FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Technical Report

Dynamic Configuration of Firewalls Using UDP Hole Punching

E. Grünter, M. Meier, R. Niederberger, F. Petri

FZJ-ZAM-IB-2006-13

August 2006

(last change: 22.08.2006)

Contents

List of Figures

List of Tables

Abstract

Firewalls separate areas of different security requirements. This major task leads to problems reagarding 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 ressources is not possible. Furthermore Grid applications often use multiple ports dynamically and in parallel. This raises the challenge of a dynamic configuration of firewalls. Current solutions are only isolated or proprietary solutions because they only address certain kinds of firewall, e.g. Netfilter and Cisco PIX. This paper describes a solution based on UDP hole punching.

Chapter 1

Firewalls and Grid applications

1.1 Firewalls and filtering of network traffic

Firewalls are used to divide areas of different security policies from each other. The major task is to prevent computing resources from unauthorized access and misuse. In order to achieve this task the firewall system processes certain information which is used as a baseline to the forwarding decision. The firewall administrator defines a ruleset which represents the implementation of the local security policy. The network traffic is divided into classes of packets which will be forwarded to the destination or 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. Stateful packet inspection engines use connection states additionally.

Firewalls store state information primarily when a TCP stream is discovered because TCP is a conection oriented protocol. Half open, established, half closed and closed states differ from each other. 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 acknowledgements discovered at the sender. Further information can be found at [St94].

The User Datagram Protocol (UDP) is a non reliable and non connection oriented transport layer protocol. The application that uses UDP has to make sure that the data is completely transmitted. Although no connections exist firewalls use a simple mechanism to feign a connection. Figure 1.1 shows a client behind a firewall sending a UDP datagram to a DNS server outside the companies network.

The client generates the UDP datagram and sends it to the firewall (#1). The firewall examines the

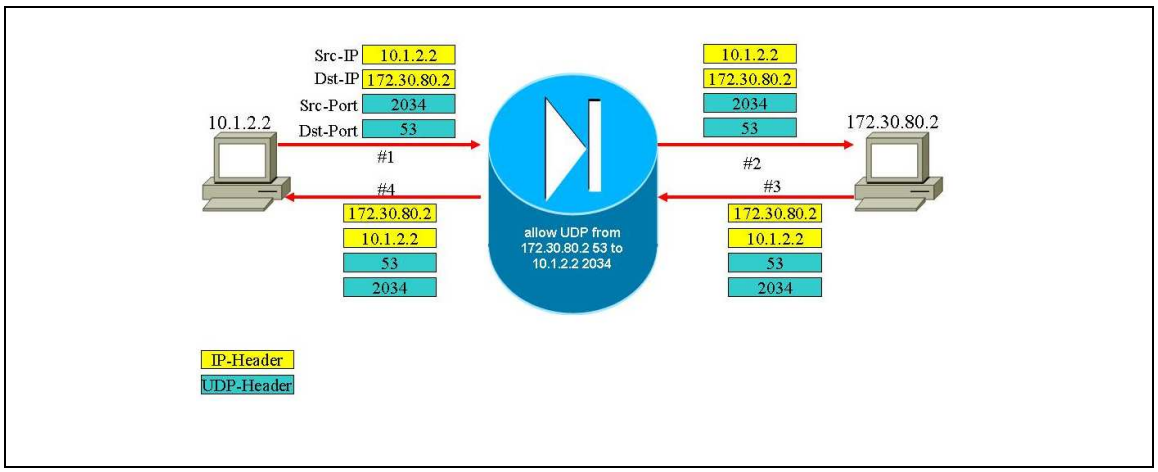

Figure 1.1: Firewalls and UDP

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 with the parameters source IP address, source port, destination IP address and destination port. However this results in a dynamic configured 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).

Any firewall currently available uses this algorithm to manage UDP connection and to forward packets that belong to established UDP communications although no connection exists at transport layer using UDP.

1.2 Grid applications and firewalls

A Grid is a distributed system which makes resources available in form of computing power, storage capacity and distributed data for its users. It forms an union of geographically distributed, independent organizations sometimes referred to as virtual organisations. 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 reliable and as fast as possible at the destination. Another trait of Grid applications is the dynamic usage of connections in parallel. GridFTP [GridFTP] can serve as an example here, which mandates that multiple parallel data connections are always established from the sender to the receiver.

Using multiple data connections demands that firewalls which are located between client and server know about these data connections. There are two ways to solve this problem. First of all the firewall can be configured statically, so that certain servers are accessible on well defined port ranges. Certainly this leads to a high number of unauthorized access probes and enforces a high level security on the server.

The second way is the dynamic configuration of firewalls and it should satisfy the following demands:

1. It can be integrated smoothly into an existing security concept.
2. It represents a concept, which can be used in open source and in commercial solutions.
3. It permits communication between the partners involved only for the minimum necessary duration.

The demands above need to be specified more detailed. The first demand guarantees that an existing security concept keeps valid. Together with the second demand cost-intensive investments for organizations are to be prevented. The investments are related to possible acquisition of hard- and/or software and on the new employment and /or further training of personnel. The best way to go would be to integrate the new solution directly into existing firewall installations. The last demand is very important, since it is clearly demanded that firewall rules are valid only if the rules are really used and only as long as they are used.

1.3 Dynamic configuration of firewals

Currently there are different solutions to configure a firewall dynamically. One solution is *Cooperative on demand opening* (CODO) [Sal05]. It supports interaction with iptables firewalls and works

with TCP and UDP. Currently CODO was not designed to fit all the particular requirements of a Grid environment [Val06].

Other solutions are proprietary ones. They are offered by firewall vendors. In general vendors offer some kind of configuration commands which identify a connection that has to be established before well defined related traffic is allowed to travers the firewall. The following configuration example is given for Cisco Secure PIX Firewall Version 7.2. It allows connections from any source IP address to host 10.1.1.1 on destination port 2811 and allows back connections from TCP ports 20000 to 25000 if a connection on TCP port 2811 is established, see [CP7].

access-list permit tcp any host 10.1.1.1 eq 2811
established tcp 0 2811 permitfrom tcp 20000-25000

Although this seems to be some kind of dynamic configuration there are restrictions to this command: It allows return access to outbound connections only. This means that inbound connections to a server cannot be handled by this command. Moreover the *establish* command does not work together with port address translation (PAT) which is often used.

This document presents another approach of dynamic configuration of firewalls. It uses a mechanism comparable to UDP hole punching. In computing UDP hole punching refers to a commonly used NAT traversal technique. NAT traversal through UDP hole punching is a method for establishing bidirectional UDP connections between Internet hosts in private networks using NAT [Sch06].

Chapter 2

UDP hole punching

2.1 UDP Hole Punching

Prerequisite to use UDP hole punching is, that

- the local firewall allows outbound UDP connections
- the local firewall handles UDP connections as streams described in section 1.1.
- a relaying server exists.

The relaying server is a central part of this concept. Each client connects to the relaying server using a persistent TCP connection. Simultaneously the relaying server gets 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.

If the clients want to talk to each other they use a UDP connection. The initiator sends a TCP segment to the relaying server C, see figure 2.1 (#1). It indicates 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 travers the firewall. 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

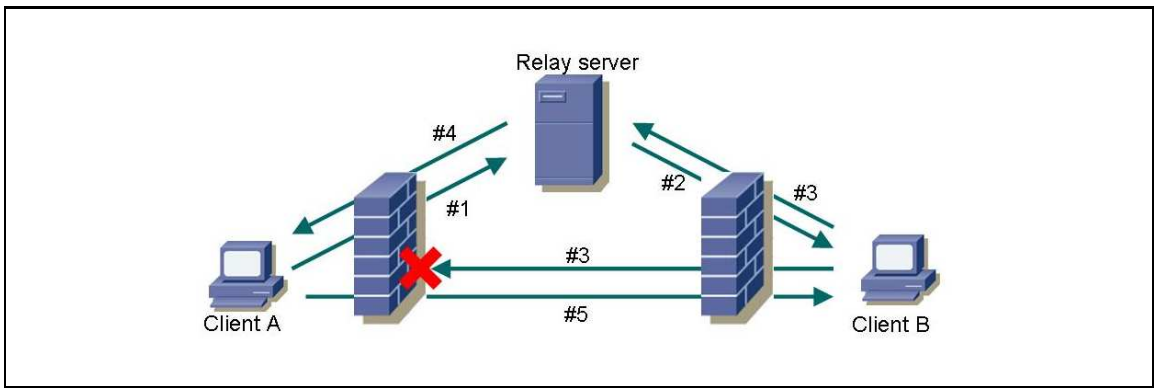

Figure 2.1: UDP Hole Punching

is established although the static ruleset of each firewall would normally deny inbound connections according to the parameters of the protocol headers.

Figure 2.2: UDP hole punching with netcat

The concept of UDP hole punching can easily be shown using netcat. netcat is a networking utility which reads and writes data across network connections, using the TCP/IP protocol [Nc06]. It is available on any common linux system. The server resides behind a firewall and listens on port 4711:

server# netcat -u -l -p 4711

Now a client from outside tries to connect to the UDP port 4711 on this server behind the firewall, see figure 2.2(#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):

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

After that 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 common linux system. It can be tested with different firewalls. In our tests it worked with iptables and Cisco PIX.

Chapter 3

UDP hole punching in Grid environments

3.1 UDP Hole Punching in Grid environments

The concept of UDP hole punching can be easily modified to be used in Grid environments. The relaying server is changed to a relaying service. It listens on a TCP connection and waits for user connections. At connect time user and server that hosts the relaying service may authenticate each other using X.509 certificates. If the TCP connection between the participants has been established data transfers can take place as described in section 2.1.

The advantage of a relaying service is obvious. The service resides at the server host. According to figure 2.1 in section 2.1 relay server C and client B become the same host. Mutual authentication between client and server is necessary only once at connect time. If the TCP connection is established the data transfers are allowed to start. Evenmore there are no problems resulting from NAT. Any server accessible from the public network is exempted from NAT algorithm and the server only sees the public IP address of the client.

Often it is useful to transfer data using multiple parallel connections. The client has to indicate how many connections should be used in parallel. It allocates the sockets and sends the port numbers to the server. The server reads the information and tries to initiate the maximum number of UDP connections available by sending small UDP datagrams to the client. This may depend on the system load. The server sends the local UDP port numbers to the client and the client starts the data transfers.

A new challenge raises regarding the encryption of UDP data traffic. Secure Socket Layer only works with TCP as transport protocol. Since UDP is the transport layer protocol in this concept we need a way to exchange data securely. The applicable algorithm is Diffie-Hellman Key Exchange [Sch96]. The Diffie-Hellman algorithm cannot be used to encrypt or decrypt data but it is used for key distribution. Server and client are able to derive a key from the chosen Diffie-Hellman parameters. This key could be used to encrypt the data. 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.

Another approach to encrypt the transferred data could be a simple symmetric algorithm. Because the TCP connection between client and server is secured via X.509 certificates the exchange of a shared secret could be done via this channel.

The second challenge raises regarding the requirements of Grid applications. Data transfers should be fast and reliable. Because UDP has no three-way-handshake and no acknowledgements it is faster than TCP, but UDP is not reliable. 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

have to be changed. They have to use UDP and need an instance to implement reliability.

An alternativ to UDP is the *UDP-based Data Transfer Protocol* (UDT). UDT uses UDP as transport protocol but it guarantees reliability in upper layer headers. Of course Grid applications have to be modified also, but the effort is very small because an API using UDT can be provided. Only signatures of subroutines allocating sockets have to be changed. The following section describes UDT in general.

3.2 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, in a UDP header there are 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 tearing down 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 application layer, thus completely in user space.

There have been several earlier approaches to this concept, like *RBUDP* or *TSUNAMI*, but currently UDT appears to be the most actively maintained project.

UDT is open source and distributed under the *LGPL*. 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 [Udt04]. The latest stable release including documentation can be downloaded from Sourceforge [Udt06].

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

- Reliability is done 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 bandwith.

Besides its own congestion control algorithm UDT can also utilize external or custom congestion control algorithms like *TCP Reno* or *TCP BIC* congestion control.

From a programmer's point of view UDT provides a *C++ API* with a semantic analogue to the TCP sockets (see figure 3.2).

- In existing applications almost all TCP socket calls can be replaced 1:1 with socket calls from the UDT namespace. To use UDT instead of TCP insert the line *#include <udt.h>* in your source code, replace all socket calls *xxx()* with *UDT::xxx()*, recompile and link the udt-library.
- For new applications: Just use *udt.h* instead of *socket.h*. For example a typical client server sequence would then look like:

  For the client:

  #include <udt.h>

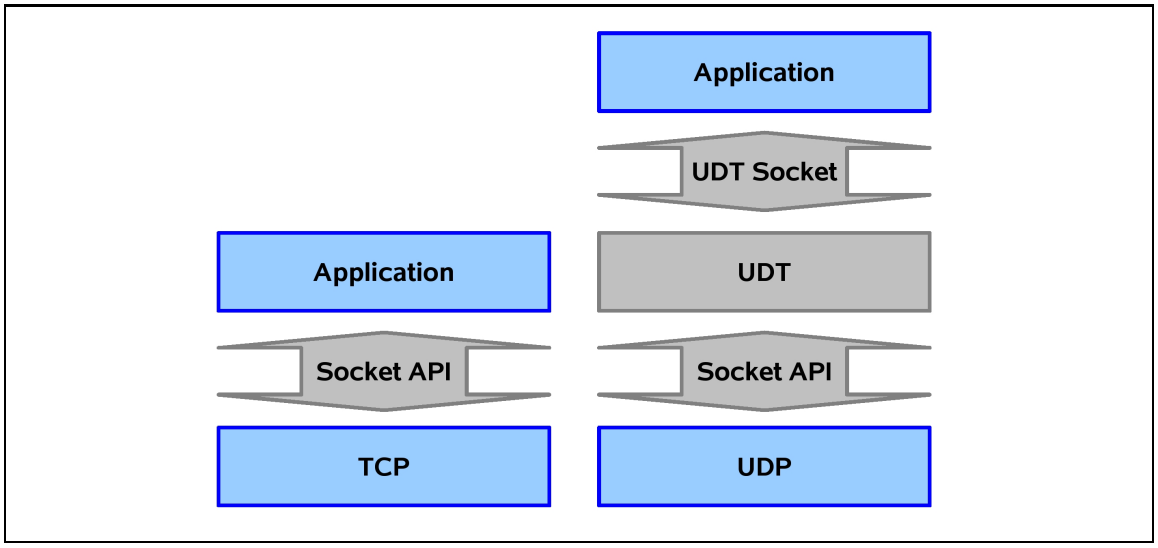

Figure 3.1: TCP and UDT sockets

UDT::socket()
UDT::connect()
UDT::recv() or *UDT::send()*

For the server:

#include <udt.h>

UDT::socket()
UDT::bind()
UDT::listen()
UDT::accept()

Custom congestion control is provided by a *class CCC*. The application can use *UDT::setsockopt()* or *UDT::getsockopt()* to assign this control class to a UDT instance, and/or set its parameters. Example:

UDT::setsockopt(usock, 0, UDT_CC, new CCCFactory<CTCP>, sizeof(CCCFactory<CTCP>))

The above code assigns the *CTCP* control algorithm to a UDT socket *usock*, meaning that this instance would use (and behave like) *TCP Reno* congestion control *(NB: utilizing UDP underneath)*.

For further examples, a tutorial and a full list of all UDT functions and references please read the UDT manual [Udt06].

3.3 The overall design

After the concept of UDP hole punching has been described and modified for the usage in Grid environments in the previous sections, the overall design of the concept will be explained shortly.

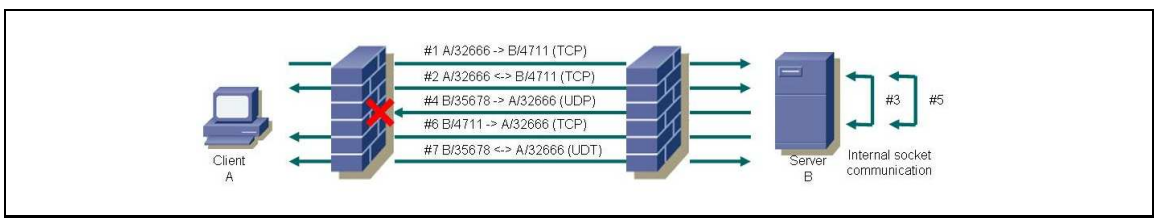

Figure 3.2: UDP hole punching in Grid environments

The Grid client application at host A connects to the server host B at a predefined port number e.g. 4711 via A's local firewall and the remote firewall at B's location (#1). The specified port

has to be opened at any firewall for every host which provides this relaying service. After mutual authentication using X.509 certificates has been done successfully, data exchange using dynamic ports can take place.

Client A and Server B exchange securely authentication and authorization information as well as key information for a later secure comunication with the required service on this host (any grid application, AGA) (#2).

The relaying service informs the AGA service on host B locally that host A wants to connect to servce AGA via e.g. UDP port 32666 (#3). AGA looks for a free port number e.g 35678. Then AGA sends a UDP packet to host A at port 32666 and uses as source port 35678 (#4). The packet traverses host B's firewallbecause outgoing UDP packets are allowed, but gets rejected at host A's firewall. Nevertheless host B's firewall assumes a "UDP streamb̈etween A/32666 and B/35678.

After the first UDP datagram has sent from host B to host A, host B's AGA now informs host B's relaying service about the port to be used as destination by A (#5). The relaying service informs host A via the initial open TCP connection that AGA is waiting for UDP communication at port 35678 (#6).

Now host A connects to service AGA on host B with the agreed UDP port 35678 and uses as source port 32666 (#7). Hosts A and B are now able to use the UDP-based Data Transfer protocol (UDT) via the established communication path (also #7). The firewall has been opened securely and dynamically without any firewall modifications.

3.4 Summary

Firewalls are absolute essential devices to improve the local security of an organisation. Although their necessity is proved they lead to problems 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 fit this requirement of Grid applications static port ranges are configured on firewalls. Currently this is the only solution but it leads to unauthorized accesses to sensitive resources. Dynamic configuration would ease this problem.

This paper introduced a possible solution which configures a firewall dynamically based on UDP hole punching. The concept is modified and adjusted to the needs of Grid environments.

An application programming interface has to be created which realizes the concept introduced here. Probably this solution can be easily integrated in current Grid applications. The well known and widely distributed middleware UNICORE as an example could be improved to use multiple data connections.

The described concept 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" dynamic configurable firewalls are available on the market.

Bibliography

[CP7] Cisco System, Inc.
Cisco Security Appliance Command Reference
For the Cisco ASA 5500 Series and Cisco PIX 500 Series, Software Version 7.2.(1), Text Part Number: OL-10086-01
Cisco Systems 2006

[GridFTP] GT4.0 GridFTP, Globus Toolkit website
http://www.globus.org/toolkit/docs/4.0/data/gridftp, August 2006

[Nc06] The GNU Netcat project
http://netcat.sourceforge.net/, Augusat 2006

[RFC959] RFC 959
File Transfer Protocol
http://www.ietf.org/rfc/rfc959.txt?number=959

[Sal05] S. Son, B. Allcock, M. Livny
CODO: Firewall Traversal by Cooperative On-Demand Opening
14th IEEE Symposium on High Performance Distributed Computing (HPDC14), Research Triangle Park, July 2005
http://www.cs.wisc.edu/ sschang/papers/CODO-hpdc.pdf

[Sch96] B. Schneier
Applied Cryptography Second Edition: protocols, algorithms, and cource code in C
Wiley, 1996

[Sch06] J. Schmidt
Der Lochtrick - Wie Skype & Co. Firewalls umgehen
Heise Verlag, C'T 2006, Heft 17, pp. 142 ff

[St94] W. Richard Stevens
TCP/IP Illustrated I. The Protocols.
Addison Wesley 1994

[Udt04] Y. Gu, R.L. Grossmann
UDT: A transport protocol for data intesive applications
Internet Draft, draft-gg-udt-01.txt
University of Illinois at Chicago, August 2004

[Udt06] Y. Gu
UDT: UDP-based data transfer library - Version 3
http://www.cs.uic.edu/∼ygu1/, May 2006

[Val06] G.L Volpato, Ch. Grimm
Practical Tests and Experiences with CODO
D-Grid project work paper, July 2006