A Reliable and Fast Data Transfer for Grid Systems Using a Dynamic Firewall Configuration

T. Oistrez, E. Grünter, M. Meier, and R. Niederberger

Research Centre Juelich, Juelich, Germany
{t.oistrez,e.gruenter,m.meier,r.niederberger}@fz-juelich.de
http://www.fz-juelich.de/jsc

Abstract. Firewalls separate areas of different security requirements. This major task leads to problems regarding the network connectivity and performance of various applications. In particular within distributed systems, like a Grid an unobstructed communication, which is essential for using distributed resources is not possible. Furthermore Grid applications often use multiple ports dynamically and in parallel. This raises the challenge of a dynamic configuration of firewalls. This paper shows a solution based on UDP hole punching and describes the implementation of a UNICORE transfer service using this technology to perform direct high speed file transfers.

1 Firewalls and Filtering of Network Traffic

Firewalls are used to separate areas of different security policies from each other. Their major task is to protect computing resources against unauthorized access and misuse. In order to achieve this task the firewall system processes certain information which is used as a basis to decide if a message may pass through the firewall. The firewall administrator defines a ruleset which represents the implementation of the local security policy. The network traffic is divided into classes of those packets that will be forwarded to the destination and those which will be rejected. This ruleset is a baseline to different tests that will be applied to each incoming packet. The firewall checks IP addresses and ports of the appropriate protocol headers. In addition, stateful packet inspection engines use connection states.

Firewalls store state information primarily when a TCP stream is discovered because TCP is a connection oriented protocol. Connections may be in one of four different states: half open, established, half closed and closed state. Therefore it is reasonable to differentiate which state a connection has entered. Additionally, TCP is a reliable protocol, i.e. if any TCP segment is lost during transfer a retransmission is triggered due to missing acknowledgments discovered at the sender. Further information can be found in [1].

The User Datagram Protocol (UDP) is a non-reliable and non-connection oriented transport layer protocol. An application that uses UDP has to make sure that the data is successfully transmitted. Although no connections exist, firewalls use a simple mechanism to simulate a connection. Figure 1 shows a

E. César et al. (Eds.): Euro-Par 2008 Workshops, LNCS 5415, pp. 94–102, 2009.

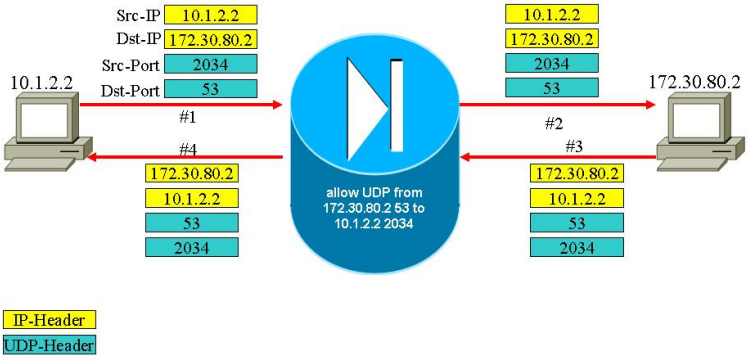

Fig. 1. Firewalls and UDP

client behind a firewall being allowed to send UDP packets to outside, which sends a UDP datagram to a DNS server outside the companies network.

The client generates the UDP datagram and sends it to the firewall (Figure 1 #1). The firewall examines the datagram and forwards it to the destination (#2). According to the information that has been gathered the firewall adds an entry to its connection table including source IP address, source port, destination IP address, and destination port. However, this results in a dynamically generated access rule like

allow UDP from 172.30.80.2 port 53 to 10.1.2.2 port 2034

which is valid for a certain configurable period of time. This rule guarantees that as long as the timeout has not been reached, UDP replies from the server (#3) to the client can traverse the firewall (#4).

2 Grid Applications and Firewalls

A Grid is a distributed system which makes resources like compute power, storage capacity, and distributed data available to its users. It forms an union of geographically distributed, independent organizations sometimes referred to as virtual organization. The usage of available resources takes place statically or dynamically at run-time according to the requirements of the user and/or the application.

Grid applications often need high transfer rates and small latencies. Moreover, these applications transfer large data sets which must arrive reliably and as fast as possible at the destination. One approach to satisfy these needs is the usage of several parallel connections. GridFTP [2] can serve as an example here: it mandates that multiple parallel data connections are always established from the sender to the receiver. Thus, a server running GridFTP must be accessible from outside on a broad port-range. Statically opening these ports at the firewall leads to severe security problems because the open server ports can be used by

malicious software while not in use by a current GridFTP transfer. GridFTP is a fast solution but its insecure design disqualifies it for use in most production networks.

An alternative is the dynamic configuration of firewalls. This should satisfy the following:

1. It can be integrated smoothly into an existing administrative security framework.
2. It can be used in open source and in commercial solutions.
3. It keeps the communication between the partners open for the shortest possible time.

Currently there are different solutions to configure a firewall dynamically. These solutions are either proprietary and vendor specific such as Cisco PIX [3] or they support only certain kinds of firewalls. CODO [4] is such a solution. This document presents a novel approach to dynamic configuration of firewalls. It uses a mechanism comparable to UDP hole punching a commonly used NAT traversal technique.

3 UDP Hole Punching

NAT traversal through UDP hole punching is a method for establishing bidirectional UDP connections between Internet hosts in private networks using NAT [5]. It takes advantage of firewalls that simulate connections for UDP traffic.

Prerequisites to use UDP hole punching are the following:

- the local firewall allows outbound UDP connections
- the local firewall simulates connections for UDP data transfer as described in section 1.
- a relay server exists.

The relay server is a central part of this concept. Each client connects to the relaying server using a persistent TCP connection. Simultaneously the relay server recognizes the IP addresses of the clients. It does not even matter if any client connects to the public network through a NAT device because the public IP address is notified.

The following describes how two clients in different security domains establish a UDP connection using UDP hole punching. The initiator (client A) sends a TCP segment to the relaying server C, see figure 2 (message #1). This contains the information that client A wants to talk to client B using a UDP source port, e.g. 4711. The server notifies client B that client A has the public IP address x.x.x.x and that it expects a UDP connection on port 4711 (#2). Client B sends the preferred UDP port, e.g. 8822 to the relaying server and simultaneously it sends a UDP datagram from source port 8822 to destination port 4711 to client A (#3).

Client B's local firewall forwards the UDP datagram, creates a connection entry and the dynamic access rule which allows responses to traverse the firewall.

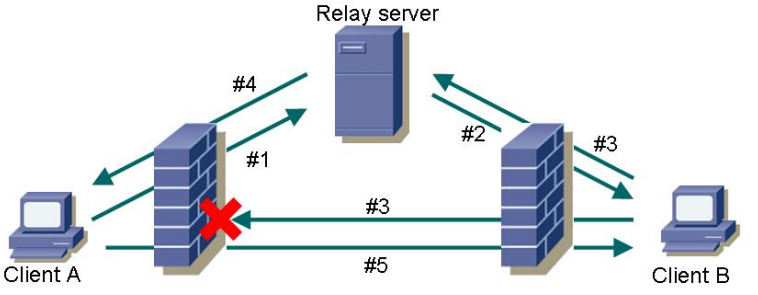

Fig. 2. UDP Hole Punching

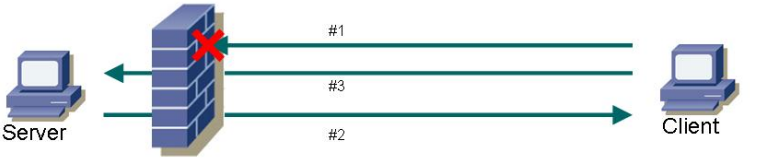

Fig. 3. UDP hole punching with netcat

Client A's local firewall rejects the packet but this does not matter at all. The relaying server C informs client A via the existing TCP connection between A and C that client B is accessible on IP address y.y.y.y and UDP port 8822 (#4).

Client A now sends a UDP datagram from source port 4711 to 8822 (#5). Client A's local firewall now creates the dynamic entries. However, the dynamic entry in B's local firewall is still active and valid, so that the UDP datagram from A to B passes the firewall. Now the communication channel is established although the static ruleset of each firewall would normally deny inbound connections according to the parameters of the protocol headers.

The concept of UDP hole punching can easily be demonstrated using *netcat*. *netcat* is a networking utility which reads and writes data across network connections. It can use TCP/IP and UDP/IP [6]. It is available on most Linux systems. The server resides behind a firewall and listens on port 4711:

server# netcat -u -l -p 4711

A client from the outside tries to connect to the UDP port 4711 on this server behind the firewall, see figure 3(#1):

client# echo Hello | netcat -p 8822 -u server 4711

This UDP connection is not allowed by the local firewall. The UDP datagram is dropped and nothing happens. Now the server sends a UDP datagramm to the client outside the firewall and punches a hole into the firewall (#2). The local firewalls allows outbound UDP communication and forwards the packet

towards the destination. Because of the algorithm used to simulate a connection the firewall passes any reply of the client to the server:

server# echo Hello | netcat -p 4711 -u client 8822

After this initial exchange the datagram from client to server is allowed to pass the firewall (#3):

client# echo Hello | netcat -p 8822 -u server 4711
server# netcat -u -l -p 4711
Hello

This simple example works on any linux system and with different firewalls. We successfully tested it with iptables [7] and Cisco PIX [3].

Because TCP uses connection states that are examined by firewalls, it is no alternative here. It is not possible to make both firewalls believe that a connection has been opened from inside.

4 A Transfer Service Using UDP Hole Punching

For a file transfer based on UDP Hole Punching, UNICORE [8] serves as a perfect environment. Many concepts described in section 3 are already realized. There is a central Gateway for all control messages which are encrypted via X.509 certificates. Users and devices are centrally authenticated and authorized once. The existing file transfers for UNICORE are all based on predefined classes which implement basic features and error handling. A new transfer service is derived from these classes and extends them with the specific methods and members for the used technique. Thus, the implementation of a file transfer using the UDP Hole Punching is done by providing the methods to exchange connection parameters like IP address and UDP port and to send the Hole Punching packet and the data using UDP.

A new challenge arises from encrypted UDP data traffic. One approach to encrypt the data could be a simple symmetric algorithm. Because the TCP connections between clients and the gateway are secured via X.509 certificates the exchange of a shared secret could be done via this channel. Moreover it would be useful to declare the encryption of the data transfer as an optional feature. Because data is not always confidential this could speed up the transfer.

The second challenge results from specific requirements of Grid applications. Data transfers should be fast and reliable but UDP is unreliable. Each UDP datagram is an instance of its own and the application has to make sure that all the data has arrived. In fact this means that Grid applications using UDP need an application layer protocol to implement reliability.

This can be accomplished by *UDP-based Data Transfer Protocol* (UDT). UDT uses UDP as transport protocol but it guarantees reliability through an upper layer. The following section describes UDT [9] in general.

4.1 UDP-Based Data Transfer Protocol (UDT)

Like TCP, UDP is a transport layer protocol. Besides the data payload UDP packets only consist of a minimal header containing information about source and destination port, packet length and a checksum. In contrast to TCP, a UDP header contains neither *flags* or *control bits* nor any sequence or acknowledgment numbers. For that reason the protocol itself is not able to read a connection state from a packet. Additionally, it cannot recognize or interpret packet loss. Therefore TCP features relating to a reliable and fair protocol such as establishing or terminating a connection, buffering packets for a resend after loss and avoiding congestion have to be implemented at higher levels.

UDT is such an implementation. UDT is not a protocol of the transport layer like TCP or UDP. It utilizes UDP as transport protocol and provides reliable communication and congestion control on the application layer, thus completely in the user space.

UDT is available as open source. It is designed and implemented by the National Center for Data Mining at the University of Illinois at Chicago. A first internet draft has been released in August 2004 [10]. The latest stable release including documentation can be downloaded from Sourceforge [9].

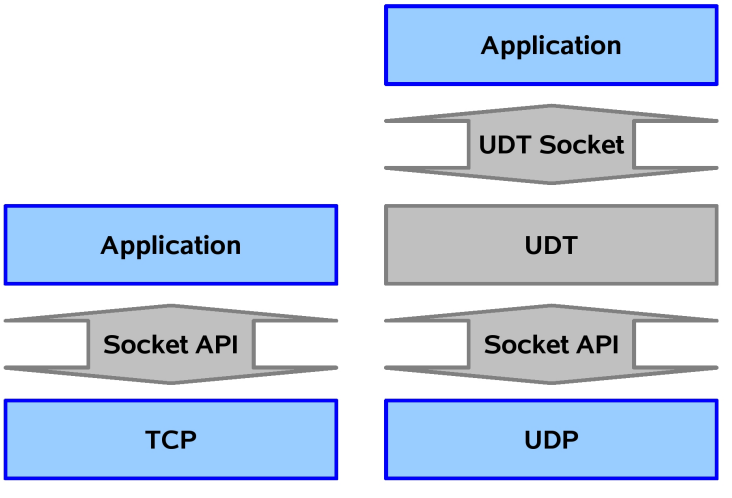

Fig. 4. TCP and UDT sockets

The UDT specific implementation for reliability and congestion control is realised as follows:

– Reliability is achieved by sequencing and acknowledgment. Each UDT packet is assigned a unique increasing sequence number. The receiver will send back acknowledgments and loss reports according to packet arrival. So lost packets will be retransmitted.

– Congestion control: unlike TCP the approach is not window but rate based, meaning that the algorithm does not open up the sender's congestion window, in fact it reduces the inter-packet delay of sent out packets, thus increasing its sending rate. Congestion avoidance uses a special case of the AIMD (Additive Increase Multiplicative Decrease) algorithm; it reduces the increase when getting close to the estimated link bandwidth.

Besides its own congestion control algorithm UDT can also utilize external or custom congestion control algorithms like *TCP Reno* or *TCP BIC* congestion control. That enables developers to find a good agreement between fairness and transfer rate, making the use of multiple parallel connections obsolete in most scenarios.

From a programmer's point of view UDT provides a *C++ API* with a semantic analogue to the TCP sockets (see figure 4).

For further examples, a tutorial and a full list of all UDT functions and references please read the UDT manual [9].

5 The Overall Design

After the concept of UDP hole punching has been described along with extensions for the usage in Grid environments in the previous sections, the overall design of an implementation as a new UNICORE transfer service can be described now.

The implementation is based on the two basic UNICORE classes "FileTransferClient" and "FileTransfer" which are extended by only one WebService method called "initUDT". After transfer objects are created by UNICORE on client- and server-side the process of transfering the file mainly consists of the following steps. The Client immediately starts preparing the UDT connection by binding its UDT socket to a randomly chosen port. It then passes its IP and port number to the server as parameters to the WebService method. The server prepares its own UDT socket and performs the UDP Hole Punching by sending an empty UDP packet to the IP and port that it just got from the client. Then it sends its own connection parameters (IP and port) back as the return value of the WebService method. The Client now can use its prepared UDT socket to connect to the server which is waiting for the connection with a UDT server socket.

During the transfer, both sides collect basic status information like the number of transfered bytes and the time needed. Status information and errors are handled accordingly to the implementation of the standard transfer classes and therefor are easily evaluated.

The fact that UNICORE and its WebServices are implemented in JAVA but UDT is implemented in C++ leads to a hybrid solution using the Java Native Interface (JNI) [11]. JNI is used by JAVA applications to involve functions that are not implemented in JAVA but in a native language. JNI can also be used to load dynamic native libraries and to involve their functions. These functions are declared but not implemented in the JAVA code. The native library is loaded

on runtime. The native functions need to follow some rules concerning the way, parameters are passed. They get pointers to standard JNI functions that give access to JAVA objects. Therefore wrapper functions are necessary to built a bridge between the JAVA classes and the UDT interface. For a more simple design and to minimize the parameter transfer between JAVA and C++, these wrapper functions also contain the code to bind the sockets and to do the Hole Punching.

When using the UDT based transfer, the native library containing the wrapper functions as well as the UDT library need to be available for the underlying hardware architecture and operating system. To simplify the installation, all native code, including the wrapper functions and the UDT source code, is compiled into one single library. The source code as well as a built script for unix like systems are provided as part of the file transfer implementation.

This transfer can be transparently used as an alternative to the commonly used ByteIO mode. Its is working in production networks and has several advantages over GridFTP and ByteIO. On one hand, it is secure enough to be used in most productive networks and on the other hand, it is faster than ByteIO and even than GridFTP. Some performance tests where done in the German X-WIN net with two machines in Juelich and Hannover, connected via 1 GBit/s ethernet. The following table shows the average throughput.

GridFTP (1 Stream)	GridFTP (4 Streams)	UDT 3	UDT 4	ByteIO
81 MBit/s	294 MBit/s	700 MBit/s	930 MBit/s	0.4 MBit/s

These values show how aggressive the standard UDT congestion control works. It must be changed to a fair algorithm or as an alternative, the throughput can be limited by quality of service rules at the networks routers.

6 Summary

Firewalls are absolute essential devices to improve the security of an organization. Consequently, this leads to restrictions regarding network connectivity and performance. Grid applications are affected by firewalls because they need high performance and low latencies. More often they use multiple connections in parallel to speed up the data transfer. To match this requirement of Grid applications static port ranges are opened on firewalls what leads potentially to unauthorized accesses to sensitive resources. Dynamic configuration can ease this problem.

This paper introduced a solution which configures a firewall dynamically based on UDP hole punching to securely establish direct transfers between hosts. The concept has been extended and adapted to the needs of Grid environments and the implementation for UNICORE has been described.

These concepts of Grid UDP hole punching can be seen as a further step in providing solutions for Grid applications dealing with existing firewalls. It

can be easily used by most of the Grid applications known today to overcome time delays until "real" dynamically configurable firewalls are available on the market.

References

1. Richard Stevens, W.: TCP/IP Illustrated I. The Protocols. Addison Wesley, Reading (1994)
2. GT4.0 GridFTP, Globus Toolkit website (August 2006), http://www.globus.org/toolkit/docs/4.0/data/gridftp
3. Cisco Security Appliance Command Line Configuration Guide - For the Cisco ASA 5500 Series and Cisco PIX 500 Series Software Version 7.2, http://www.cisco.com/en/US/docs/security/asa/asa72/configuration/guide/asacfg72.pdf
4. Son, S., Allcock, B., Livny, M.: CODO: Firewall Traversal by Cooperative On-Demand Opening. In: 14th IEEE Symposium on High Performance Distributed Computing (HPDC14), Research Triangle Park (July 2005), http://www.cs.wisc.edu/~sschang/papers/CODO-hpdc.pdf
5. Schmidt, J.: Der Lochtrick - Wie Skype & Co. Firewalls umgehen. In: CT 2006, Heft 17, p. 142. Heise Verlag (2006)
6. The GNU Netcat project (August 2006), http://netcat.sourceforge.net/
7. The netfilter.org project firewall, NAT, and packet mangling for linux (1999 - 2007), http://www.netfilter.org/
8. UNICORE Grid computing Technology UNiform Interface to COmputing REsources (August 2006), http://www.unicore.eu/
9. Gu, Y.: UDT: UDP-based data transfer library - Version 3 (May 2006), http://www.cs.uic.edu/~ygu1/
10. Gu, Y., Grossmann, R.L.: UDT: A transport protocol for data intesive applications Internet Draft, draft-gg-udt-01.txt University of Illinois at Chicago (August 2004)
11. Liang, S.: The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley, Longman, Amsterdam (1999)