. Without this explicit configuration, SmartSockets will still create a direct connection (if possible), and the parallel application will run, but its performance may be suboptimal.

When a connection has been established, an identity check is performed to ensure that the correct machine has been reached. This would be a simple comparison if the complete identifier of the target is available, but unfortunately this is not always the case. User provided addresses are often used to bootstrap a parallel application. These addresses are often limited to a single hostname or IP address, which may only be part of the addresses available to the target machine. Therefore, the identity check used by SmartSockets also allows the use of *partial identifiers*.

Whenever a connection is created, the target machine provides its complete identity to the machine initiating the connection. This machine then checks if both the public and private addresses in the partial identity are a subset of the ones in the complete identity. If so, the partial identity is accepted as a subset of the complete identity, and the connection is established. Note that although the connection is created to a machine that matches the address specified by the user, it is not necessarily the correct machine from the viewpoint of the parallel application. Unfortunately, in such cases it is up to the user to provide an address that contains enough information to reach the correct machine.

3.2.3 Open Port Ranges and Port Forwarding

When a firewall has an *open port range* available, SmartSockets can ensure that all sockets used for incoming connections are bound to a port in this range. There is no way of discovering this range automatically, however, so it must be specified explicitly by the user.

In addition, SmartSockets can use the UPnP protocol to configure a NAT to do *port forwarding*, i.e., automatically forward all incoming connections on a certain external port to a specified internal address. However, as explained before, this protocol is mainly used in consumer devices.

3.2.4 SSH Tunneling

In addition to regular network connections, the direct connection layer also has limited support for SSH tunneling. This feature is useful for connecting to machines behind a firewall that allows SSH connections to pass through. It does, however, require a suitable SSH setup (i.e., public key authentication must be enabled).

Creating an SSH tunnel is similar to a regular connection setup. The target addresses are sorted and tried consecutively. Instead of using the port specified in the connection setup, however, the default SSH port (i.e., 22) is used. When a connection is established and the authentication is successful, the receiving SSH daemon is instructed to forward all traffic to the original destination port on the same machine. If this succeeds, the regular identity check will be performed to ensure that the right machine has been reached.

Although this approach is useful, it can only be used to set up a tunnel to a different process on the target machine. Using this approach to forward traffic to different machines requires extra information. For example, setting up an SSH tunnel to a compute node of a site through the site's frontend, can only be done if it is clear that the frontend must be contacted in order to reach the target machine. Although this approach is used in some projects [8], the necessary information cannot be obtained automatically and must be provided by the user. Therefore, SmartSockets uses a different approach which will be described in detail in Section 3.3.

3.2.5 Limitations

The direct connection layer offers several types of connection setup which have in common that they can be initiated by a single machine. No explicit cooperation between machines is necessary to establish the connection. There are many cases, however, where connectivity is too limited and the direct connection layer cannot be used.

In general, direct connections to sites that use NAT or a firewall are not possible. Although SSH tunneling and open port ranges alleviate the firewall problems, they require a suitable SSH setup or extra information from the user. Port forwarding reduces the problems with NAT, but is rarely supported in Grid systems. Therefore, these features are of limited use. In the next section we will give a detailed description of the virtual connection layer, which solves these problems.

3.3 Virtual Connection Layer

Like the direct connection layer, the virtual connection layer implements several types of connection setup. It offers a simple, socket-like API and has a modular design, making it easy to extend. Besides a *direct* module that uses the direct connection layer described above, it contains several modules that offer more advanced types of connection setup. These modules have in common that they require explicit cooperation (and thus communication) between the source and target machines in order to establish a connection. As a result, side-channel communication is required to implement these modules.

3.3.1 Side-Channel Communication

In SmartSockets, side-channel communication is implemented by creating a network of interconnected processes called *hubs*. These hubs are typically started on the frontend machines of each participating site, so their number is usually small.

When a hub is started, the location of one or more other hubs must be provided. Each hub will attempt to setup a connection to the others using the direct connection layer. Although many of these connections may fail to be established, this is not a problem as long as a spanning tree is created that connects all hubs.

The hubs use a gossiping protocol to exchange information about themselves and the hubs they know, with the hubs that they are connected to. This way information about each hub quickly spreads to all hubs in the network. Whenever a hub receives information about a hub it has not seen before, it will attempt to set up a connection to this hub. This way, new connections will be discovered automatically.

All gossiped information contains a *state number* indicating the state of the originating machine when the information was sent. Since information from a hub may reach another hub through multiple paths, the state number allows the receiver to decide which information is most recent.

By recording the length of the path traversed thus far in the gossiped information, hubs can determine the distance to the sites that they can not reach directly. Whenever a hub receives a piece of information about another hub containing a shorter distance than it has seen so far, it will remember both the distance and the hub from which the information was obtained. This way, we automatically create a *distributed routing table* with the shortest paths between each pair of hubs. This table is later used to forward application information (as will be described below).

When an application is started, the virtual layer on each machine creates a single connection to the hub local to its site. The location of this hub can either be explicitly specified or discovered automatically using UDP multicast.

3.3.2 Virtual Addresses

The connection to the hub can now be used as a side channel to forward requests to otherwise unreachable machines. To ensure that the target machines can be found, *virtual addresses* are used, consisting of the machine identifier (see Section 3.2), a port number, and the identifier of the hub the machine is connected to.

All requests for the target machine can then be sent to the local hub, which forwards it in the direction of the target hub using the information contained in its routing table. The request will continue to be forwarded until the target hub is reached and the request is delivered to the machine.

3.3.3 Modules

The current implementation of SmartSockets contains four different connection modules, one for each of the connection setup mechanisms described in Section 3.1.

Direct.

The direct connection module simply forwards all connection requests to the direct connection layer. It does not make use of side-channel communication and has the features and limitations described in Section 3.2.

Reverse.

A direct connection setup will generally fail if the target is behind a firewall or NAT. However, as explained in Section 2, outgoing connections are usually allowed on such sites. The reverse connection module exploits this property by reversing the direction of the connection setup.

Instead of creating a connection, the reverse connection module creates a new socket locally. It then sends a request to the target machine using the side channel. This request contains the target's address and destination port, and the address of the new socket. When the request is received on the target machine, it will check if the destination port exists. If it does, the target machine attempts to create a direct connection back to the new socket. If successful, this connection is returned as the result of original connection setup call on the source. On the target, the new connection will be queued, awaiting an accept from the application.

Splicing.

The reverse connection module requires the source machine to be publicly accessible. When both machines are behind a firewall or NAT the reverse connection setup will fail. However, it may still be possible to create a connection using TCP splicing [10, 20]

When the machines have public addresses (i.e., they are not behind a NAT), the actions performed by the splicing module are relatively straightforward. First, the source machine sends a request to the target using the side channel. This request contains the target address and destination port, the complete identifier of the source, and a port the source will use to create the outgoing connection. When the request is received on the target machine, it will check if the destination port exists. If it does, a reply is returned and both machines repeatedly attempt to create a direct connection to the other (using only public addresses). Since this mechanism is sensitive to timing and both machines may have multiple public addresses, the number of attempts required may be large.

When one or both machines are behind a NAT (i.e., they only have private addresses), a different approach is used. As explained in Section 2.2, most NAT implementations translate the address and port number of outgoing connections. TCP splicing requires both machines to know each other's exact external address and port number. Although the external address of a NAT site is often constant, the port mapping is hard to determine, since a different port may be used for every connection attempt.

Fortunately, most NAT implementations use a predictable port mapping scheme [19]. Therefore, once a single mapping has been determined, a prediction can be made on the range of port numbers that is likely to be used in the immediate future. By using the external address and port range in the connection setup attempts, TCP splicing can still be used.

To obtain an initial mapping, the assistance of an external machine (outside of the NAT) is required. SmartSockets uses the hubs for this purpose. If the source machine is behind a NAT, it will request a list of available hubs from its local hub, and attempt to find an external hub to which it can connect directly. When such a connection is successful, the external hub echoes the source address and port number to the source machine. If this address is public, it is likely to be the external address of the NAT site, and the address and port are included in the request sent to the target. The

target will use the same approach if it is behind a NAT, and return its external address and port number to the source.

Both machines will now repeatedly perform connection attempts using the external address of the other site and trying all port numbers in the predicted range. Currently, SmartSockets uses a range of *[port...port+5]*.

It is obvious that there are many cases where the splicing module will not be able to set up a connection. For example, a machine may be unable to find its external address, the external address may be wrong (e.g., in case of multiple consecutive NATs), the port range prediction may be wrong, or the connection attempts may not succeed in time. As shown in [19], the maximum success rate is approximately 86%. Fortunately, there is a backup solution, the routed connections module explained below.

Routed.

The last module available is the routed connection module. Provided that there is at least a spanning tree connecting the hubs, this module should always be able to create a connection between two machines, even if these machines are only connected to non-routed networks.

Whenever a connection is created using the routed connection module, the source sends a connection request to its local hub using the side channel, containing the target address and port. The hub will add this *virtual connection* in its administration and forward the request to the next hub using the routing table described in Section 3.3.1. When the request reaches the target machine, it will make sure that the destination port exists and queue the request.

Once the connection is accepted by the application, a reply is sent back via the hubs, and the virtual connection is established. Both machines now return a *virtual socket*, which, instead of sending its data directly to the target machine, forwards all data through a series of hubs.

3.3.4 Module Order and Caching

To create a new connection, each module is tried until a connection is established, or until it is clear that a connection can not be established at all (e.g., because the destination port does not exist on the target machine). By default, the order *Direct, Reverse, Splice, Routed* is used. This order prefers modules that produce a direct connection.

When a connection is established, the time required for subsequent connection setups can be reduced by caching which module was successful. This information is cached based on the hub address of the target, and not its machine address (see Section 3.3.2). Caching in this way allows an entire site to be represented using a single cache entry (since machines in a site typically share the same hub). Not only does this save memory, but it also improves the effectiveness of the cache. After a connection is created to a single machine of a site, all other connection setups to the same site benefit from the cached information. In Section 4 we will show the benefits of this approach.

3.4 Programming Interface

We will now give a short description of the programming interface of the virtual connection layer of SmartSockets, which is currently implemented using Java [2]. Converting an application to SmartSockets is relatively straightforward, provided that the application uses a *Factory Pattern* [16] to create sockets. The *javax.net* package of the Java class libraries contains interfaces for two such factories, one to create *Sockets* (outgoing connections), the other to create *ServerSockets* (incoming connections). Java also offers implementations which create regular or secure sockets (SSL).

SmartSockets extends the Socket and ServerSocket implementations of Java, and offers two factories which adhere to the interfaces described above. This allows SmartSockets to be plugged in to existing applications by simply changing the factory implementations that are used.

Unfortunately, the addressing scheme used in connection setup cannot be replaced this easily. Currently, Java used three forms of addressing. The first is based on an *InetAddress*, which is hard coded to be either an IPv4 or IPv6 address and cannot be extended. When this scheme is used, SmartSockets has no other choice then to attempt a direct connection to the given address. None of the other connection setup schemes can be used due to a lack of information.

The second addressing scheme is more flexible. It is based on a *SocketAddress* interface, which is implemented by a *VirtualSocketAddress* in SmartSockets. Unfortunately, because Java does not not offer a factory to create these SocketAddresses, most applications explicitly use *InetSocketAddresses*, which consists of a InetAddress and a port number. To make full use of SmartSockets, it is necessary to replace these with a VirtualSocketAddress. For this purpose, SmartSockets offers a *SocketAddressFactory*, that can be used to create both VirtualSocketAddress and InetSocketAddress objects. As with the first scheme, SmartSockets is also backward compatible with InetSocketAddress, although this may restrict the connectivity.

The third scheme simply uses a *String* as a machine address. Although this string is originally intended to contain a host name, it can just as easily be used to carry a string representation of a virtual address. Therefore this mechanism can be used by SmartSockets without any code modification. As with the previous schemes, the connectivity may be restricted when the information in the string is limited.

```
class Example {
  SocketFactory createFactory(String type) {
    if (type.equals("plain"))
      return SocketFactory.getDefault();
    if (type.equals("SSL"))
      return SSLSocketFactory.getDefault();
    if (type.equals("SmartSockets"))
      return SmartSocketFactory.getDefault();
    // else print error
  }
  void run(String type, String address) {
    SocketFactory f = createFactory(type);
    SocketAddressFactory a = new SocketAddressFactory();
    Socket s = f.createSocket().
    s.connect(a.createSocketAddress(address));
    // we can now use the socket.
  }
}
```

Figure 1: Example Application

In Figure 1, an example is shown that is can make use of regular sockets, SSL, or SmartSockets. By varying the *type* parameter of *run*, a different socket factory can be selected. Using the *SocketAddresFactory* provided by SmartSockets (explained above), the target machine address can be translated to a SocketAddress in a portable manner, and used for connection setup.

Table 2: The testbed

Machine	Restrictions (frontend)	Restrictions (nodes)
DAS3-V	MH	MH
DAS3-D	MH	MH
Grid5000	MH, FW	NR
Rockstar	MH, FW	MH, FW
Hiroshi	MH, FW, NAT	NR
Desktop	NAT	n/a

MH = multi-homing, FW = firewall, NR = non-routed

4. EVALUATION

In this section we will evaluate the performance of SmartSockets. We use a testbed consisting of 5 different clusters and a desktop machine. The machines have varying connectivity restrictions, shown in Table 2.

The DAS3 system consist of 5 different clusters in the Netherlands. We will use two, *DAS3-V*, the cluster at the Vrije Universiteit Amsterdam, and *DAS3-D*, the cluster located at the Delft University of Technology. The *Grid5000* system consist of several clusters distributed over 9 sites in France. We use the cluster located at the University of Nice-Sophia Antipolis. The *Rockstar* cluster[2] is located at the San Diego Supercomputing Center, University of California, USA, and the *Hiroshi* machine is located at the School of Information Technologies, University of Sydney, Australia. Finally, *Desktop* is a single machine located in Haarlem, The Netherlands. All sites use AMD or Intel processors of 2.0 Ghz or faster.

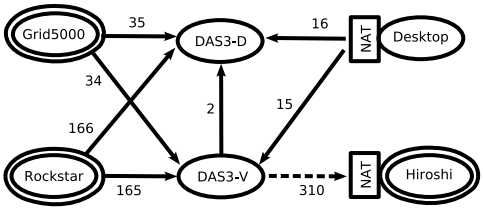

Figure 2: Hub connections on testbed.

Before running the experiments, a hub is started on the frontend machine of each of the clusters and on the desktop machine. Each hub is provided with locations of all other hubs. As explained in Section 3.3.1, each hub attempts to set up a direct connection to all others. The resulting setup is shown in Figure 2. Double circles indicate a site with a firewall. Sites using NAT are explicitly marked. The arrows between the hubs indicate a connection and show the direction in which the connection was established. Each arrow is annotated with the round-trip time between the sites (measured with *ping*). The Hiroshi machine could only be reached using an SSH-tunnel. This tunnel was automatically setup by SmartSockets, but only after a suitable SSH configuration was created on the DAS3-V site.

4.1 Performance

We will start by evaluating the connection setup performance, comparing it to the basic socket performance when possible. The results are shown in Table 3. This table shows the time required to set up a connection between compute nodes of each combination of sites. Connections to and from the desktop machine are also included.

[2]We would like to acknowledge Frank Seinstra for his assistance in running experiments on the Rockstar and Hiroshi machines.

The entries are annotated to indicate which connection style is selected by SmartSockets. This selection is performed automatically and the result is cached. As a result, the correct connection style is immediately selected for all but the first connection.

As the table shows, a conventional connection could only be created in 6 out of the 30 combinations. As expected, SmartSockets selects a direct connection setup in these cases. The time required by SmartSockets to establish a direct connection is roughly twice that of conventional sockets. This is caused by the identity check (see Section 3.2) which requires an extra round trip.

In four cases, SmartSockets selected a reverse connection setup. These correspond to the combinations of machines where the source machine is open, but the target is behind a firewall or NAT. Reverse connection setup adds an additional round trip latency to the time required by the direct connection setup (in the opposite direction). This additional time is needed to forward a reverse connection request message from the source to the target (via the hubs) and to send an accept message (via the new connection) once the application on the target has accepted the incoming connection.

In the other 20 cases, SmartSockets decided to set up a virtual connection by routing all data via the hubs. In these cases the connection setup time is dominated by the round trip time to the machine, as can be seen by comparing the numbers in Table 3 to those in Table 4. Although it is possible to use splicing between the Desktop and Rockstar machines, this method fails occasionally due to the timing sensitivity of this approach. When this occurs, a virtual connection is selected instead. Since SmartSockets uses a cache to remember previous selections, splicing will not be used afterward.

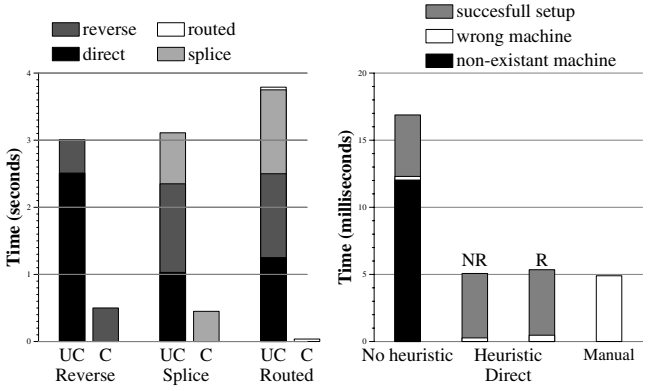

Figure 3: Breakdown of connection setup time.

The first graph of Figure 3 shows the impact of the selected module cache on connection setup time. Only three connection setup mechanism are shown. The connection setup mechanisms are initially tried in the order *Direct, Reverse, Splice, Routed*. Since the *Direct* approach is always tried first, no caching is needed when it is successful. For each mechanism, Figure 3 shows two bars. The first shows the connection setup time when no information is available, the second shows the time when the correct mechanism can be retrieved from the cache. A connection timeout of 5 seconds was specified for both.

The reverse connection setup is performed from DAS3-V to Rockstar. First, about 2.5 seconds is spent in a direct

Table 3: Connection setup time of SmartSockets (time in milliseconds).

	Source					
Target	DAS3-V	DAS3-D	Rockstar	Grid5000	Hiroshi	Desktop
DAS3-V		4.9^{d} (2.4)	332^{d} (166)	68^{v}	595^{v}	33^{d} (17)
DAS3-D	4.9^{d} (2.4)		335^{d} (167)	70^{v}	595^{v}	33^{d} (18)
Rockstar	500^{r}	503^{r}		206^{v}	718^{v}	182^{v}
Grid5000	35^{v}	38^{v}	206^{v}		593^{v}	54^{v}
Hiroshi	630^{v}	603^{v}	750^{v}	670^{v}		640^{v}
Desktop	49^{r}	52^{r}	183^{v}	84^{v}	606^{v}	

Annotations indicate connection style: d for direct, r for reverse, s for splicing, and v for routed. When applicable, the connection setup time of regular sockets is shown between brackets.

connection attempt, which only fails after the timeout has fully expired. This is common behavior when connecting to a machine behind a firewall which blocks the incoming connection but sends no reply. As a result, the source machine has to wait for the timeout. Next, about 0.5 seconds are needed to set up a reverse connection. All subsequent connections are created in 0.5 seconds.

For the spliced connection setup we perform a separate experiment connecting the Desktop machine to the Grid5000 frontend. As Figure 3 shows, the direct and reverse connection setup fail after using most of their 1.25 second time slots. A spliced connection is then created in 0.45 seconds.

The routed connection setup is performed from a DAS3-V to a node of Grid5000. The first three connection setup mechanisms fail, each using 1.25 seconds. A virtual connection is then established between the machines in 37 milliseconds. All subsequent connections are created in 37 milliseconds, reducing the connection time by a factor of 102.

The second graph of Figure 3 shows the impact of the target address heuristic on the direct connection setup time between the DAS3-D and DAS3-V site. DAS3-V has one public and two private addresses, DAS3-D has one of both. The range of one of the private addresses overlaps on both sites. The figure shows four experiments, one with the heuristic turned off, two with the heuristic turned on, and one where the correct address was manually selected.

The first experiment shows that if no heuristic is used, the connection setup requires 17 milliseconds, of which 12 are spent attempting a connection setup to a private address which does not exist on DAS3-D. Next, about 0.2 milliseconds is required to discover that no connection is possible to the second private address (since no one is listening). Finally, the correct connection is set up in 4.5 milliseconds.

The second experiment does use the heuristic. In this experiment there is no process listening on the DAS3-D machine that shares the private address with the target DAS3-V machine. Therefore, the first connection attempt fails immediately. The correct connection is then established in 5 milliseconds. The third experiment uses a similar setup, but now there is a process listening on the DAS3-D machine that shares the private address. As a result, a handshake is required to discover that a connection has been established to the wrong machine. This handshake requires approximately twice the time needed for the failed connection attempt of the previous experiment. In the last experiment the correct address is manually selected. As expected, the connection is set up in 5 milliseconds.

Table 4 shows the round-trip time of the connections. In the six cases where regular sockets can also establish a connection, the round-trip time of SmartSockets and the regular socket connection is the same. For the other cases, the round-trip time is approximately the same as the network round-trip time as measured by *ping*. The only exception is the Hiroshi cluster. The frontend of this machine can only be reached using SSH-tunneling. Unfortunately, the overhead of this approach is high, roughly doubling the required round trip time.

Table 5 shows the achievable throughput. As with latency, the throughput of SmartSockets is similar to that of regular sockets (where applicable). The performance of the Hiroshi cluster is limited both by the distance to the other machines and by the encryption performed by the SSH-tunnel used to reach it. The performance of the Desktop machine is limited by its ADSL connection (approximately 3.5 Mbit/s downstream and 800 KBit/s upstream).

5. RELATED WORK

The system described in [10] can be seen as a predecessor to SmartSockets. It was developed in cooperation with our group. The focus of this work was mainly on using splicing to traverse firewalls. Only a limited form of message routing was available (no further than two hops) and the system did not support multi homing, SSH tunneling, or reverse connection setup. To allow the side-channel communication through firewalls, the system needed *gateway nodes* with access to the networks inside and outside of the firewall (e.g. by using an open port range). Machines could then connect to such a gateway whenever side-channel communication was necessary with a machine behind the firewall. Instead, in SmartSockets the hubs use outgoing connections from behind the firewall. This is easier to set up and it does not require any open ports.

Generic Connection Brokering (GCB) [28] can serve as a replacement for traditional sockets, provided that the application follows certain programming guidelines. When a GCB client creates a socket for listening, this socket is registered at a GCB server. This server must be located in a publicly accessible network. Similar to NAT, the server then creates a socket with a public address that acts as a proxy for the private or firewalled client socket. This public address is then returned to the client to be used as the 'official' address of the client socket. When any non-GCB client tries to connect to this address, it will reach the proxy instead. The server will then forward this incoming connection to the client and relay any subsequent data. Connections created by GCB-aware clients will be forwarded to the server instead of the proxy (by replacing the port number in the client address with a well-known server port). This allows the server to mediate in the connection setup between the two clients and, depending on their connectivity restrictions, instruct them to use a direct connection, reverse the connection order, or use the server itself as a relay.

Table 4: Roundtrip latency of SmartSockets (time in milliseconds).

	Source					
Target	DAS3-V	DAS3-D	Rockstar	Grid5000	Hiroshi	Desktop
DAS3-V		2.3 (2.3)	166 (166)	56	528	14 (14)
DAS3-D	2.3 (2.3)		167 (167)	57	533	15 (15)
Rockstar	166	167		205	590	195
Grid5000	56	57	205		524	50
Hiroshi	528	529	590	522		539
Desktop	14	15	190	43	522	

When applicable, the roundtrip latency of regular sockets is shown between brackets.

Table 5: Throughput of SmartSockets (in Mbit/second).

	Source					
Target	DAS3-V	DAS3-D	Rockstar	Grid5000	Hiroshi	Desktop
DAS3-V		182 (183)	2.6 (2.5)	2.5	0.25	0.65 (0.65)
DAS3-D	185 (186)		2.6 (2.5)	2.6	0.26	0.65 (0.65)
Rockstar	2.8	2.7		6.9	0.23	0.65
Grid5000	7.6	8.2	2.4		0.20	0.65
Hiroshi	0.73	0.73	0.70	0.73		0.61
Desktop	3.3	3.3	2.2	2.2	0.25	

When applicable, the throughput of regular sockets is shown between brackets.

Although GCB is similar to SmartSockets , there are some significant differences. Because GCB clients directly connect to the (remote) GCB server representing the target client, this server must be on a publicly accessible network. Also, outgoing connectivity is required on all client nodes. GCB only supports two hop message routing, and does not have support for multi homing on the client nodes. In SmartSockets the hubs are not required to be publicly accessible, but instead, the hubs must be capable of setting up a spanning tree. It is generally not a problem when a subset of the hubs can only use outgoing connections or can only be reached through SSH tunneling. Outgoing connectivity is not required for the clients, since they can route their connections over the hubs, using multiple hops if necessary. Unlike SmartSockets , GCB does have support for incoming legacy TCP connections.

Many projects attempt to solve the connectivity problems by using peer-to-peer overlay networks. In WOW [18] virtual machines are combined with peer-to-peer techniques to create a *virtual cluster*. By running VMware [31] on all machines a uniform system image can be provided to the applications. All traffic to the (virtual) network device is intercepted and routed to the target using the IPOP [17] overlay network. To the application the system appears as a single cluster using a local area network with private addresses. The system also supports transparent migration of virtual machines. The advantage of this approach is that no changes are required to the application. It is a heavy weight solution, however. Instead of just deploying the application to the Grid sites, VMware must also be deployed, including a copy of the required operating system. In addition, all network traffic is routed using the overlay network, even when two machines are located in the same site. The experiments in [18] shown that this limits the network bandwidth in a single site to 12.5 MBit/s. Even when IPOP [17] is used directly by the application, the network bandwidth is reduced to 61% on a local-area network, and 51% on a wide-area network. VNET [32] is similar to WOW, but uses tunneling instead of peer-to-peer techniques to forward network traffic. Like WOW, VNET shows significant performance degradation in both local and wide-area experiments. VIOLIN [22] and ViNe [33] propose similar solutions. In [25], this performance degradation is solved by only using peer-to-peer techniques for resource discovery and allocation. The applications use regular sockets instead, thereby significantly improving the performance, but also reintroducing the connectivity problems described in this paper.

SmartSockets uses a combination of the approaches described above. It prefers to create direct connections and only uses routing or tunneling as a last resort. This results in a performance on par with regular sockets when possible, but also offers improved connectivity when it is needed.

Several mechanisms exist that allow two machines using NAT to set up a communication channel. STUN [27] only allows the exchange of UDP messages. STUNT [19, 20] and NATBlaster [6] support TCP, but require access to raw sockets, for which special user privileges are needed. All three use external servers to provide information on address and port translation. Both STUN and NATBlaster use port range prediction. The system described in [13] does not require raw sockets, but does not use port range prediction during connection setup. As shown in [19], the connection setup success rate of this approach increased from 45% to 84% when port range prediction was added. Since SmartSockets uses the same mechanism as [13], but includes port range prediction, we also expect the connection setup success rate to be around 84%.

Although SmartSockets was initially designed to increase the connectivity, it is also used to do the exact opposite. By extending the handshake performed in the direct connection layer with a check that selectively refuses connections based on the address of the source machine, a simple firewall can be simulated. This allows a complex network with limited connectivity to be simulated on single cluster. In [11], this mechanism is used to evaluate the effectiveness of peer-to-peer gossiping techniques when machines have limited connectivity. Based on these experiments, the authors propose a new gossiping algorithm, Actualized Robust Random Gossiping (ARRG), that outperforms existing algorithms in situations where the network connectivity is restricted.

In MOB [9], the multi homing support of SmartSockets is used to run cluster to cluster multicast experiments. SmartSockets automatically selects the fast local network for intra-cluster traffic, while only using the regular ethernet between

clusters. This project also uses the firewall simulation described above to divide a single cluster into several smaller ones. By forcing all communication to be routed via a small number of machines, shared links between the clusters can be simulated. By artificially reducing the bandwidth or increasing the latency on those shared links, the robustness of the multicast algorithms can be tested.

6. CONCLUSIONS

In this paper we have introduced an integrated framework, called SmartSockets, which is capable of solving the connectivity restrictions found on many Grid sites, with very little help from the user. We have shown that by using caching, the connection setup time can be reduced significantly. In 30 connection setup experiments, using 6 different sites worldwide, our framework was always capable of creating a connection, requiring a maximum time of 750 milliseconds once the necessary information was cached. A conventional connection could only be created in 6 out of the 30 combinations. By preferring direct connections, the bandwidth and latency offered by SmartSockets is similar to that of a conventional connection (in the situations where a conventional connection can be created).

By using a heuristic that prefers suitable private addresses of a target machine during connection setup, a fast local network is often selected for intra-site communication, thereby potentially improving the application performance. As we have shown, it is essential that an identity check is performed to prevent a connection to a wrong machine.

So far we have only shown the results of low-level performance benchmarks. The following step in our work will be to further evaluate the performance and scalability of SmartSockets using a wide range of parallel and distributed applications. We are also planning to improve the throughput on long distance connections.

7. REFERENCES

[1] Netfilter. http://www.netfilter.org.

[2] SUN Java 5.0. http://java.sun.com.

[3] The Grid5000 system. http://www.grid5000.fr.

[4] The InfiniBand Trade Alliance architecture. http://www.infinibandta.org.

[5] Universal Plug and Play (UPnP). http://www.upnp.org.

[6] A. Biggadike, D. Ferullo, G. Wilson, and A. Perrig. NATBlaster: Establishing TCP connections between hosts behind NATs. In *In Proc. of ACM SIGCOMM Asia Workshop*, April 2005.

[7] N. Boden, D. Cohen, R. Felderman, A. K. and C.L. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, Jan. 1995.

[8] D. Caromel, C. Delbe, A. di Costanzo, and M. Leyton. ProActive: an Integrated Platform for Programming and Running Applications on Grids and P2P systems. *Computational Methods in Science and Technology*, 12, 2006.

[9] M. den Burger and T. Kielmann. "MOB: Zero-configuration High-throughput Multicasting for Grid Applications". In *Proc. of the 16th International Symposium on High-Performance Distributed Computing (HPDC-16)*, Monterey, California, USA, June 2007. Accepted for publication.

[10] A. Denis, O. Aumage, R. F. H. Hofman, K. Verstoep, T. Kielmann, and H. E. Bal. Wide-area Communication for Grids: An Integrated Solution to Connectivity, Performance and Security Problems. In *Proc. of the 13th International Symposium on High-Performance Distributed Computing (HPDC-13)*, pages 97–106, Honolulu, Hawaii, USA, June 2004.

[11] N. Drost, E. Ogston, R. V. van Nieuwpoort, and H. E. Bal. "ARRG: Real-world Gossiping". In *Proc. of the 16th International Symposium on High-Performance Distributed Computing (HPDC-16)*, Monterey, California, USA, June 2007. Accepted for publication.

[12] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631, May 1994. Obsoleted by RFC 3022.

[13] B. Ford, D. Kegel, and P. Srisuresh. Peer-to-peer Communication Across Network Address Translators. In *Proceedings of the 2005 USENIX Technical Conference*, 2005.

[14] P. Francis. Is The Internet Going Nutss? *IEEE Internet Computing*, 7(6):94–96, 2003.

[15] N. Freed. Behavior of and Requirements for Internet Firewalls. RFC 2979, Oct. 2000.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading (MA), USA, 1995.

[17] A. Ganguly, A. Agrawal, P. O. Boykin, and R. Figueiredo. IP over P2P: Enabling Self-configuring Virtual IP Networks for Grid Computing. In *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS-2006)*, April 2006.

[18] A. Ganguly, A. Agrawal, P. O. Boykin, and R. Figueiredo. WOW: Self-organizing Wide Area Overlay Networks of Virtual Workstations. In *Proc. of the 15th International Symposium on High-Performance Distributed Computing (HPDC-15)*, Paris, France, June 19-23 2006.

[19] S. Guha and P. Francis. Characterization and Measurement of TCP traversal Through NATs and Firewalls. In *In Proc. of Internet Measurement Conference (IMC)*, 2005.

[20] S. Guha, Y. Takeda, and P. Francis. Nutss: a SIP-based Approach to UDP and TCP Network Connectivity. In *FDNA '04: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 43–48, New York, NY, USA, 2004. ACM Press.

[21] T. Hain. Architectural Implications of NAT. RFC 2993, Nov. 2000.

[22] X. JIANG and D. XU. VIOLIN: Virtual Internetworking on Overlay Infrastructure. In *Proc. of the 2th International Symposium on Parallel and Distributed Processing and Applications.*, December 2004.

[23] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, July 2005.

[24] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. RFC 1928, Mar. 1996.

[25] Z. Pan, X. Ren, R. Eigenmann, and D. Xu. Executing MPI Programs on Virtual Machines in an Internet Sharing System. In *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS-2006)*, April 2006.

[26] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918, Feb. 1996.

[27] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489, Mar. 2003.

[28] S. Son and M. Livny. Recovering Internet Symmetry in Distributed Computing. In *In Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, May 2003.

[29] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, Jan. 2001.

[30] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, and A. Rayhan. Middlebox Communication Architecture and Framework. RFC 3303, Aug. 2002.

[31] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proc. of the USENIX Annual Technical Conference*, June 2001.

[32] A. SUNDARARAJ and P. DINDA. Towards Virtual Networks for Virtual Machine Grid Computing. In *Proc. of the 3rd USENIX Virtual Machine Research And Technology Symposium (VM 2004).*, 2004.

[33] M. Tsugawa and J. A. Fortes. A Virtual Network (ViNe) Architecture for Grid Computing. In *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS-2006)*, April 2006.

[34] R. V. van Nieuwpoort, J. Maassen, A. Agapi, A.M. Oprescu, and T. Kielmann. Experiences Deploying Parallel Applications on a Large-scale Grid. In *Proc. of EXPGRID - Experimental Grid Testbeds for the Assessment of Large-scale Distributed Applications and Tools. Workshop in conjunction with (HPDC-15)*, Paris, France, June 2006.