Wide-Area Communication for Grids: An Integrated Solution to Connectivity, Performance and Security Problems

Alexandre Denis[1] Olivier Aumage[2] Rutger Hofman[3] Kees Verstoep[3] Thilo Kielmann[3] Henri E. Bal[3]

[1] IRISA/Univ. Rennes 1, Rennes, France, alexandre.denis@irisa.fr
[2] INRIA LaBRI, Talence, France, olivier.aumage@labri.fr
[3] Vrije Universiteit, Amsterdam, The Netherlands, {rutger,versto,kielmann,bal}@cs.vu.nl

http://www.cs.vu.nl/ibis/

Abstract

Grid computing applications are challenged by current wide-area networks: firewalls, private IP addresses and network address translation (NAT) hamper connectivity, the TCP protocol can hardly exploit the available bandwidth, and security features like authentication and encryption are usually difficult to integrate. Existing systems (like GridFTP, JXTA, SOCKS) each address only one of these issues. However, applications need to cope with all of them, at the same time. Unfortunately, existing solutions are often not easy to combine, and a particular solution for one subproblem may reduce the applicability or performance of another.

In this paper, we identify the building blocks that are needed for connection establishment and efficient link utilization. We present an integrated solution, implemented within the Java-based Ibis runtime system. Our NetIbis implementation lets applications span multiple sites of a grid, and copes with firewalls, local IP addresses, secure communication, and TCP bandwidth problems.

1 Introduction

The promise of grid computing is to let performance-hungry applications simultaneously tap the aggregated power of multiple sites of a grid environment. For this purpose, grid applications need to communicate across the individual sites, using available wide-area connections of the Internet.

Grid application runtime systems originate from traditional parallel computing. Here, all nodes are directly connected using high-performance local or system area networks (LANs or SANs). Within a parallel or cluster computer, connectivity is unrestricted and communication protocols are provided that can successfully exploit the bandwidth capacity of the LAN or SAN. However, integrating multiple parallel computers across wide-area networks (WANs) causes additional problems for application runtime systems:

Connectivity To protect their machines from intruder attacks, many site administrators have drastically restricted the connectivity to the Internet. Many sites are using firewall routers, non-routed private networks [18], or hide their machines via *Network Address Translation* (NAT) [5]. The most common way to achieve connectivity in the presence of firewall routers is to selectively open some TCP ports. However, this approach violates site security and requires manual interaction of site administrators. The SOCKS protocol [14] allows building general-purpose TCP proxies in combination with firewalls and NAT. SOCKS enables secure connectivity but it is known for inferior performance.

Performance Because of its congestion-control mechanism, TCP/IP is the only ubiquitously available protocol suite. Also, because of its congestion-control mechanism, vanilla TCP can hardly exploit the bandwidth capacity of WAN connections. One option to improve TCP performance in WANs is to use multiple TCP streams in parallel. The Globus implementation of GridFTP [1] is probably the best-known tool implementing this approach. Alternatively, WAN performance can be improved using data compression, as implemented, e.g., in the AdOC library [11].

Security As WAN connections across the Internet are vulnerable to attackers, many applications require authentication of communication partners and privacy based on encryption, features that are not considered at all in traditional application runtime systems. For TCP, authentication and encryption is implemented in the Transport Layer Security protocol (TLS) [4], a successor of the Secure Sockets Layer.

All of the solutions mentioned above only address individual problems of WAN communication. Applications, however, need to address all these problems. Unfortunately, existing solutions are often not easy to combine, and a particular solution for one subproblem may reduce the applicability or performance of another. In this paper, we present an *integrated* solution addressing all WAN-specific problems for grid application runtime systems outlined above. For example, our NetIbis implementation of the Ibis runtime system allows the use of data compression over parallel TCP streams through firewall routers. NetIbis uses TCP splicing for connection establishment through firewalls, and uses its own dedicated proxies for message routing on machines that lack direct connectivity. Orthogonal to connection establishment, NetIbis can use parallel streams, as well as data compression and encryption.

In Section 2 we identify the building blocks for WAN communication. In Sections 3 and 4, we discuss possible methods for connection establishment and utilization, respectively. Section 5 presents our Ibis grid programming environment in which we have implemented our integrated WAN communication methods. We evaluate the performance of our new Ibis implementation in Section 6. Section 7 discusses related work, and Section 8 outlines conclusions and directions for further work.

2 Building Blocks for WAN Communication

In a grid application, individual processes communicate with each other. We use *connections* as basic communication abstractions. A connection is a logical communication channel that connects endpoints within the grid application. Such a *logical* (application-level) connection may be implemented using available transport protocols, for example using one or more TCP connections. However, the underlying transport protocol does not necessarily need to be connection oriented. For communication in grid environments, we can identify the following classes of connections:

Data Connections are used to exchange application data between endpoints. Application processes typically use many of them, either to communicate with different remote endpoints, or to allow individual threads to communicate independently. Data connections can be established either statically during application startup, or dynamically at run time.

Service Connections are used to exchange control information related to the runtime system, e.g., requests for data connection establishment. Application processes typically have statically initialized service connections. However, dynamic scenarios are also possible, e.g., when processes join or leave an application at runtime.

Bootstrap Connections are used to initialize connectivity of an application process with its peers. This includes the exchange of addresses and port number information. Bootstrap and service connections may actually coincide.

These three classes of connections have different requirements to the runtime system. Bootstrap connections have to be established without any pre-existing connection between the endpoint hosts. This may require using a central registry or application-level relays. Service or data connections can be established via connections already existing between the endpoints. A negotiation between the endpoints might take place to determine performance or security properties.

For data connections, the most important performance requirement is the achievable bandwidth. Achieving low communication latency, either for one-way or roundtrip messages, is usually somewhat less important in a grid environment. Another property is the connection establishment delay. Its importance depends on how frequently a particular application creates new data connections. As data, service, and bootstrap connections between a pair of hosts might be implemented differently, their respective performance properties might vary.

Connections, as an abstract concept, have methods to send data to and to receive data from another node, independent of the actual implementation in terms of underlying transport protocol entities. Establishment and utilization of a connection are orthogonal operations:

Connection Establishment is the action of creating a path between two nodes. It has to resolve the connectivity issues caused by the available transport protocols and mechanisms like firewalls and private IP addresses. Also, during connection establishment, security properties can be negotiated and implemented.

Link Utilization is the transfer of data across an already established connection. Performance properties (like compression) can be applied independently after connection establishment. For clarity, we use the term *link* for an established connection.

3 Connection Establishment Methods

For connection establishment, we distinguish between standard TCP client/server handshake, symmetrical TCP splicing, and relayed connection establishment. The goal is to connect under various conditions and for the different purposes of data, service, and bootstrap connections.

3.1 Standard TCP client/server handshake

The standard way of establishing a TCP connection is a client/server handshake; see the left part of Figure 1: host

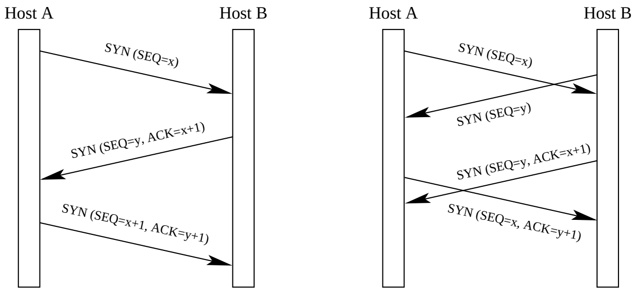


Figure 1. TCP connection establishment. Regular client/server handshake (left); TCP splicing (right).

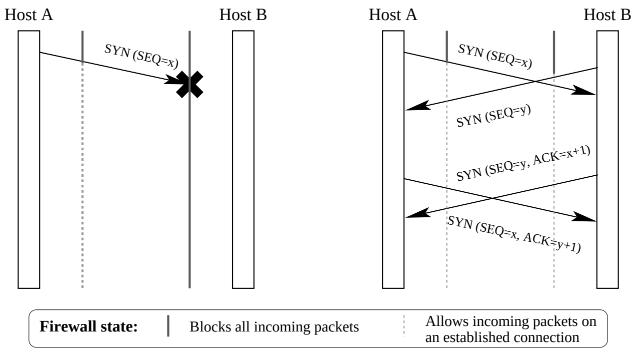


Figure 2. TCP connection establishment with firewalls. Regular client/server handshake fails (left); TCP splicing succeeds (right).

A is the client, host B is the server. At user level, the server invokes `listen` to express that it is ready to receive connection requests from clients. The client invokes `connect` to send a connection request. The last step is an invocation of `accept` from the server to accept the incoming connection.

At the network level, the connection request is a `SYN` packet sent by the client to the server. When the server accepts the incoming connection, it sends a `SYN` packet with an acknowledgment (`ACK`) back to the client. As a last step, the client acknowledges the `SYN` from the server.

3.2 TCP splicing

In addition to the asymmetrical client/server handshake, the TCP standard [17] defines *simultaneous initiation*, also called "*simultaneous SYN*" or "*TCP splicing*" in the literature. This way of establishing a connection is symmetrical. At user level, both sides invoke `connect` at the same time to connect to each other. Neither of them invokes `listen` or `accept`.

At the network level, this mechanism is depicted in the rightmost part of Figure 1. Both sides send a `SYN` packet to request a connection. Then, both sides acknowledge the connection with a `SYN` packet containing an `ACK`. As a result, the TCP connection is established, exactly as with the client/server handshake. This mechanism requires a negotiation between both endpoints, however. Since the nodes have to invoke `connect` simultaneously —both act as clients and do not know whether the other is ready—, brokering is required.

Firewalls and TCP splicing. TCP splicing is especially interesting for connections crossing firewalls. Firewalls introduce a change in the connection establishment schemes. Firewalls are devices which selectively block packets to prevent malicious users to connect to the hosts inside a given site. Nowadays, most firewalls are *stateful*: they usually allow all outgoing packets and drop all incoming packets, except packets belonging to an already established connection. Thus, the firewall allows connections going out of the site and blocks connections coming into the site. Therefore, a firewall-protected site can contain clients but no servers (unless explicitly configured otherwise).

Figure 2 depicts a client/server handshake and a TCP splicing connection establishment between two hosts in different sites, each site having a firewall. On the left, the client/server handshake fails because the firewall of site B blocks the incoming connections to host B. On the right, TCP splicing succeeds in establishing a connection through the firewalls. When the simultaneous `SYN` packets are sent, each firewall goes into a state where incoming packets are allowed for the given port number. Indeed, both firewalls consider the connection as an *outgoing connection*, and therefore allow it.

3.3 Relayed connection establishment

In some circumstances, a direct TCP connection — client/server or splicing— is not possible. The direct connection may be made impossible by a severe firewall (e.g., one which even forbids outgoing connections except through a well-controlled proxy), or by an implementation of network address translation (NAT) that is incompatible with TCP splicing. In these cases, the connections have to go through a *relay* or *proxy* running on a *gateway host*: a machine connected both inside and outside of the firewall. Here, the client connects to the proxy, and the proxy connects to the server on behalf of the client.

TCP proxies. Proxies are widespread, especially for Web and FTP protocols. The main versatile TCP proxy is

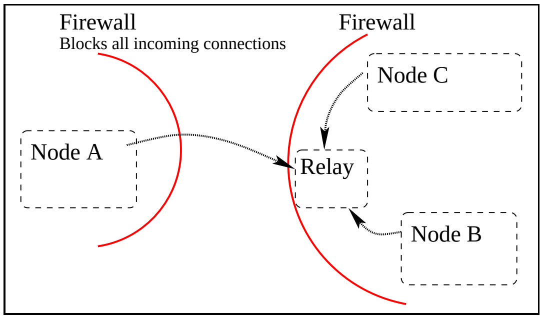


Figure 3. Routed messages: all nodes are connected to a relay located on a gateway machine accessible from the outside; the relay forwards messages to their final recipient.

	client/ server	TCP splicing	TCP proxy	routed messages
Crosses firewalls	no	yes	yes	yes
NAT support	client	partial	yes	yes
For bootstrap	yes	no	no	yes
Native TCP	yes	yes	yes	no
Relayed	no	no	yes	yes
Needs brokering	no	yes	yes	no

Table 1. Connection establishment methods summary.

“SOCKS”, which also has been standardized [14]. The behavior of a SOCKS proxy is mostly transparent: a client connects to the proxy; it sends the address of the server to connect to; once the proxy has established a connection with the server, the link may then be used exactly like a direct TCP connection. A SOCKS proxy allows an outgoing connection to cross a firewall; it also allows hosts with private IP addresses on sites without NAT to connect to the outside.

Though it is mostly transparent from the *client* side, a SOCKS proxy is less transparent if the *server* is behind the proxy. In fact, in this case clients have to connect to a dynamically-allocated port number on the proxy itself, which requires some information exchange. Therefore, if a connection needs to be established between endpoints that are both behind SOCKS proxies, some negotiation is required. This is why connections through SOCKS are not suitable as bootstrap links.

Specific proxy: routed messages. For bootstrap connections behind firewalls, we propose a technique called *routed messages*. It is based on a relay located on a gateway machine visible from the Internet. When a node is started, it connects to the relay as depicted in Figure 3. When a node wants to establish a connection to another node, it sends a request to the relay, which forwards the request to its final recipient. This is made possible by the connection established by each node to the relay. The response and all data messages are routed similarly by the relay.

Every node connected to the Internet —directly, through NAT, or through a SOCKS proxy— can connect to the relay and thus has a bootstrap connection. Connections built with routed messages are likely to exhibit relatively poor performance. However, their goal is mainly to serve as a bootstrap link for establishing other connections requiring initial negotiation (e.g., TCP splicing). Routed messages are not supposed to be used for data, except in extreme cases when there is no other connection method possible.

3.4 Discussion

Table 1 summarizes the properties of the various connection establishment methods discussed. The first three properties describe under which circumstances a connection is possible:

Crosses firewalls indicates whether a connection may be established between sites protected by a firewall blocking incoming connection requests. All methods except client/server TCP can cross firewalls.

NAT support indicates whether a connection is possible if a host uses network address translation. Both connection methods for TCP (client/server and splicing) work in some cases with NAT, but cannot deal with every case. Client/server only works when the client does NAT, not the server; splicing works with NAT only with NAT gateways based on a known and predictable port translation rule.

Usable for bootstrap indicates whether the method can be used without negotiation, i.e., when there is no pre-existing connection between the hosts. Only client/server TCP and routed messages are suitable for bootstrap.

The next two properties are related to performance criteria and the way the connection created can be utilized:

Native TCP — All establishment methods except routed messages create native TCP sockets. In contrast, routed messages links must be utilized by a specific method. Only native TCP connections can be composed with the utilization methods described in the next section.

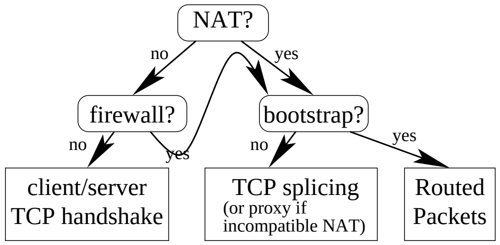


Figure 4. Choosing a connection establishment method.

Relayed — Because the data of several nodes are routed through a unique relay, the relay itself is likely to be a bottleneck, lowering the achievable bandwidth. Since the relay adds a receipt/send on the route between the sender and the receiver, the use of a relay is also likely to raise the communication latency.

To choose a communication establishment method, the first criterion is connectivity. Given the network topology (firewall, NAT) and nature of the link being built (bootstrap or not), the choice of connection methods is narrowed down to a set of *possible methods*.

The second criterion is performance, and in particular the utilization methods usable on the links built. In general, the native TCP and non-relayed methods are preferable for composability and better performance, respectively. Finally, methods without brokering are preferable over the ones requiring it, since the latter are likely to exhibit a higher connection establishment delay due to the negotiation phase.

When combining these criteria, we get the following precedence list: client/server TCP, TCP splicing, TCP proxy, routed messages, which is depicted in Figure 4. The best connection establishment method is the first possible (according to firewalls, NAT and bootstrap) from this list.

4 Link Utilization Methods

In this section, we describe the link utilization methods which can be used on a WAN. The goal of these methods is to send and receive data in an efficient and secure way, over links established with the methods described above. All utilization methods discussed can be combined: for example, it is possible to use compression over secured parallel streams. However, to do so, a carefully designed runtime system is necessary, like our NetIbis implementation described in the next section.

4.1 Plain TCP socket

The simplest utilization method for a TCP connection is to invoke `send` and `recv` for each packet of data. However, this method usually exhibits poor performance with small packets. Unfortunately, small packets are used frequently in parallel applications.

A solution is to deal with this problem in the interface of the communication framework. Here, data is aggregated in buffers. A buffer is sent off due to overflow or due to an explicit flush by the user. TCP does have a built-in mechanism for packet aggregation, called TCP_DELAY, but this is unfortunately unfit for parallel programming since it adds significantly to the latency. The alternative, buffering in user space in combination with an explicit flush, allows disabling TCP_DELAY, and ensures a high bandwidth (around 11.8 MB/s on a 100 Mbit/s Ethernet LAN) in combination with a minimal latency.

4.2 Parallel Streams

Due to its sliding-window mechanism for congestion control, TCP's achievable bandwidth is rather limited, especially in WANs with high packet latency. To overcome this limitation, the send window size has to be enlarged beyond the amount of data that can be sent in the interval determined by the product of bandwidth and packet roundtrip-time. For WANs, the necessary window size often lies beyond the limits imposed by the operating system. But even with TCP-modifications like *window scaling* [10], achieving good TCP performance on a high-latency WAN is still difficult, due to TCP's inert recovery from lost packets.

On such high latency WANs, using multiple TCP streams —or *parallel streams*— for a single logical connection can improve the achievable bandwidth by increasing the window size beyond the operating-system limits. Moreover, using multiple streams has been shown to reduce the impact of packet loss, resulting in a higher overall bandwidth [19]. In order to use parallel streams, sender and receiver have to fragment and multiplex the data over the underlying, individual TCP streams.

4.3 Compression

With a fast CPU and a slow network, it may be worthwhile to compress the data before sending it, to maximize the effective throughput. Most compression techniques have a tunable compression level, where higher levels result in a better compression ratio but also cause higher CPU consumption. Some advanced mechanisms of on-the-fly compression, like AdOC [11], are able to dynamically adapt the compression level according to their environment. However, in our measurements with the `zlib` compression library only the first level of compression turned out to be

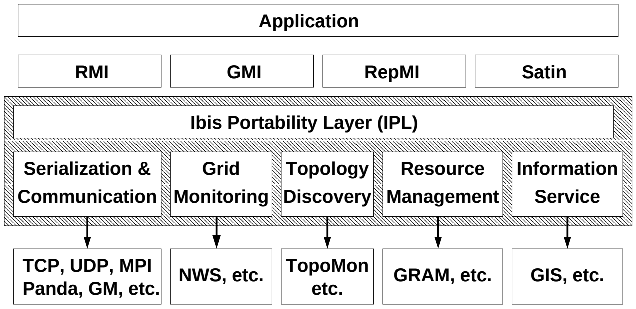


Figure 5. Design of Ibis. The various modules can be loaded dynamically, using run time class loading.

useful: higher levels consumed much more CPU time (too much for real-time application use) for only a limited gain in compression.

4.4 SSL/TLS security

When data privacy is critical and the connections span across multiple sites, users may require authenticated and encrypted communication. This can be performed through the use of the standard SSL/TLS [4] (*Transport Layer Security*) infrastructure, performing data encryption and peer authentication over a socket connection. SSL/TLS security may be added over a link built with any of the establishment methods described in Section 3.

5 The Ibis Grid Programming Environment

We have implemented a runtime system providing efficient communication for wide-area networks within our Ibis grid programming environment [22]. The global structure of the Ibis system is shown in Figure 5. Central is the Ibis Portability Layer (IPL), a thin interface layer. The IPL can have different implementations, that can be selected and loaded into the application *at run time*. The IPL defines serialization and communication and provides interfaces to grid services such as topology discovery and monitoring. Ibis currently implements four application programming models on top of IPL: RMI, group method invocation (GMI), replicated objects (RepMI), and Satin, a divide-and-conquer programming model. All of them have efficient implementations for grids.

The IPL provides one elementary communication abstraction, *unidirectional message channels*. Endpoints of communication are *send ports* and *receive ports*. For supporting group communication, one send port might be connected to multiple receive ports, and vice versa.

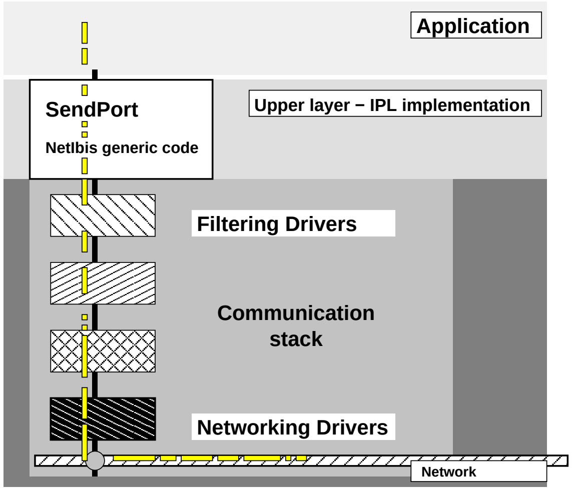


Figure 6. A communication driver stack in NetIbis.

Unlike many message passing systems, the IPL has no concept of hosts or threads, but uses location-independent *Ibis identifiers* to identify Ibis nodes. A registry, called *Ibis Name Service*, is provided to locate receive ports, allowing to bootstrap connections.

5.1 The NetIbis Implementation

NetIbis is the implementation of the Ibis Portability Layer (IPL) used for the work presented here. It is basically organized in two independent layers (see Figure 6). The upper layer implements the IPL while the lower layer, a stack of *drivers*, is responsible for matching the specific application needs with the underlying networking hardware properties. NetIbis has been designed to make the communication paths between send and receive ports completely configurable, either by configuration file or by run-time properties.

The communication paths are built using one or more *drivers* organized as a *driver tree*. Each driver provides one single *added value*, either a *filtering* capability, like Java object serialization, or a *networking* capability like block-oriented transfer over Myrinet. NetIbis drivers have uniform interfaces which makes them interchangeable, allowing to compose complex communication stacks. Each filtering driver may have one or more sub-drivers. Each configured send or receive port communication stack thus is a tree of zero or more filtering drivers with each leaf being a networking driver.

Each driver may itself provide *input* and *output* functionality. Together, input and output drivers are used to actually carry network connections. Each NetIbis connection is

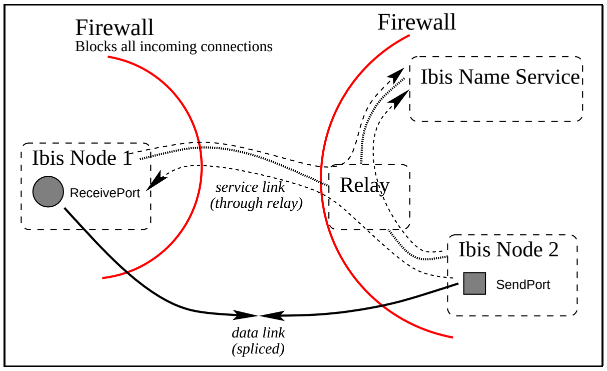


Figure 7. Service links are routed through the relay. The data link uses TCP splicing with brokering through the service link.

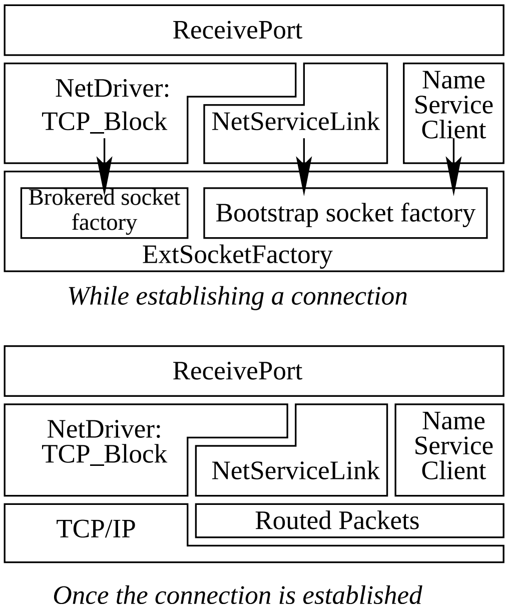
an isolated, unidirectional, FIFO-ordered virtual networking link from a source to a destination port.

5.2 WAN Communication in NetIbis

Resolving the WAN connection and communication issues discussed can be simplified significantly by employing a framework that explicitly supports the separation of connection establishment and link utilization. For example, when link utilization is implemented independent of connection establishment, it suddenly becomes much easier to *compose* different communication layers. In NetIbis, this principle of separation between connection establishment and link utilization has been implemented, using *socket factories* for connection establishment, and networking and filtering *drivers* for link utilization, respectively. Although the NetIbis architecture also supports networking drivers for other protocols than TCP, we only used our block-oriented TCP driver (`TCP_Block`), because of TCP's ubiquitous availability.

Data links and service links. There are three kinds of links in NetIbis. The *data links* actually carry data for a connection of a given message channel. Each data link has an associated *service link*, used for driver assembly consistency on both endpoints, and connection establishment negotiation. Finally, *name service links* are links between computing nodes and the name server.

Service links and name service links have low performance requirements, although their latency influences the connection establishment delay for data links; service links require a connection establishment method able to bootstrap connections. In contrast, data links are expected to use the method with the best achievable performance, and may use brokering through the service link.



While establishing a connection

Once the connection is established

Figure 8. Software layers in NetIbis corresponding to node 1 in example of Figure 7.

An example is given in Figure 7. Service and name service links use the *routed messages* method, through the relay. The data link uses TCP splicing for direct connection establishment, and the standard TCP utilization method with aggregated data blocks.

Socket factories. When a networking driver needs to establish a connection, it delegates this to a socket factory which builds the connection using the decision tree shown in Figure 4.

Figure 8 shows the software layers corresponding to the scenario of Figure 7 while establishing connections (at the top) and once the connection is established (at the bottom). For connection establishment, the name service and service links rely on connections built by the *bootstrap socket factory*; the networking driver `TCP_Block` relies on the *brokered socket factory* for its data link, since negotiation can take place on the service link.

Link utilization methods. The various link utilization methods presented in Section 4 are implemented as NetIbis drivers. The basic networking driver for TCP is `TCP_Block`. Compression is implemented as a filtering driver based on the `zlib`. This filter is composable with other drivers. All the networking drivers rely on the brokered socket factory to establish their data links, and may thus be composed with the various connection establishment methods.

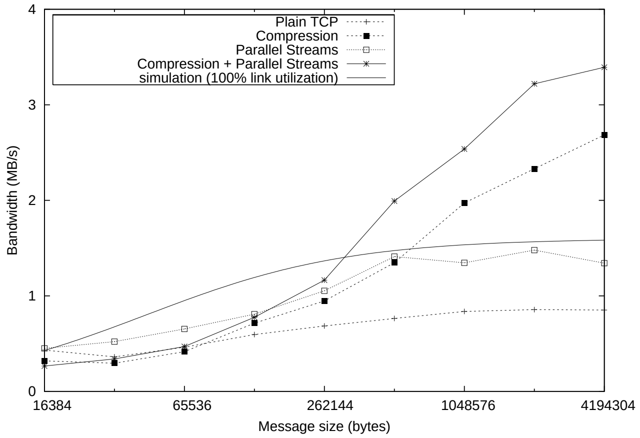


Figure 9. Bandwidth obtained with various methods between Amsterdam and Rennes.

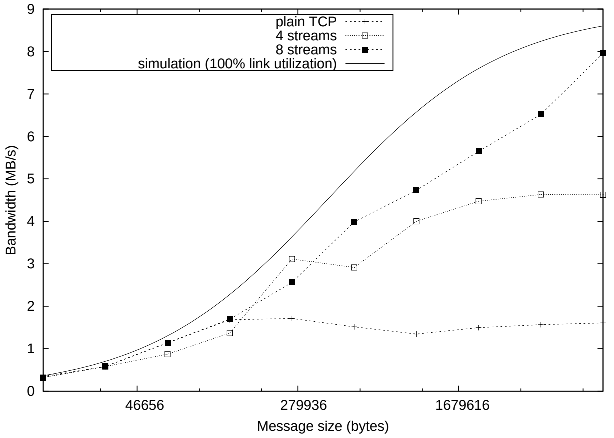


Figure 10. Bandwidth obtained with TCP and parallel streams between Delft and Sophia.

We also plan to implement an encryption driver as a filtering driver using SSL. SSL implementations require either sockets or standard Input/OutputStreams as their basic data exchange type. The encryption driver would have to convert between NetIbis' basic data exchange type, ByteBuffer, and Input/OutputStreams.

6 Evaluation

We evaluated the integrated approach using our implementation in NetIbis on various WAN configurations.

Qualitative results. We deployed NetIbis on multiple sites in the Netherlands, France, Poland and Germany. Most of the sites are protected by stateful firewalls, and some use NAT and private IP addresses. In all cases, we were able to establish a connection from every node to every other node without opening ports in firewalls. Most of these connections are TCP (by means of client/server or splicing), and thus can reach decent performance with the standard utilization methods.

In the presence of firewalls, NetIbis chooses *routed messages* for service links and TCP splicing for data links. Connections through firewalls were always successful with splicing, also in combination with parallel streams.

We were less lucky with some of the NAT implementations, however. It turned out that several NAT implementations were not fully standards-compliant, and did not let TCP splicing connections across, even though they should have. In some cases during our experiments, there was no choice but to revert to a standard SOCKS proxy.

Quantitative results. For our quantitative results, we measured the performance of the various communication methods on two WANs.

The first link used for our performance measurement was the high-latency, low-bandwidth WAN between Amsterdam and Rennes. The capacity of the link is 1.6 MB/s with a typical latency of 30 ms. As shown in Figure 9, the achievable bandwidth for plain TCP was 0.9 MB/s (56% of the capacity). With 4 parallel streams, the bandwidth reached 1.5 MB/s (93%), while the latency remained unchanged. With `zlib` compression level-1 enabled, the bandwidth reached 3.25 MB/s (203% of the capacity). When we combined both compression and parallel streams, the peak bandwidth reached 3.4 MB/s with a better overall performance than with compression only.

The second link used for our performance measurement was the high-latency, high-bandwidth WAN between Delft (The Netherlands) and Sophia (France). The capacity of the link is 9 MB/s with a typical latency of 43 ms. As shown in Figure 10, with plain TCP the achievable bandwidth was 1.7 MB/s (only 19% of the capacity). With 4 parallel streams, the bandwidth reached 4.6 MB/s (51%); with 8 parallel streams it even reached 7.95 MB/s (88%). On this fast link, compression *degraded* performance: the bandwidth reached 5 MB/s with compression and 3.5 MB/s with both compression and parallel streams. Additional measurements showed that compression could improve the bandwidth for networks with a capacity up to 6 MB/s; beyond this threshold, compression degrades the performance, with the CPUs used in this particular case.

7 Related Work

Many researchers are working on improving the behavior of transport protocols, especially in the context of wide-area networks [9]. The results of these efforts are orthogonal to our work and can be subsumed automatically, once deployed. However, due to the site autonomy which is typical in grid environments, extensions to standard transport protocols cannot easily be enforced universally. Instead, grid application runtime systems will have to use whatever implementation is available at the given sites.

An alternative approach to resolving connectivity problems in WANs is to build so-called overlay networks, consisting of application-level relays performing data routing on top of standard Internet protocols. A prominent example is implemented by Project JXTA [7], a communication infrastructure designed for peer-to-peer networks. As message forwarding in JXTA is performed using application-level relays, similar to our *routed messages* method for bootstrap links, it will presumably not be suitable for high-performance data connections, however [8].

The separation between socket creation and utilization is already present in the Java language environment with its *SocketFactory* pattern. However, the built-in *SocketFactory* does not support negotiated methods like splicing, and all connections must use the same method. Our proposal goes further by distinguishing between several types of connections and by introducing negotiation for connections.

The Java CoG Kit [24] provides access to Grid services through the Java framework. Components delivering client and limited server side capabilities are provided. So far, the Java CoG Kit has relied on the communication mechanisms provided in Java by means of standard TCP sockets. As we argue in this paper, it would be very useful to resolve network connectivity and performance problems by an integrated solution. As such, adding NetIbis-like functionality to the Java CoG Kit would be an interesting option. Multiple higher-level distributed computing paradigms could then portably and efficiently be implemented using the Java CoG Kit, without having to resolve the same WAN communication issues over and over.

A widely used grid programming model is MPI. The most popular implementation for grids is MPICH-G2 [12], an MPI implementation over Globus. However, WAN communication methods in MPICH-G2 are rudimentary; it does not cross firewalls, and the only advanced link utilization method for WAN is a fixed-ratio compression. PACX-MPI [6] is an implementation of MPI that has been designed from scratch for grids. For each grid site, PACX-MPI uses a dedicated gateway node for relaying messages across the WAN. This static configuration solves some of the existing connectivity problems. However, some refactoring would be needed to implement the whole spectrum of possibilities implemented within NetIbis. Finally, our own MagPIe library [13] optimizes the performance of MPI's collective operations in grid systems. It addresses neither security nor connectivity as it assumes all connections to be established in advance.

Both Ibis and MPI are designed for rather fine-grained communication within grid applications. Alternatively, systems like Ninf-G [20] and NetSolve [2] are facing fewer connectivity problems as they solely rely on RPC-based communication without the notion of long-lived connections. This simplification comes at the price of supporting only coarser-grained applications that can tolerate higher connection establishment delays for the individual remote procedure calls.

8 Conclusions and Future Work

Applications are challenged by the lack of connectivity, performance, and security in current grid environments. Existing systems typically only implement a solution to individual communication subproblems. Unfortunately, it is often difficult to integrate existing communication subsystems, or to combine them efficiently. In this paper, we have presented an integrated solution, allowing, for example, the use of data compression over parallel TCP streams through firewall routers. We have implemented our solution within the Java-based Ibis grid programming environment.

We have identified the building blocks for WAN communication in grids by characterizing different classes of connections and their requirements. Most importantly, we consider connection establishment and link utilization as two orthogonal concepts, allowing a combination of different solutions according to the needs of each individual connection.

For connection establishment, we have demonstrated how TCP splicing and relayed connection establishment can complement the standard TCP client/server handshake to achieve connectivity in all settings that can be found in current wide-area networks.

For link utilization, parallel TCP streams as well as adaptive data compression can be integrated with connections, independent of their establishment mechanism. Our performance evaluation has shown that Ibis can achieve bandwidth close to the theoretical maximum when deploying parallel TCP streams. Likewise, we have demonstrated significant bandwidth improvements using data compression.

So far, the NetIbis system implements the basic mechanisms to enable WAN communication in grids. The following step in our work is to combine these mechanisms with grid resource management and information systems. This combination will allow the automated selection of the proper communication methods for given WAN settings. Also, parameter adaptation, like selection of the optimal number of parallel TCP streams [23] or the dynamic enabling or disabling of compression will then become possible. To validate our approach within a second implementation, we will integrate our WAN communication methods into the PadicoTM [3] runtime system which is implemented in C. Such an implementation will give further insights into the performance aspects of our methods.

Acknowledgments

Part of this work has been supported by the European Commission, grant IST-2001-32133 (GridLab) and by the Netherlands Organization for Scientific Research (NWO) grant 612.060.214 (Ibis: a Java-based grid programming environment). We would also like to thank Mathijs den Burger, Niels Drost, Ceriel Jacobs, Jason Maassen, Rob van Nieuwpoort, Maik Nijhuis, and Gosia Wrzesińska for their contributions to the Ibis code. Christoph Schlechtingen (Siegen University) generously gave us access to machines in a NAT environment.

References

[1] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. GridFTP Protocol Specification. GGF GridFTP Working Group Document, 2002.

[2] D. Arnold, H. Casanova, and J. Dongarra. Innovations of the NetSolve Grid Computing System. *Concurrency: Practice and Experience*, 14(13–15):1457–1479, 2002.

[3] A. Denis, C. Pérez, and T. Priol. PadicoTM: An open integration framework for communication middleware and runtimes. *Future Generation Computer Systems*, 19(4):575–585, May 2003.

[4] T. Dierks and C. Allen. The TLS Protocol Version 1.0. Request for comments 2246, IETF, Jan. 1999. http://www.ietf.org/rfc/rfc2246.txt.

[5] K. Egevang and P. Francis. The IP Network Address Translator (NAT). Request for comments 1631, IETF, May 1994. http://www.ietf.org/rfc/rfc1631.txt.

[6] E. Gabriel, M. Resch, and R. Rühle. Implementing MPI with Optimized Algorithms for Metacomputing. In *Message Passing Interface Developers and Users Conference (MPIDC)*, pages 31–41, Atlanta, Mar. 1999.

[7] L. Gong. Project JXTA: A technology overview. Technical report, Sun Microsystems, Palo Alto, USA, Oct. 2002. http://www.jxta.org/project/www/docs/TechOverview.pdf.

[8] E. Halepovic and R. Deters. JXTA performance study. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, B.C., Canada, Aug. 2003. IEEE Computer Society.

[9] *IEEE Infocom 2004*, Hong Kong, Mar. 2004. http://www.ieee-infocom.org/2004/.

[10] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. Request for comments 1323, IETF, May 1992. http://www.ietf.org/rfc/rfc1323.txt.

[11] E. Jeannot, B. Knutsson, and M. Björkman. Adaptive Online Data Compression. In *11th International Symposium on High-Performance Distributed Computing (HPDC11)*, pages 379–388, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[12] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, May 2003.

[13] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MAGPIE: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–140, Atlanta, GA, May 1999.

[14] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. Request for comments 1928, IETF, Mar. 1996. http://www.ietf.org/rfc/rfc1928.txt.

[15] J. Maassen, T. Kielmann, and H. Bal. GMI: Flexible and Efficient Group Method Invocation for Parallel Programming. In *In proceedings of LCR-02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, pages 1–6, Washington DC, March 2002.

[16] J. Maassen, T. Kielmann, and H. E. Bal. Parallel Application Experience with Replicated Method Invocation. *Concurrency and Computation: Practice and Experience*, 13(8-9):681–712, 2001.

[17] J. Postel. Transmission Control Protocol. Request for comments 793, IETF, Sept. 1981. http://www.ietf.org/rfc/rfc0793.txt.

[18] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. Request for comments 1918, IETF, Feb. 1996. http://www.ietf.org/rfc/rfc1918.txt.

[19] V. Sander et al. Grid high performance networking research group, working draft. Technical Report draft-ggf-ghpn-netissues-1, Global Grid Forum, Sept. 2003. http://forge.gridforum.org/projects/ghpn-rg/.

[20] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 2003.

[21] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient Load Balancing for Wide-area Divide-and-Conquer Applications. In *Proceedings Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 34–43, Snowbird, UT, June 2001.

[22] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, Seattle, Washington, USA, November 2002.

[23] S. Vazhkudai, J. M. Schopf, and I. Foster. Predicting the Performance of Wide Area Data Transfers. In *International Parallel and Distributed Processing Symposium (IPDPS 2002)*, 2002.

[24] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):643–662, 2001.