

Distributed Information Systems Laboratory

Implementing a NAT and Firewall traversal library

Author:
Damien AUROUX

Supervisors:
Prof. Karl ABERER
Nicolas BONVIN

January 10, 2009

Contents

Introduction	1
1 Background	2
1.1 Motivation	2
1.2 Existing NAT Traversal Techniques	3
1.3 NATaWare	4
1.4 Goals and Requirements	5
2 Design and Implementation	6
2.1 Architecture of the Project	6
2.2 Socket Service	7
2.3 Directory Service	8
2.4 Rendez-vous Service (1)	8
2.5 NAT Discovery Service	9
2.6 Rendez-vous Service (2) and Tunnels	11
2.7 Connection Service	12
2.8 Chat Application	13
2.9 Summary	14
3 Evaluation	15
3.1 Test Setup	15
3.2 Test Run	15

3.3	Qualitative Test Results	17
3.4	Possible Improvements	18
	Conclusion	20
A	Installation Guidelines	22
A.1	Requirements	22
A.2	Using the Framework	22
A.3	Example : Chat Application	23

Introduction

Network Address Translation (NAT) [1] is more and more widely used as a means to connect local area networks to the Internet. While this is a cheap and efficient way to share an Internet connection between multiple users, many network applications are unable to work properly through these devices without complex configuration.

However, there are some techniques that can be used by an application in order to get through NAT devices and firewalls without any manual configuration. The goal of this project is to implement them into a simple library that software developers can easily use for developing network applications.

This report first tackles the background of this project by explaining the reasons why this work is useful and analyzing previous related work, before giving an overview of the project, its goals and requirements.

The second part deals with the implementation of the NAT traversal techniques: it shows the general architecture of the project, and then focuses on the different components of the framework and the chat application developed as proof of concept.

Finally an evaluation of the project is done and qualitative results are determined from practical experiments, and the conclusion is an opportunity to think about possible improvements.

1 Background

1.1 Motivation

Nowadays, firewalls and NATs are widely used in corporate network architectures, and become more and more common in home networks. Firewalls block a part of the network traffic, based on rules; they are used for security reasons, in order to prevent attackers from gaining access to the internal network from the Internet, but sometimes also in order to limit the Internet capabilities of internal users and software. NATs are used in response to the penury of public IPv4 addresses: they enable a large number of private IP addresses to simultaneously use one single public Internet IP address, by translating dynamically the IP headers of every network packet.

Firewalls and NATs are very useful; however they have a side effect that is sometimes not desirable: they seriously limit the network capabilities of software in the local network, unless some specific rules are added to take it into account. This is the case in particular for softwares that have to act as servers: a firewall simply blocks non-authorized incoming traffic, while a NAT does not know to which internal IP address the traffic should be routed. These applications are more and more common: they include peer-to-peer applications (e.g.: BitTorrent), VoIP applications (e.g.: Skype, TeamSpeak, SIP phones), VPNs (e.g.: Hamachi).

In order for these applications to work properly, we would normally need to add exception rules in firewalls and NATs configurations in order to allow inbound traffic for these applications. However these configurations are not accessible by a standard user in a company network; in a home network, a software firewall is easily configurable for such exceptions, but adding a port forwarding rule in the NAT configuration is often too complex for a lambda user.

For these reasons, it is necessary to implement technology for firewall or NAT traversal in these applications. Because all NATs and firewalls do not behave the same way, it is usually necessary to implement many techniques together in order to achieve a good efficiency. This is very time-consuming for the application developer, who usually does not like to spend much time on features that do not concern his application directly. The goal of this project is to spare him that part, i.e. implement firewall and NAT traversal techniques in an open-source library in Java that could easily be used by any

network application.

1.2 Existing NAT Traversal Techniques

NAT traversal techniques are well known and well documented, and they can be divided into two categories. Some use functionalities of the NAT device so that it willingly lets the application get through :

- *Port forwarding*: this simply consists in manually configuring the NAT device and adding a rule that forwards incoming traffic on a specific port to a specific local address. This requires an access to the device configuration, and an application using a specific unmodifiable port would only work on one host.
- *Universal Plug 'n Play* (UPnP) [2]: UPnP-enabled NAT devices provide a programmatic interface to applications so that they can dynamically control the behaviour of the device and route incoming traffic to the correct destination. This is probably the best way to perform NAT traversal, however the UPnP functionality is not available on most NAT devices because of security issues in the past.

Since these techniques are authorized by NAT devices, they need the user or application to be able to access it, and they work differently from one device to another. This is why we focus on the other techniques, less reliable but without any interaction with the NAT device:

- *Hole punching*: this method tries to take advantage of properties existing in many NAT implementations to overcome our problem. The idea is to have both endpoints try to initiate a connection to each other at the same time so that both NAT devices identify incoming traffic as part of the outgoing connection and forward it accordingly. UDP hole punching is easy to implement, TCP hole punching is much more complicated because of the stateful aspect of TCP and does not always work (in particular hole punching does not usually work with symmetric NATs).
- *Relay*: if direct connection cannot be established between two NATed hosts, a publicly available host can act as a relay between them. This method always works but communicating through a relay is costly, so it should only be used as a last resort.

Other techniques exist, but most of them are built on the same basic principles. However it must be emphasized that these are only the techniques for establishing a connection between two NATed hosts. In order to apply them, one must first discover if it lies behind a NAT and what its public IP address is, learn who else is using the application and how to contact them, and register to a relay at least for knowing when and how someone tries to connect to it. Establishing the connection is just one part of the job, and getting everything ready for two hosts to communicate represents much more work. STUN (Simple Traversal of UDP through NAT) [3] and TURN (Traversal Using Relay NAT) [4] are two protocols that cover all these aspects, however the first one only works for UDP while the second one only uses costly relays.

A few frameworks that combine multiple techniques have been developed in the past few years, such as NATTrav [5], STUNT (Simple Traversal of UDP through NAT and TCP too) [6] or JXTA [7]. But each of them has either performance, reliability or simplicity issues, and for that reason they are not widely used by software developers, hence the need for a performant, reliable, easy-to-use and open-source framework.

1.3 NATaWare

NATaWare [8] is a Java-based framework for NAT traversal. This open-source project created by Daniel Eichhorn during his Master Thesis [9] in 2006 was born from this idea to develop a simple and yet performant framework for NAT traversal, which would be available for any Java network application developer to use. A first working version of NATaWare was released at the end of his Master Thesis. It was working well, but because of some unstable used libraries and a few perfectible design choices, it was not as efficient and simple as necessary for a wide use by developers.

So it was decided to restart the implementation of NATaWare from scratch, this time using updated and more efficient communication libraries, after well-considered design choices taking advantage of the experience from the first version. The current work is the result of that new implementation of NATaWare. The design choices and the general architecture were decided mainly by Daniel while I did most of the implementation under his supervision.

1.4 Goals and Requirements

The goals and requirements for this project are the following :

- Implement a Java-based framework for NAT traversal with a universal architecture. It should be possible for developers to use the framework in their network applications to overcome the NAT device issue.
- Implement a simple chat application on top of the framework as a proof of concept.
- *No configuration*: the framework should work whether the host is behind a NAT or not, without the need for the final user to configure his NAT device or even know anything about the topology of his network.
- *Simple API* for the developer: the framework will not be widely used by developers unless it is very simple to use and behaves like a black box. The developer of a network application should only need to add a few generic lines of code to his program to implement NAT traversal.
- *Performance*: the framework must succeed in establishing a connection towards a NATed host, and it must do so in short time.
- *Scalability*: the implementation of the framework should be scalable and therefore avoid single points of failure and traffic bottlenecks.

2 Design and Implementation

2.1 Architecture of the Project

This project was implemented using Java 1.5 [10]. Instead of using the raw Java NIO (New Input/Output) package for network communications, we used Apache MINA 2.0 [11], a very powerful network library built on top of Java NIO, which offers asynchronous handling of messages among other useful features. We also considered using a Representational State Transfer (REST) [12] approach for simplifying access to other hosts' services, but we were already in the implementation phase at that time and adopting it would have required too much additional work.

The framework is split into 5 components, all of them running concurrently and performing their own tasks. Most of them have two or more different possible behaviours, depending on whether they are acting as a client or a server at a given time, or whether the host is NATed or not. The components are the following:

- The *Socket Service* is the low-level base for all other services. Its task is to listen for all incoming connections and forward messages to the appropriate handlers.
- The *Rendez-vous Service* has two roles: first, it keeps a list of available public nodes and updates it periodically; second, it enables a NATed node to have rendez-vous nodes. A public node P being a rendez-vous for a NATed node N means that N maintains an active connection to P and P acts as a front for N with regard to the rest of the nodes, so that N can receive messages from other nodes through P .
- The *NAT Discovery Service* is responsible for finding out whether a host is behind a NAT or not, what its public IP address is and whether its Socket Service is reachable from outside or not.
- The *Directory Service* aims at keeping up-to-date contact information about all nodes using the framework; in particular it stores the address of the Socket Service for a public node, and the addresses of its rendez-vous for a NATed node. This component only runs on a few predetermined public nodes, which is why it does not appear on Figure 1.

- The *Connection Service* is the highest level component, it uses information from all other services to establish a connection between two hosts and hand it over to the application.

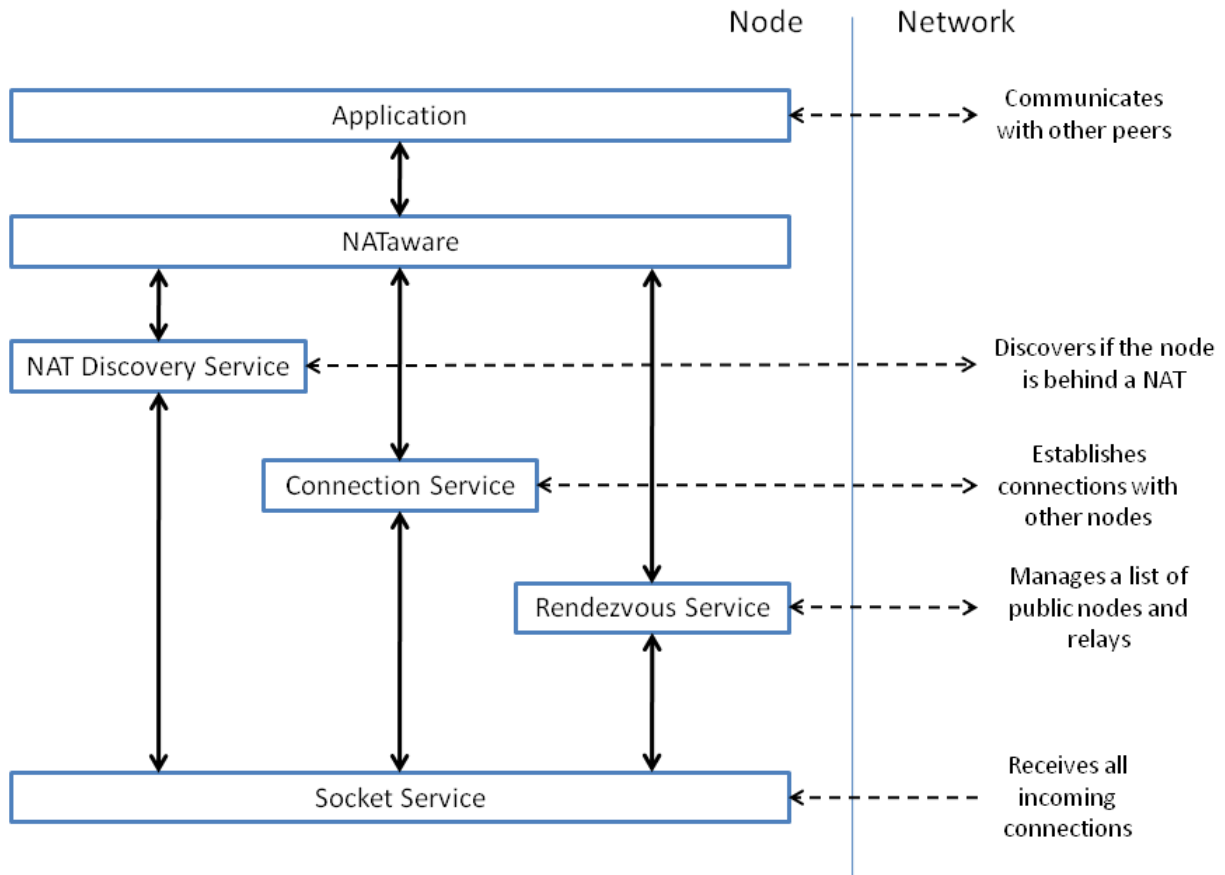


Figure 1: General architecture of the framework

2.2 Socket Service

The Socket Service is the first component to be launched when the framework runs. A port number is chosen by the application (or a default one is selected) and the Socket Service listens to incoming connections on this port. Later, when other services are launched, each of them registers its message handler and message type to the Socket Service. When a message is received, it is forwarded according to its type to the message handler of the appropriate service. The main advantage of using a single demultiplexing

listener for all incoming connections is that only one port is used for reception and shared between all services.

2.3 Directory Service

The Directory Service is started immediately after the Socket Service, but only on one or a few predetermined nodes. Every node that starts running the framework gives contact information about itself to the directory, and later sends an update whenever it may be useful. That information includes the identifier of the node, its network topology (public or NATed) and the address(es) where it can be reached, i.e. its public address if it is publicly reachable, or else the address(es) of its rendez-vous. Any node that is registered to the directory can ask for the list of all node identifiers (i.e. the name of every host that runs the same application) or detailed information about one target node (i.e. a way to establish a connection with it).

In order for the framework to work properly, at least one node must run the directory service before any other node can join the network. The directory must run on a publicly available node and its address must be initially known by all other nodes (hard-coded in the application source code or in a configuration file for example).

2.4 Rendez-vous Service (1)

The next component to start is the Rendez-vous Service. Its role is to keep an up-to-date list of addresses of public nodes that run the framework. Directories run this component like every other node, but they remain passive and only deal with incoming requests. The other nodes have both a passive server part (processing incoming requests and sending the answers) and an active client part (sending requests to other nodes and waiting for the answers) that run simultaneously.

The client periodically chooses one random public node among the list and sends a request to it; the request contains the list of public addresses. The server receives it, updates its own list if necessary and answers with the list of additional addresses it may have. The client receives these missing addresses and updates its own list. This way, both clients and servers receive periodic updates. The Rendez-vous Service keeps running and updating all the time, so that it can quickly learn about new nodes and delete dead nodes from the list. The goal is for every node to have a list of available

public nodes that may become rendez-vous if needed.

Rendez-vous Service

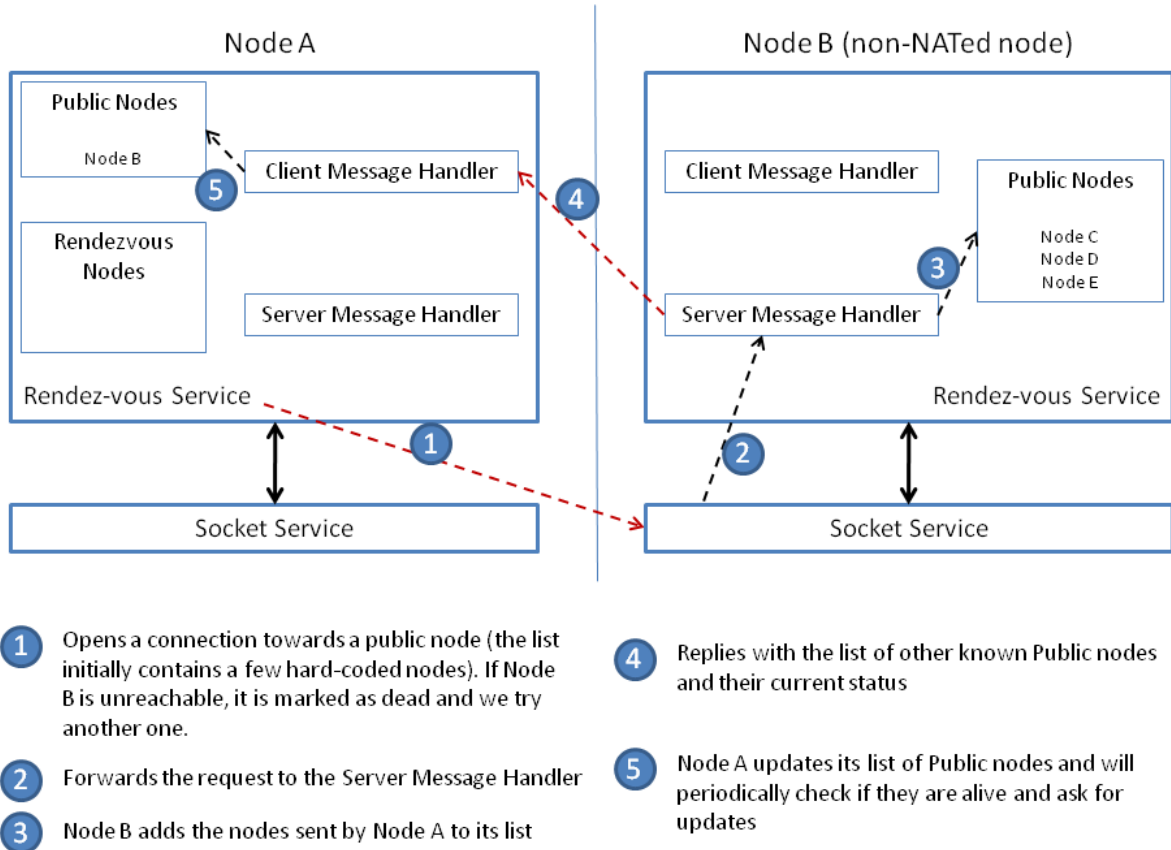


Figure 2: Rendez-vous Service - Public nodes exchange

2.5 NAT Discovery Service

Now that every node has a list of available public nodes, they can help each other to perform NAT Discovery. This service also has a passive server part (processing NAT discovery requests and sending back NAT discovery results) and an active client part (asking a public node for help to discover whether the current node is NATed or not). Directories do not run the client part since being a non-NATed node is a prerequisite for becoming a directory.

The client that wants to know whether it is NATed or public sends a NAT discovery

NAT Discovery Service

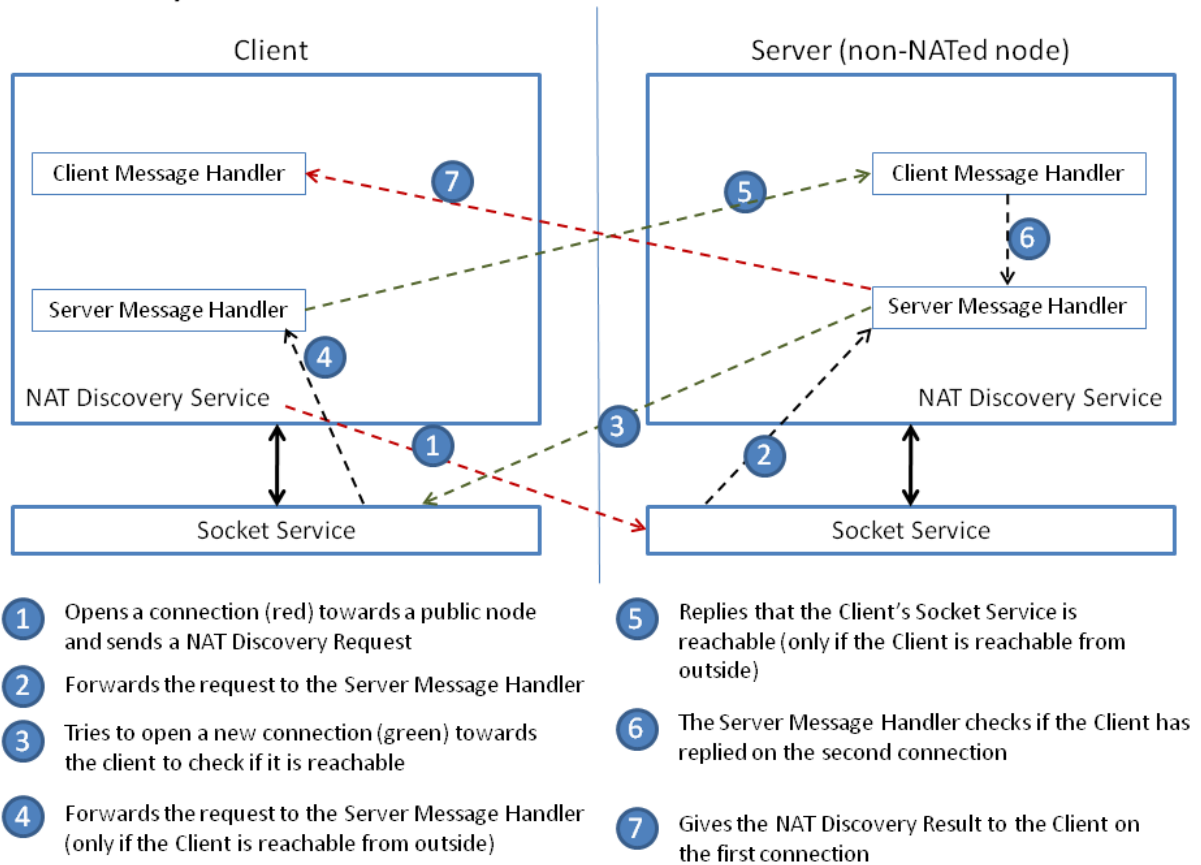


Figure 3: NAT Discovery Service

request to a public server, containing the port number of the client's socket service. The server obtains the public IP address of the client as being the source of the request. It opens a new connection to this public IP address and the given port number and waits for an acknowledgement in order to find out whether the client's service socket is publicly reachable or not. Then it answers back to the client on the first (still open) connection, sending the public IP address and whether the service socket is reachable or not. The client gets this result, compares the received public IP address to its known private IP address and concludes about the presence of a NAT device. This information is then sent to the directory for update.

2.6 Rendez-vous Service (2) and Tunnels

Once a node knows whether it is NATed or not, some adjustments need to be done to the behaviour of the Rendez-vous Service. If the node is publicly available, then the only required modification is to add, from now on, oneself's public address to the periodically exchanged lists of known public nodes.

Rendez-vous Service (2)

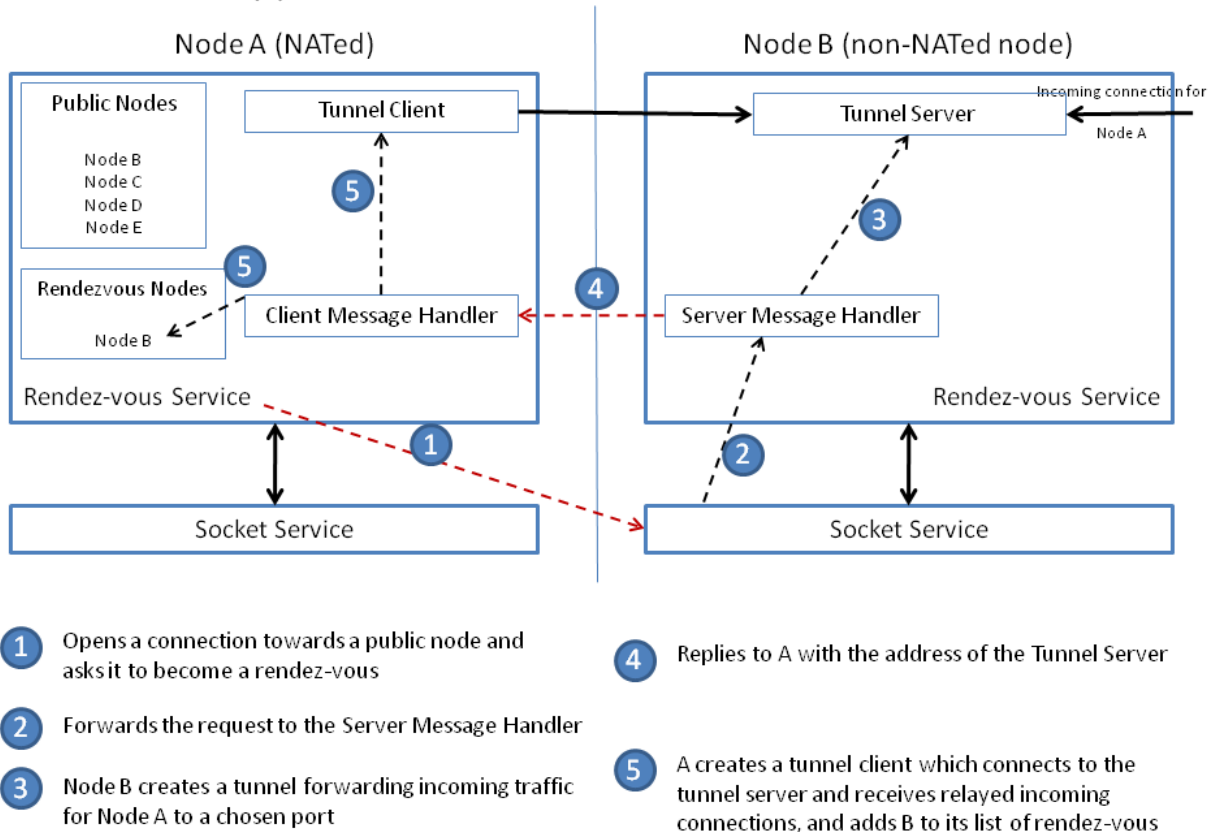


Figure 4: Rendez-vous selection and tunnel creation

If the node is not publicly reachable on the other hand, it needs to find rendez-vous in order to become reachable. A new client thread is started in the Rendez-vous Service in addition to the others, whose task is to maintain a few active rendez-vous nodes all the time. It chooses one of the least occupied alive public nodes (their workload is obtained together with their addresses) and asks it to become a rendez-vous. The targeted rendez-vous candidate may accept, in that case it creates the server side of the

tunnel. Then the client creates its own side of the tunnel and connects it to the server; the tunnel is now established, and any node may communicate with the NATed node through the tunnel by connecting to the rendez-vous. The directory is then advertised about this new rendez-vous, and the client periodically checks if the tunnel is alive.

The default number of rendez-vous a NATed node tries to keep at any time is 2, so that it would still be reachable if one fails. Directories may become rendez-vous for a NATed node like any other public node, however they have a more crucial role so their workload is initially set to a very high number, so that they will only be chosen for rendez-vous if no other public node is available.

The detailed functioning of a tunnel is not important, but what needs to be underlined is that once the tunnel is created, its behaviour is transparent for both nodes: on the rendez-vous node, incoming traffic to be relayed is not going through the Socket Service and is directly forwarded without looking at the contents; on the client node, tunneled traffic is forwarded to the Socket Service and then processed like any direct communication. This way, tunneled communications do not need to be processed in their own way and are more reliable, and rendez-vous nodes get limited additional workload.

2.7 Connection Service

Now everything is ready to process a connection request from the application. When the application asks the framework to connect to a specific node identifier, the steps are the following: first we ask the directory for an address corresponding to this identifier; we receive either a public address for direct communication if the node is reachable, or the address of one or more rendez-vous for a tunneled communication. If there is a public direct address, we simply connect to it, there is no need for NAT traversal. Otherwise we connect to the rendez-vous and send a message through the tunnel, so that the destination node knows we are trying to reach it. If the source node is not NATed itself, then the informed destination node can create a direct reverse connection towards the source node. If both nodes are NATed, the tunnel is used for communication. Anyway, the best connection that is obtained at the end of the connection process is handed to the application for its personal use.

NAT Traversal : Node A wants to establish a connection with Node B

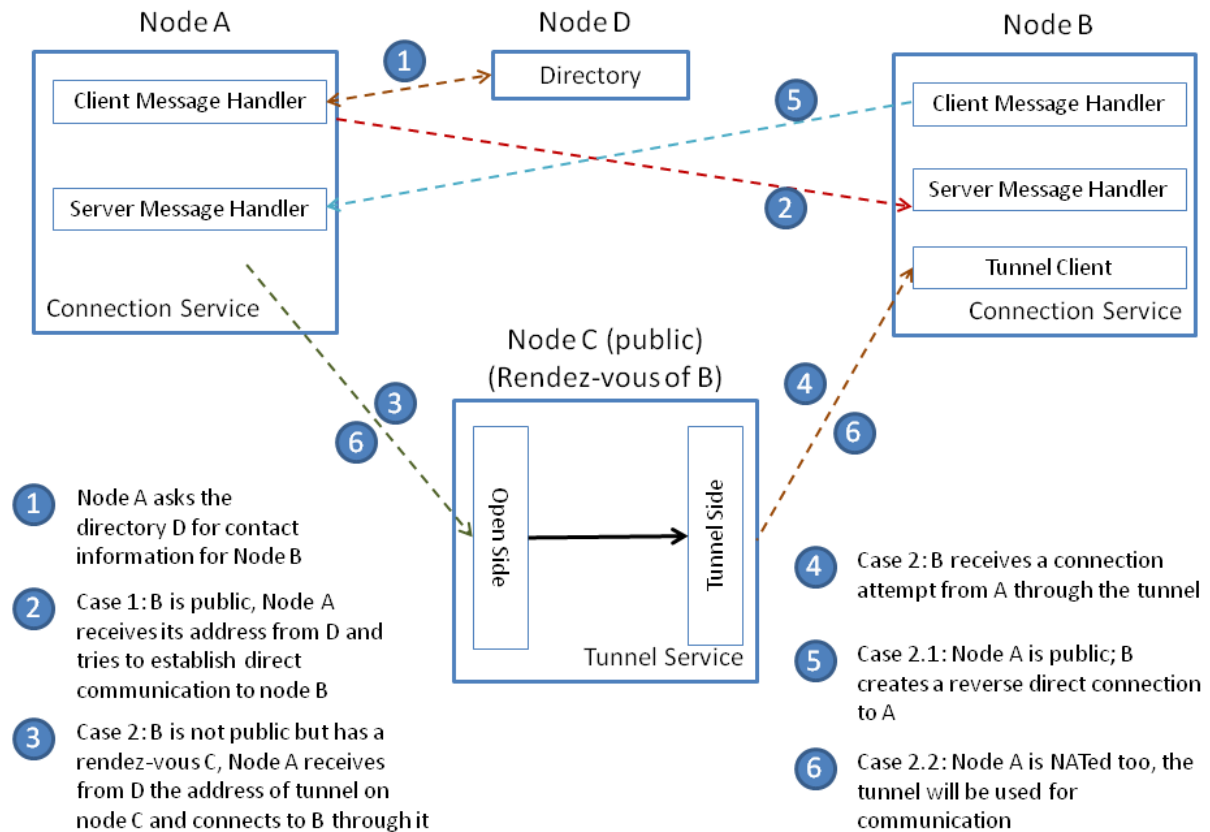


Figure 5: Connection Service

2.8 Chat Application

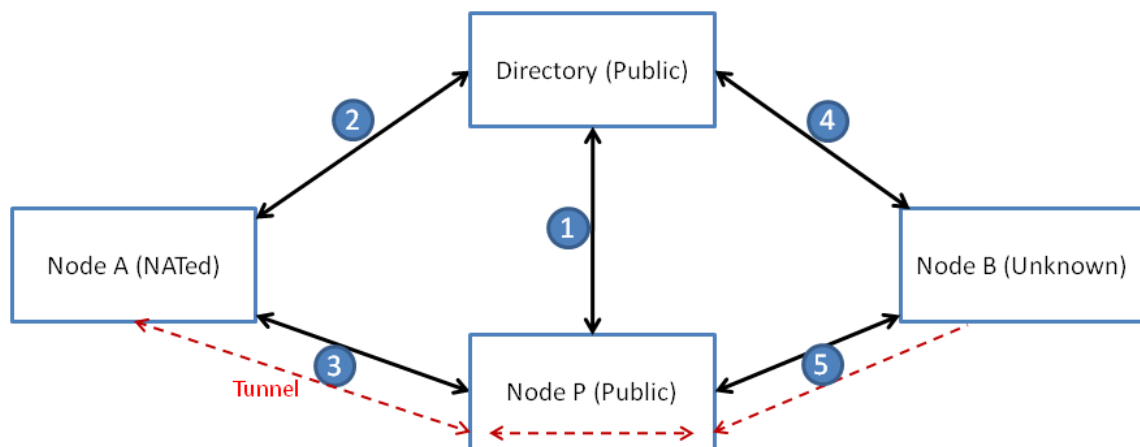
The Chat Application is not part of the NAT traversal framework. It is a very simple network application built on top of it whose sole purpose is to demonstrate the framework's capabilities.

Three classes make up the application: one of them is only a container for text messages; the second one is a message handler that displays received messages; the main one contains two methods, one for the directory and one for the other nodes. Once the application is started, the user is prompted for an identifier and a port number of his choice. After that, the NAT traversal framework is initialized, service after service; when a node is ready, the list of participants is displayed and the user is invited to

choose one. A connection to this node is established by the framework, through NATs if necessary, and the user can then send messages to the chosen participant.

2.9 Summary

Figure 6 summarizes the sequence of relevant events during a typical use of the framework :



- 1 Node P registers to the directory and will periodically exchange lists of public nodes with the directory and other public nodes
- 2 Node A registers to the directory and learns about node P
- 3 Node A asks Node P for public nodes; Node A asks Node P for NAT Discovery and learns that it is NATed; Node A asks Node P to become a rendezvous, Nodes A and P establish a tunnel for incoming connections for Node A
- 4 Node B registers to the directory and learns about node P; node B asks the directory for node A, the directory gives the address of the tunnel
- 5 Node B establishes a connection towards A via P; if possible, Nodes A and B try to establish a direct communication

Figure 6: Sequence of events when using the framework

3 Evaluation

3.1 Test Setup

The test platform consists of four nodes: one is the directory, another is a standard public node and the last two are NATed nodes. Since we only had two computers at our disposal for the tests, these two were running the directory and the public node, while each NATed node was a virtual machine behind a NAT software on one of the two computers, unreachable from the other computer.

- The directory node is running on Windows Vista with the IP address 10.0.0.1 on port 7878. This host is behaving like a publicly reachable node and has to be started first. The directory can only receive messages, it is not designed to send messages.
- The public node is running on Windows Vista with the IP address 10.0.0.2 on port 7878. This host is publicly reachable.
- NATed node 1 is running on Windows XP with the private IP address 192.168.253.128 on port 4242. Its public IP address is 10.0.0.2 but an incoming connection to 10.0.0.2:4242 fails, the Socket Service is not reachable.
- NATed node 2 is running on Fedora 9 Linux with the private IP address 192.168.253.128 on port 4242. Its public IP address is 10.0.0.1 but an incoming connection to 10.0.0.1:4242 fails, the Socket Service is not reachable.

3.2 Test Run

This section describes the execution of the test, step by step:

1. The directory is started; services are initialized and the node is available for processing requests.
2. The public node is started; it registers itself to the directory, then initializes the services. The Rendez-vous service begins to ask periodically the only node it

knows (i.e. the directory) for public addresses, and the NAT Discovery service performs NAT Discovery with the help of the directory too. Once the result of the NAT discovery comes back and the node knows it is public, the directory is updated and the node is fully operational. It gets the list of available nodes (i.e. the directory and itself) and waits for the user to choose an action.

3. NATed node 1 is started; it registers itself to the directory, then initializes the services like the public node did. Once the result of the NAT discovery comes back and the node knows it is not reachable, the directory is updated and the node gets the list of available nodes (i.e. the directory, public node and itself) and waits for the user to choose an action. It also starts picking rendez-vous: it chooses the public node as best candidate and creates the corresponding tunnel while the public node does the same on its side. It then asks the directory to become its backup rendez-vous, since no other public node is available. If NATed node 1 was started before the public node, the result would be the same except that the directory would be the only rendez-vous until the public node is available. The rendez-vous service periodically asks for updated lists of public addresses and checks if the tunnels are alive.
4. NATed node 2 is started, and follows the same steps as NATed node 1.
5. Case 1: NATed node 2 wants to connect to the public node. It asks the directory for contact information about the public node and learns a public address; the direct communication is established and NATed node 2 can send messages to the public node.
6. Case 2: NATed node 1 wants to connect to NATed node 2. It asks the directory for contact information about NATed node 2 and learns the address(es) of its one or two rendez-vous. It connects to NATed node 2 through one of the tunnels (preferably the non-directory node), and since both endpoints are NATed and unreachable directly, messages are exchanged along the tunnel through the public node.
7. Case 3: the public node wants to connect to NATed node 2 (we suppose that the public node is no rendez-vous for NATed node 2 or else the connection is already established). It asks the directory for contact information about NATed node 2 and learns the address of its rendez-vous (i.e. the directory in this case). It connects to NATed node 2 through the tunnel and sends a message asking NATed node 2 to try a reverse connection. NATed node 2 opens a direct connection

to the public node and this direct connection is then used for exchanging chat messages. If the reverse connection fails then the tunnel is used for exchanging chat messages (but there is no reason why it should fail).

3.3 Qualitative Test Results

Quantification tests with many subsequent runs would suffer from test runs tampering with each other (NAT devices keep connection states in their translation table for a given time, and operating systems do not free unused network resources immediately), moreover they would not be representative of the reality because of the proximity between test nodes. This is why a qualitative evaluation seemed to be reasonable within the scope of this work.

- *Efficiency*: as long as the directory is running, the framework always succeeds to establish connections in every test case where a success can reasonably be expected, even if it sometimes means trying again a few times.
- *Robustness*: a fair number of simulated test cases have allowed us to check the proper behaviour of the framework in every considered pathological case (brutal disconnection of a node, huge lag, message loss, message tampering, malicious node, ...). In almost every case, as long as the directory runs properly, the framework has a fair expectable behaviour that does not block any service or cause any definitive error.
- *Performance*: the initialization of all services and transition to a ready-to-use state usually takes at most a few seconds, which is no longer than the network discovery time of P2P applications implementing their own NAT traversal techniques. The establishment of a connection is usually quite instantaneous when the conditions are good. However, the use of relays for communication in the case where both endpoints are NATed implies lower performance compared to frameworks that implement an efficient and successful TCP hole punching.
- *Scalability*: our test equipment did not allow us to verify in practice the scalability of the framework. A particular effort was devoted to saving resources and building a scalable system during all the implementation phase. Most services should be fairly scalable if the number of available public nodes is sufficient; however, the directory is currently not scalable because it is hosted and replicated on a few

nodes which may be single points of failure. In order for the system to become fully scalable, the directory would have to be distributed along a Distributed Hash Table (DHT). Hosting the directory on one or a few nodes was a deliberate design choice motivated by the will to later implement security measures controlled by the directory.

- *Simplicity for the final user*: this of course depends on the application used on top of the framework. However the framework is fully transparent for the user who may only notice some delays during initialization or connection establishment. In particular, the framework does not need the user to know anything about his network topology nor to configure anything related to the network, as long as the application chooses a port number and a directory server address.
- *Simplicity for the software developer*: the framework works as a black box for the application, so that the developer does not have to care about NAT traversal at all. All he has to do in order for his application to use NAT traversal, is use a compatible version of Java, add the few required lines of code to his program and provide the framework with a directory address, an identifier and a port number that it can use. Appendix A provides more details about practical use of the framework in an application.

3.4 Possible Improvements

However good the results are, there is still some work left to do before the new version of NATaWare may be distributed and widely used by developers. The main leads for improvement are the following :

- Implement TCP hole punching and/or UDP-based NAT traversal techniques: relays are too costly, the framework needs a solution to establish direct communication between two NATed nodes, even if that solution is difficult to implement and does not work in every case. Relay must only be a fallback plan in case everything else fails.
- Improve the directory: it is possible either to distribute the directory along a DHT to improve scalability, or to add security features to the directory server (for example authentication and confidentiality).

- Save the framework's state from one execution to another: by saving in a file or a database all information about the current state of the framework (configuration, known addresses, network topology, ...), it would be possible to greatly reduce the initialization time in the next execution of the framework in case the network has not changed much.
- Perform advanced tests on a real-world large-scale platform in order to verify scalability and good behaviour regarding the large variety of NAT devices, and do quantitative performance measurements.

Conclusion

This report dealt with the implementation of a framework for NAT traversal. In a first part it introduced the necessary concepts for a good understanding of the nature and importance of NAT traversal for network applications, before analyzing previous related work and giving an overview of the goals for this project.

With this background, the second part showed the design choices and the multi-service architecture of the project, before focusing on the precise role and behaviour of each main service. It also introduced the simple chat application that has been developed to demonstrate the framework, before summarizing the main steps of the execution.

The third part of this report was finally devoted to qualitatively evaluating the accomplished work. The environment in which the framework had been tested in, was described as well as the tests themselves and their results. Conclusions were drawn and checked against the initial goals and requirements for a satisfying result, and possible improvements were suggested for future work.

References

- [1] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631 (Informational), Mai 1994. Obsoleted by RFC 3022.
- [2] UPnP. Universal plug 'n' play group, link: <http://www.upnp.org>
- [3] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489 (Proposed Standard), March 2003.
- [4] J. Rosenberg et al, Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN), Internet Draft, October 2008, link: <http://tools.ietf.org/id/draft-ietf-behave-turn-11.txt>
- [5] J.L. Eppinger. TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem. reports-archive.adm.cs.cmu.edu, 2005.
- [6] Paul Francis Saikat Guha. STUNT, Simple Traversal of UDP Through NATs and TCP too, link: <http://nutss.gforge.cis.cornell.edu/stunt.php>
- [7] JXTA, Sun Microsystems, Java-based P2P Framework, link: <http://www.jxta.org>
- [8] NATaWare, Sourceforge project, link: <http://sourceforge.net/projects/nataware/>
- [9] D. Eichhorn, A Peer-to-Peer Network Framework with Network Address Translation Traversal, Master Thesis, University of Zurich, 2006, link: http://www.ifi.uzh.ch/fileadmin/site/teaching/Diplomarbeiten/Abgeschlossene_Diplomarbeiten/Jahrgang_2006/Eichhorn_Daniel.pdf
- [10] Java 2 Platform Standard Edition 5.0 API Specification, link: <http://java.sun.com/j2se/1.5.0/docs/api/>
- [11] Apache MINA Project, Apache Software Foundation, link: <http://mina.apache.org/index.html>
- [12] R.T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, University of California, Irvine, 2000, link: <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

A Installation Guidelines

A.1 Requirements

In order to use the framework for the development of an application, the following requirements are needed:

- Java Development Kit 1.5 or later
- Eclipse IDE
- The NATaWare jar archive or source code

The NATaWare project must be imported as a library inside the application Eclipse project. A zip archive containing the full chat application project with all necessary libraries is given.

A.2 Using the Framework

In order to use the framework in the development of an application, you need to add the following contents to the source code of the application :

Code	Explanation
<code>import ch.squix.nataware.NATaware;</code>	Import the NATaWare project
<code>public class ChatMessage extends ApplicationMessage</code>	Creates an application message type to be handled
<code>public class ChatMessageHandler implements MessageHandler<ApplicationMessage></code>	Creates a message handler for application messages
<code>NATaware framework = new NATaware(NodeID, port, directory?, DirectoryId, DirectoryAddress, MessageHandler);</code>	Creates the objects from the framework
<code>library.initDirectory();</code>	Initializes the directory server (must be executed for all nodes)
<code>library.joinNetwork(id);</code>	Joins the network with the chosen node identifier
<code>library.startServices();</code>	Initializes remaining services (non-directory nodes only)
<code>library.getListOfNodes();</code>	Obtains the list of identifiers of all alive nodes
<code>library.openConnection(id);</code>	Connects to the node with the selected id

A.3 Example : Chat Application

In order to run a test with the chat application :

1. Change the hardcoded IP address and port number in both methods `testDirectory()` and `testClient()` to the ones where the directory server will be.
2. Run `testDirectory()` on the directory server.
3. Run `testClient()` on each client. The console prompts you for an ID and a port number. Entering an invalid or already used ID will trigger a new ID prompt; entering no port number or an invalid one will make the framework use the default port number (7878).
4. After a few seconds, the client is given a list of available participants and prompted for an ID. An invalid destination ID will trigger a new list and prompt. Otherwise a connection is established towards the destination ID node; after a few instants, the console displays “Connection succeeded”. Every message you type in will be sent to the destination and displayed in its console.
5. Type “quit” in the console to close the connection to the destination and return to the list of available nodes. Type “quit” a second time to quit the application.
6. Type “quit” in the directory server console to stop it, after all clients have been stopped the same way.