NAT Traversal Techniques and UDP Keep-Alive Interval Optimization

by
Christopher Daniel Widmer

Bachelor of Science
Computer Science
Florida Institute of Technology
2009

A thesis submitted to
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Software Engineering

Melbourne, Florida
July, 2015

We the undersigned committee hereby recommend that the attached document be accepted as fulfilling in part the requirements for the degree of Master of Science in Software Engineering.

"NAT Traversal Techniques and UDP Keep-Alive Interval Optimization"
a thesis by Christopher Daniel Widmer

Marius C. Silaghi, Ph.D.
Associate Professor, Computer Sciences and Cybersecurity
Major Advisor

Ronaldo Menezes, Ph.D.
Associate Professor, Computer Sciences and Cybersecurity
Committee Member

Syed Murshid, Ph.D.
Professor, Electrical and Computer Engineering
Committee Member

Joerg Denzinger, Ph.D. (University of Calgary)
Associate Professor, Computer Sciences
Committee Member

Richard Newman, Ph.D.
Professor and Program Chair, Computer Sciences and Cybersecurity

ABSTRACT

NAT Traversal Techniques and UDP Keep-Alive Interval Optimization

by

Christopher Daniel Widmer

Thesis Advisor: Marius C. Silaghi, Ph.D.

NAT traversal presents a challenge to Peer to Peer (P2P) applications, as in many instances a third party is needed to help initiate communication between two hosts when one is behind one or more NAT devices. Much work has gone into facilitating communication across NATs using various protocols, and several standards have been developed to this end. This thesis explores the advantages and disadvantages of several of these standards, including protocols for interacting with the NAT device itself (NAT Port Mapping Protocol (NAT-PMP), Port Control Protocol (PCP), and Universal Plug and Play (UPnP)), and those using an external server to obtain the external facing address and port mapping of a client (Session Traversal Utilities for NAT (STUN)). The results from a small series of performance tests are also described.

A common technique for maintaining connections through NATs is to use some form of "keep-alive" message so that the connection mapping in the NAT

device will not expire. This thesis explores several existing methods for encoding and scheduling keep-alive messages. In addition, it introduces a new state-based technique, "STUN Calc Keep-Alive", as an extension of the STUN protocol for calculating an appropriate keep-alive interval for a User Datagram Protocol (UDP) connection with the current network configuration and efficiently adapting to NAT mapping lifetime changes. In addition to having the algorithm run on a dedicated connection, two possible implementations for running the algorithm using a "single channel", or on the same connection as other application traffic, are described.

Contents

List of Figures

List of Tables

Glossary

ACK TCP Acknowledgement Packet.

ACT Access Control Table.

ALF Application Layer Framing.

ARP Address Resolution Probe.

CRLF Carriage Return and Line Feed.

CSV Comma-Separated Values.

DCCP Datagram Congestion Control Protocol.

DDNS Dynamic DNS.

DDP2P Direct Democracy P2P (software).

DHCP Dynamic Host Configuration Protocol.

DHT Distributed Hash Table.

DNS Domain Name System.

DNS-ALG Domain Name System - Application Level Gateway.

DoS Denial-of-Service.

FTP File Transfer Protocol.

GENA Generic Event Notification Architecture.

H.323 Audio-Visual Network Communication Recommendations.

HTTP Hypertext Transfer Protocol.

HTTPMU Variant of HTTPU that makes use of IP multicast.

HTTPU Extension of HTTP using UDP instead of TCP for transport.

ICE Internet Connectivity Establishment.

ICE-TCP ICE over TCP.

ICMP Internet Control Message Protocol.

IETF Internet Engineering Task Force.

IP Internet Procotol.

IPv4 Internet Procotol (Version 4).

IPv6 Internet Procotol (Version 6).

NAT-PMP NAT Port Mapping Protocol.

NRT Name Relation Table.

P2P Peer to Peer.

PAL Peer to Peer Abstraction Layer.

PCP Port Control Protocol.

RFC Request For Comments (from the IETF).

RTCP RTP Control Protocol.

RTP Real-time Transport Protocol.

RTSP Real Time Streaming Protocol.

SCTP Stream Control Transmission Protocol.

SDP Session Description Protocol.

SIP Session Initiation Protocol.

SMTP Simple Mail Transfer Protocol.

SNA Systems Network Architecture.

SOAP Simple Object Access Protocol.

SSDP Simple Service Discovery Protocol.

STUN Session Traversal Utilities for NAT.

STUNT Simple Traversal of UDP Through NATs and TCP too.

SYN TCP Synchronization Packet.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TTL Time-To-Live.

TURN Traversal Using Relays around NAT.

UDP User Datagram Protocol.

UPnP Universal Plug and Play.

UPnP-IGD Universal Plug and Play - Internet Gateway Device.

VAT Virtual Address Translation Table.

VoIP Voice-over-IP.

VPN Virtual Private Network.

XML Extensible Markup Language.

XOR Exclusive OR (Logical Operator).

Acknowledgements

I would like to thank my advisor, Dr. Marius Silaghi, for his guidance, mentorship, and patience during the course of my studies. I would also like to thank my thesis committee members for their time and effort: Dr. Ronaldo Menezes, Dr. Syed Murshid, Dr. Joerg Denzinger, and Dr. Richard Newman. Additionally, I would like to thank the Computer Science and Software Engineering departments for their continued hard work to better the education of the students.

Dedication

This thesis is dedicated to my parents for their unwavering support and encouragement, and to my employer for providing support for higher education.

Chapter 1

Introduction

NAT devices, or Network Address Translators, are devices that transparently route traffic between an external network such as the public Internet, and devices on an internal network. A common example would be routers for a corporate network or a smaller home network. Devices, or "hosts", on the internal network behind the NAT device are assigned local addresses that are guaranteed to be unique only within that network. These addresses are not known to devices outside of the NAT. A network or subnetwork that contains a unique set of Internet Procotol (IP) addresses is known as an "address realm" [43]. In the context of NATs, this term is commonly used to differentiate between the set of IP addresses in use behind a particular NAT and those used on the network outside of the NAT.

To enable communication with devices outside of the NAT, the NAT device assigns an external facing, or public, IP address and port to each socket on the internal host, and during communication sessions modifies the internal host's address and port information within the network packets. This is in effect

"translating" the data and allowing network traffic to flow between hosts behind and outside of the NAT [43]. Since pure NATs only modify the transport layer, some applications and protocols need the help of ALG's (Application Level Gateways) to translate addresses in the payload of the packet. In some cases these are built into the NAT itself.

In order for the NAT device to perform this address translation, a mapping, also known as a binding, is created from the local address and port of a host behind the NAT to an external facing address and port that can be seen from outside of the NAT's local network. Most NAT devices will automatically create a mapping when a host on the local network contacts a device outside of it. Actions such as a computer in a corporate network accessing a public website such as "www.google.com" fall into this category. Specialized protocols for traversing NAT architectures such as STUN or Traversal Using Relays around NAT (TURN) utilize this mechanism as well. Another method is for a client behind the NAT to explicitly create a mapping to an external address and port and set up the forwarding on the NAT device. This can be done manually by user configuration, or programmatically by specialized protocols such as NAT-PMP, PCP, and UPnP. One objective of this thesis is to evaluate the performance of these various protocols.

In cases where the NAT device created the mapping automatically, the mapping will time out after a set period of time which is usually configurable. Client applications generally maintain these mappings during periods of inactivity using "keep-alive" messages sent intermittently through the NAT device. The second objective of this thesis is to explore a new method of calculating an optimium interval for these keep-alive messages using an extension to the STUN

protocol.

1.1 The Problem

The first issue addressed by this thesis is to evaluate different methods of creating an external facing address and port mapping with a NAT device, highlighting the advantages and disadvantages of each. While some of these methods have been studied in the past, one goal of this thesis is to provide a more focused comparison that can serve as the basis for further research.

Mappings created automatically by NAT devices time out after a specified period of time. As previously mentioned, this is often addressed by client applications sending periodic "keep-alive" messages to maintain the connection when there is no data to be sent. There are suggested intervals for sending these messages based on various protocols, such as the ones described in [33]. Some research has been done to determine the optimum keep-alive interval for a given network architecture between two devices, but to our knowledge little experimental data is available. The second problem examined by this thesis is how to determine an optimium keep alive interval for a UDP connection when the network architecture is unknown.

1.2 The Approach

Research on both the official standards specifications and related work was done for the STUN, NAT-PMP, PCP, and UPnP protocols, both for client/server and P2P usage. These materials were used for evaluating and comparing these

protocols when used for obtaining and mapping an external facing address and port. In addition, timing tests were done using these protocols to map an external port (automatically in the case of STUN) and obtain their external address in order to determine if there was any performance advantage to using a particular approach.

The exploration of existing approaches to determining the optimum keep-alive interval, as well as the presentation and experimental evaluation of a new algorithm in later chapters offer a look at the best ways to approach the issue of keep-alive messages over UDP.

1.3 Research Objectives

The objectives of this thesis are summarized below.

- Summarize and compare common problems with NAT Traversal.
- Evaluate the advantages and disadvantages of using NAT-PMP, PCP, UPnP, and STUN to obtain an external address and port mapping from behind a NAT device.
- Experimentally evaluate the performance of NAT-PMP, PCP, UPnP, and STUN for obtaining an external address and port mapping.
- Analyze existing research into optimum keep-alive intervals for various protocols, as well as methods to calculate the optimum interval over UDP for a given situation.

- Describe and experimentally evaluate a new extension to the STUN protocol used to calculate an optimum keep-alive interval over UDP.

1.4 Thesis Outline

This thesis is organized as follows:

- Chapter 2 introduces general concepts, configurations, and problems involved in NATs, and discusses several common traversal techniques.
- Chapter 3 offers an introduction and comparison of the NAT-PMP, PCP, UPnP, and STUN protocols, as well as an experimental evaluation of their performance. It also explores some more specialized protocols used for NAT traversal.
- Chapter 4 explores the need for UDP keep-alive messages when communicating through NATs, describes existing methods for determing an optimium keep-alive interval, and introduces and evaluates a new method for finding an optimium value for the keep-alive interval.
- Chapter 5 offers conclusions and ideas for future work to summarize the presented research.

1.5 A Note on Terminology

The terminology used throughout the literature reviewed for this thesis varies in describing common concepts. For example, individual devices involved in communication through a NAT are described as “hosts”, “peers”, “nodes”, and

"servers" in various pieces of literature. For the sake of clarity when discussing the topic, the term "node" will be used in most cases to describe a device on the network, and a "mapping" or "binding" from a private local address and port behind a NAT to an external facing address and port will be referred to as a "mapping". Alternative terms will be used in situations where it makes sense to do so.

Chapter 2

NAT Traversal Background

2.1 Common Configurations

As stated earlier, there are several possible types and configurations for NATs. This section discusses some of the most common [43].

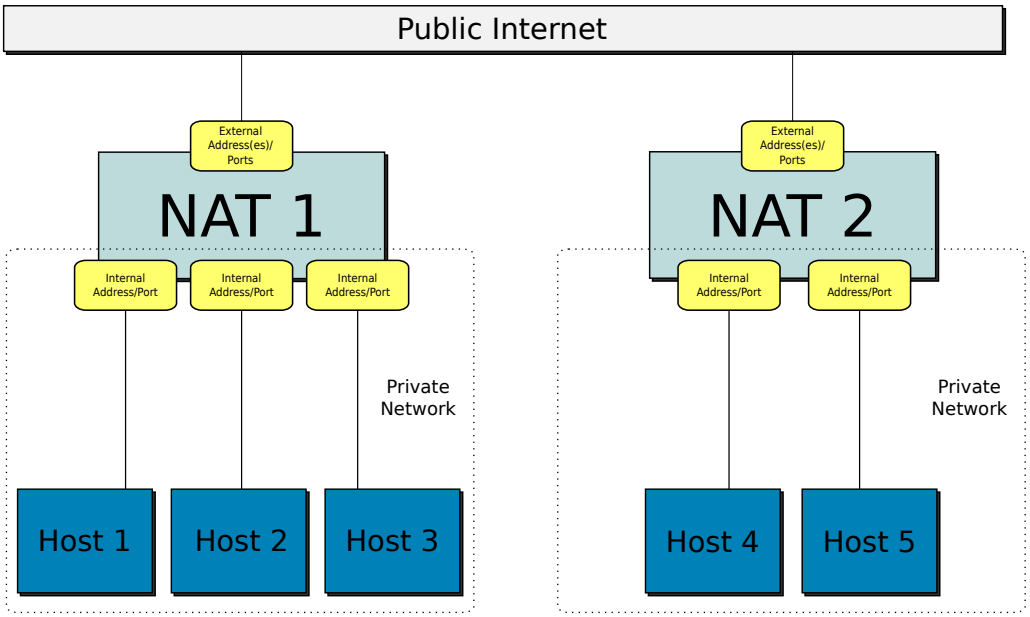


Figure 2.1: An Example Showing Two NATs.

2.1.1 NAT Configurations

Traditional NAT Nodes in the private network behind the NAT can initiate sessions with nodes outside of the NAT, but not vice versa. In addition to the IP address, checksums for IP, Transmission Control Protocol (TCP), UDP, and Internet Control Message Protocol (ICMP) headers are also translated. There is only a mapping in the outbound direction, so only outbound-initiated sessions are allowed with this configuration. An extension of this type is called a *Basic NAT*, in which a set of addresses are set aside to be mapped to internal nodes and allow inbound communication sessions to those nodes. A further extension, known as *Network Address Port Translation (NAPT)* also translates port identifiers (such as port numbers for TCP and UDP). With this setup, multiple nodes behind the NAT can use the same external address with different ports. Only the address and port of either the source of destination are translated.

Bi-Directional NAT Sessions can be initiated both from a node outside the NAT to one behind it, and vice-versa, as private addresses behind the NAT are mapped to unqiue external facing addresses, either statically or dynamically as needed. Nodes behind the NAT are accessible using a Domain Name System (DNS) for address resolution. A Domain Name System - Application Level Gateway (DNS-ALG) is also needed to assist with address mapping. As with the *Traditional NAT*, only the address and port of either the source of destination are translated.

Twice NAT Unlike the previously mentioned NAT types, in this case the IP addresses of both the source and the destination are translated. It requires

DNS lookups for addresses and a DNS-ALG. This type is most useful when addresses may be duplicated in the local network behind the NAT and the network outside of the NAT.

Multihomed NAT A private network makes use of multiple NATs, with the sessions going through a particular NAT device depending on the destination. This also sets up a backup in case one NAT device fails.

There are four main types of NAT behavior, although some configurations use aspects of one or more of these in combination [23]. The original specification for the STUN protocol [41] supplied a method to detect which of these types the client node was behind, but this functionality was not included in the current specification [54] due to unreliability and the existence of additional NAT configurations used in practice.

Symmetric In this case a local address and port behind a NAT is bound to an external facing address and port, but that external facing address and port can only be used to communicate with one particular destination address and port outside of the NAT. This is the most restrictive form of NAT.

Full-Cone Like most NATs, this involves the mapping of a local address and port behind a NAT to an external facing address and port. However, unlike with Symmetric NATs, the external facing address and port can be used to connect with any destination address and port outside of the NAT. This is the least restrictive form of NAT.

Restricted-Cone This is a more restricted version of Full-Cone. The external

facing address and port of the node behind the NAT can only be used to communication with a particular destination address, but any port on that address can be used.

Port-Restricted-Cone This is another more restrictive version of Full-Cone. Unlike Restricted-Cone, the external facing address and port of the node behind the NAT can be used to communicate with any destination address, but it is restricted to the port originally used for the binding.

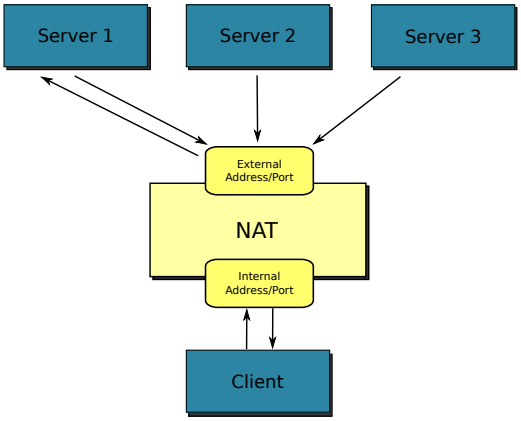


Figure 2.2: Full Cone NAT

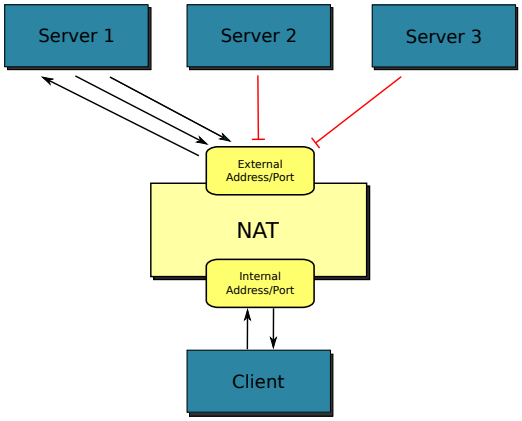


Figure 2.3: Restricted Cone NAT

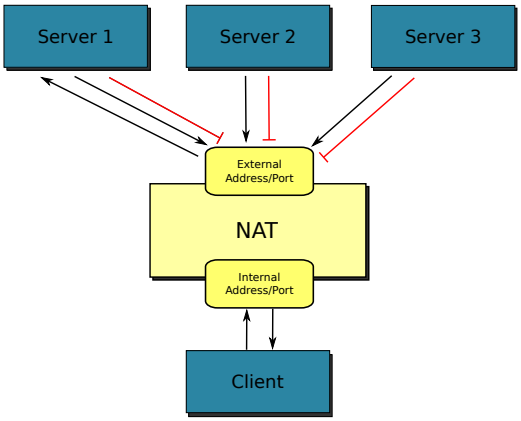


Figure 2.4: Port Restricted Cone NAT

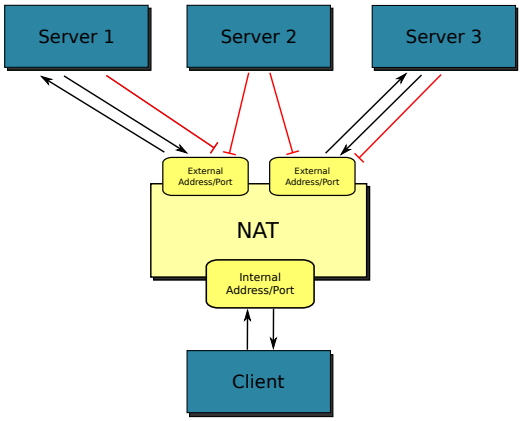


Figure 2.5: Symmetric NAT

Another way to describe the mapping and filtering behavior of NAT devices is based directly on the endpoints. The descriptions below are based on information from [23], [21], and [7].

2.1.2 Mapping Behaviors

Endpoint-Independent Mapping The mapping to an external facing address and port for a node's private address and port does not include the destination endpoint, and is re-used for multiple sessions. For example, a public mapping created for Node A to communicate with Node B outside of the NAT can be re-used for Node A's communication with Node C outside of the NAT.

Endpoint-Dependent Mapping Either the address or port of the destination endpoint are included in the mapping, so there is no re-use for different destinations. The inclusion of destination information can include only the address, or both the address and port.

Session-Dependent Mapping This is relevant only for TCP, and involves the NAT device creating a new mapping if a TCP connection between a node behind the NAT and a node outside of the NAT has been closed and re-opened.

2.1.3 Filtering Behaviors

Endpoint-Independent Filtering No filtering is enforced on existing external facing address and port mappings. Any external source can send data to the mapped external facing address of a node behind the NAT.

Endpoint-Dependent Filtering The NAT device will filter out any data that does not originate from the original destination of the external facing address and port mapping. This can apply to only the destination address, or it can apply to both the address and port.

NAT behavior can also be characterized by whether or not a NAT supports the "hairpin" operation, where the external facing address and port mapping can be used for communication by two nodes when both are behind the NAT [28]. Another aspect is whether or not the NAT is deterministic. Some NATs may produce different bindings depending on the ordering of outbound traffic.

2.2 Common Problems with NATs

In general, NATs cause issues with applications that use IP addresses in the packet payload, although this can sometimes be relieved with the use of ALGs (e.g. File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), DNS, etc). This can also cause issues with less popular transport protocols (such as Stream Control Transmission Protocol (SCTP) and Datagram Congestion Control Protocol (DCCP)) that may not be recognized or supported by all NATs [43]. A related example are security protocols such as IPSec, which include an integrity check that involves a checksum the entire payload, and are broken by NATs due to the IP address change [23].

Applications that have separate control and data sessions that are interdependent can run into issues when used over NATs [43]. This is true of protocols such as FTP, H.323, Session Initiation Protocol (SIP) and Real Time Streaming Protocol (RTSP) [44]. In these cases the NAT does not know which sessions

depend on which, so it treats them all independently. In addition, without special handing, any application that requires that the two communicating nodes retain the same address across sessions may be broken by a NAT, since the NAT device has no knowledge that this is required. IP fragmentation is also a problem, since only the first fragment contains the transport header. There are various of methods to handle this depending on the NAT; for example, the NAT reconstructing the packet itself, or simply guessing based on previously received packets. Even the use of TCP as a transport protocol presents issues, as the normal client and server roles need to be reversed to establish a connection.

P2P applications also have issues traversing NATs without help, as each node may be behind the same NAT as the other, behind another NAT, or on a public network. Additional issues related to NAT traversal for P2P applications are discussed in Section 2.3.

2.3 General Peer-to-Peer (P2P) NAT Traversal Methods

Because this research is done in support of P2P applications, some general information on P2P NAT traversal is presented in this section. In the Request For Comments (from the IETF) (RFC) 5128 [28], common NAT traversal techniques currently in use are discussed, as are some relevant security considerations when implementing various solutions. Many of these techniques have been adapted, extended, or included as part of more specialized techniques to facilitate NAT traversal in additional situations. Selected research in those areas is described

in the Section 3.2.7. P2P applications are more affected by NATs than those using client/server communication since both of the nodes may be behind one or more NATs. With client/server architectures, the server is usually on a public network and only the client is potentially behind a NAT.

Relaying In this situation, two nodes use an external node or server outside of the NAT to relay messages between them. It is the basis of the TURN protocol, and is also used in several of the NAT traversal solutions discussed in Section 3.2.7.

Connection Reversal With this solution, one of the nodes needs to be outside of the NAT. Node A behind the NAT registers its public address with a rendezvous server. Since it cannot directly contact Node A behind the NAT, the Node B on the outside of the NAT can use the rendezvous server to tell Node A behind the NAT to instead initiate a connection with it.

Hole Punching This technique involves creating mappings initially using a rendezvous server in order to learn the address of the other node. When using UDP, the nodes can then establish a direct connection by contacting the public address of the other node and also requesting that the rendezvous server have the other node contact their public address. Once the connection is established, the rendezvous server is no longer needed. For TCP, the two nodes send each other TCP Synchronization Packet (SYN) packets at the same time, and then respond with TCP Acknowledgement Packet (ACK) packets to establish the connection. This is reliable as long as the NAT devices support Endpoint-Independent Mapping. Many ap-

plications use this technique, making use of STUN to obtain their external address.

Port Number Prediction This technique involves predicting the next port number assigned based on the past behavior of the NAT device. It is not always reliable, especially for TCP.

Some issues with these solutions are discussed in [14]. These include items such as the need for excessive keep-alive messages for UDP connections, as well as dealing with unknown NAT architectures.

If applications do not implement some type of authentication, they risk security problems, especially in situations where UDP/TCP hole punching is used [28]. Public rendezvous servers are also vulnerable to Denial-of-Service (DoS) attacks, as nodes can register themselves and claim to have any IP address. In this situation there is usually not a reliable way to verify the private addresses. The literature suggests throttling network traffic as a mitigation technique. The use of public rendezvous servers also opens the door to man-in-the-middle attacks, where an attacker could pretend to be the server and intercept client node data. The only mitigation suggestion is to encrypt application data.

Chapter 3

Analysis of Methods to Obtain an External Facing Address

3.1 Introduction

This chapter provides a summary and comparison of five protocols related to NAT traversal: NAT-PMP, PCP, UPnP, STUN, and TURN. It also explores some more specialized NAT traversal tecthniques designed overcome issues with the more standardized protocols.

In addition, it describes an experiment testing the performance of the first four of those protocols in various situations, along with a discussion of their use in NAT environments today.

3.2 Background

The protocols described in this section are commonly used, well supported, and designed to assist with NAT traversal. The first three: NAT-PMP, PCP, and UPnP, are designed for interaction with the NAT device directly connected to a client. They allow a client to explicitly create, renew, and destroy external facing address and port mappings. The STUN protocol is used to create an external facing address and port mapping automatically by creating an outoing request to a server outside of the NAT and returning the public address and port information to the client. The TURN protocol is used to relay messages between clients that are behind NAT devices using an external server. An overview of each is given below, followed by a summarized comparison.

3.2.1 NAT-PMP (NAT Port Mapping Protocol)

NAT-PMP is a UDP protocol used in NAT devices to create external facing address mappings for nodes behind the NAT in order to allow nodes outside of the NAT to communicate with them [29]. The server, located on the NAT device, uses a request/response system to communicate with the clients. The protocol supports determining a client's external facing address, announcing address changes, and both requesting and destroying address mappings. While the protocol itself operates only over UDP, it supports mappings for both UDP and TCP. It has been "superseded" by PCP, which retains backwards compatibility by using a version bit in the packets to signify whether the protocol being used is NAT-PMP (version 0) or PCP (version 2). Many aspects of NAT-PMP were inspired by the Dynamic Host Configuration Protocol (DHCP) protocol.

It includes a "Seconds Since Start of Epoch" field (up-time of the NAT device) to keep track of mapping lifetimes and to aid in determining whether or not a mapping is still valid. The protocol will only work on NATs that are capable of creating UDP and TCP mappings independently. In order to request a mapping, a client sends a NAT-PMP request and receives a response. In the case where a mapping already exists for the requested port, an alternative mapping will be returned to the client. An interesting note is that there is no field in the packet format for addresses. This is because the protocol uses the source IP address of the client as the internal address, as it is only meant to be used by hosts creating their own mappings. Hosts behind the NAT can request information about the external facing address mapping via a specialized two byte request.

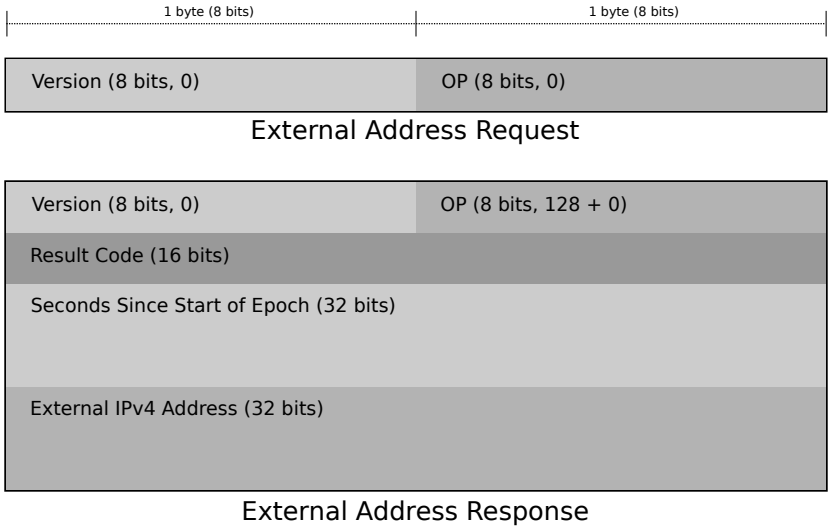
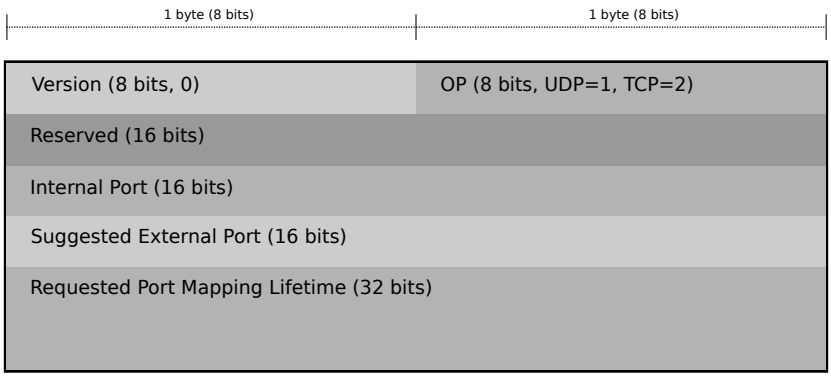
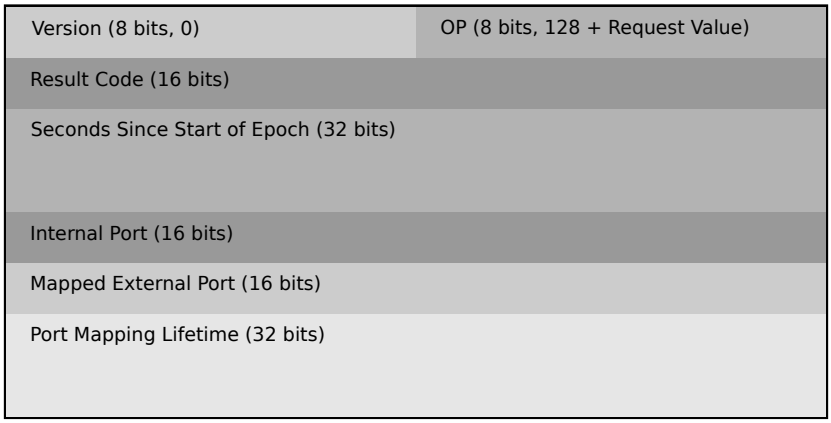


Figure 3.1: NAT-PMP External Address Request/Response



Mapping Request



Mapping Response

Figure 3.2: NAT-PMP Mapping Request/Response

Mappings are deleted by the client sending a new mapping request for a particular port with a requested mapping lifetime of zero, or by the client simply allowing them to time out. In the former case the server sends a response as usual. All mappings created via NAT-PMP are bi-directional. When a NAT device reboots, it must reset the "Seconds Since Start of Epoch" value and re-announce its address via multicast to all clients so that they can send mapping renewal requests sooner than planned if necessary.

NAT-PMP is only designed to work with a client behind a single NAT device. It does not offer direct support for nested NATs. The protocol will also not work

in cases where the NAT device has multiple external Internet Procotol (Version 4) (IPv4) addresses, or where the router address may be different from the device actually handling NAT functionality. It also depends on the NAT device supporting "hairpinning", or being aware when two clients both behind it are trying to communicate and acting accordingly. It is the simplest of the protocols described in this section.

3.2.2 PCP (Port Control Protocol)

PCP is an evolution of NAT-PMP and like NAT-PMP, is used to create and maintain external address and port mappings for hosts behind NAT or firewall devices. In addition, one of the features is an included mechanism to help reduce the "keep-alive" network traffic normally used to keep the mappings alive. Like NAT-PMP and Universal Plug and Play - Internet Gateway Device (UPnP-IGD), PCP allows clients to create, renew, and destroy address and port mappings [7]. The protocol supports most NAT configurations and can set up mappings for common transport protocols including UDP, TCP, SCTP, and DCCP. However, like with NAT-PMP, actual PCP messages must be sent using the UDP protcol.

PCP uses a server/client architecture and allows communication via a request/response model. However, the request/response ratio is not always one to one and in the case of multiple requests from a host, the ordering of responses from the server may vary. In most cases the PCP server is implemented as part of the NAT device. However, unlike NAT-PMP, PCP supports Internet Procotol (Version 6) (IPv6) and uses a 128-bit address field for the IP

address for both IPv4 and IPv6 addresses. Another difference is that there is no mechanism in PCP to simply request information about an external address mapping without requesting a new one (or renewing). Like with NAT-PMP, the client is responsible for retransmitting request messages to ensure reliable delivery. The mapping request includes a lifetime parameter, but the lifetime of the actual mapping may be less than the requested length. PCP has a more complex message structure than NAT-PMP and offers several operations, which are described below.

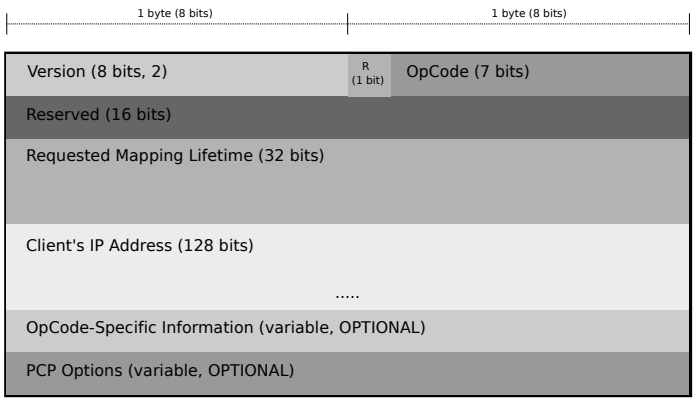


Figure 3.3: PCP Message Headers

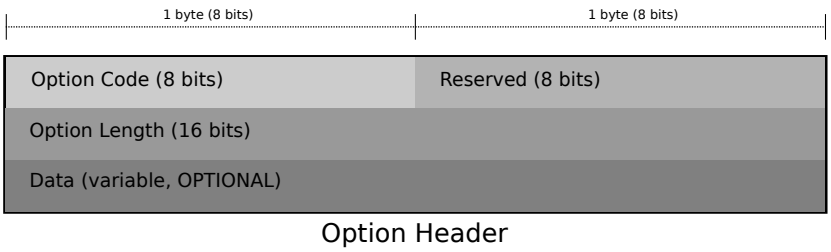


Figure 3.4: PCP Option Header

Which options are appropriate depend on the Opcode-Specific Information entry for each request or response. The Epoch Time field is based on several aspects of the PCP server state (e.g. uptime and external facing address change) and is used by clients to determine if the server has lost its mappings. The version field is currently used to distinguish the request/response as PCP (version 2) or NAT-PMP (version 0), and allows for future PCP versions to change the packet format. The protocol includes error definitions to indicate that the server does not support the packet version, but does support previous versions.

The MAP opcode is used both to establish a mapping between an internal address and port and an external facing one and to renew an existing mapping. These mappings are endpoint-independent.

In order to renew a mapping, a new request is sent from the client containing the currently assigned address and port as suggestions. The specified Internal Port can be 0 to map all incoming traffic for a particular protocol. In order to remove a mapping, a request with a MAP opcode is sent with a Requested Mapping Lifetime of 0.

The PEER opcode is used to establish or renew an outbound mapping, from the NAT to a remote node's address and port. These mappings behave similarly to the implicit outbound mappings created by NATs. The RFC suggests that PCP servers should have an ability to disable these request types.

One specialty use of PCP is to reduce the amount of "keep-alive" network traffic generated by applications. To do this, a client uses PCP requests with the PEER opcode with a custom mapping lifetime. Periodic messages can then be sent at the custom interval to extend the mapping.

There are several options that can be attached to MAP and PEER opcode

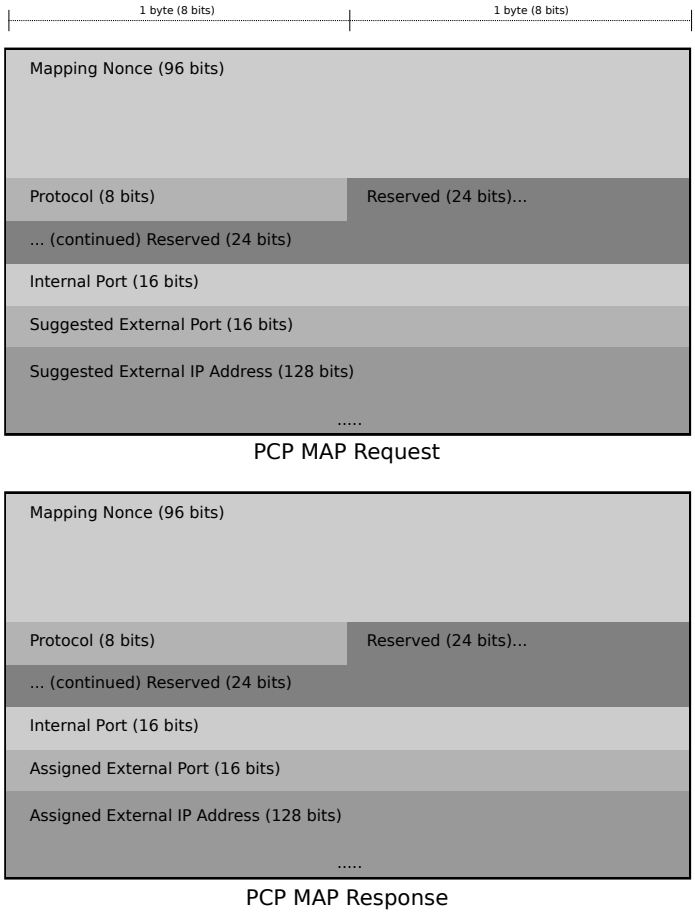
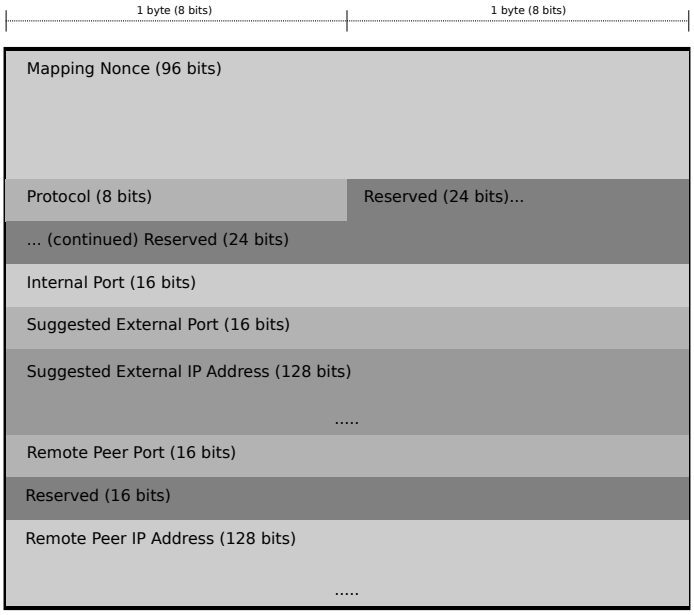


Figure 3.5: MAP OpCode

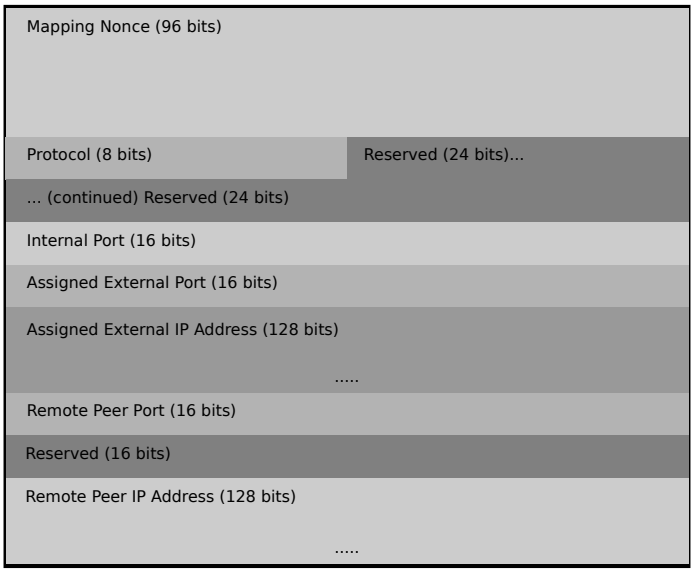
messages to modify the mapping behavior.

THIRD_PARTY Used in both MAP and PEER opcodes. Allows a PCP client to control mappings on internal nodes other than itself (e.g. a situation where a node manages others).

PREFER_FAILURE Used with the MAP opcode to tell the PCP server that if the suggested address and port cannot be mapped, no mapping should be created. This is most useful when working with other protocols such as UPnP-IGD.



PEER OpCode Request



PEER OpCode Response

Figure 3.6: PEER OpCode

FILTER Used with the MAP opcode to indicate that incoming packets should be filtered based on the protocol specified in the MAP request.

PCP adds many capabilities to NAT-PMP, including IPv6 support and im-

proved threat mitigation. The message format is more complex, allowing for more flexibility and extensibility. A particularly interesting feature is the inclusion of the Client IP Address item in PCP request messages. This allows the NAT device to verify that the client node's local address as seen by the client node is actually the one assigned to it by the NAT device; in other words there are no unexpected NAT devices between the one receiving the request and the client node.

PCP also enhances the NAT device recovery mechanism present in NAT-PMP (Seconds Since Start of Epoch). When using PCP, one recovery method that works for TCP connections is for the PCP clients to send new PEER requests with their old address and port as the suggested items. PCP also supports "rapid recovery" which is a much quicker method [7]. This is done via the ANNOUNCE opcode, which allows a PCP server to let the clients know that it has lost its state via multicast to a specific address or alternatively via unicast to known clients. This opcode can also be used by clients to check if a server is still available. Unlike the MAP and PEER opcodes, there is no specific payload for messages with the ANNOUNCE opcode. Another rapid recovery feature is a mapping update, where the PCP server has not lost any mappings, but knows that they are no longer valid. In this case the server unilaterally sends MAP or PEER responses to its clients that have existing mappings to update each of them with the correct new mapping.

As mentioned earlier, PCP contains mechanisms for easy operation with other protocols. In [6], functionality is described to allow both PCP and UPnP-IGD to co-exist on a NAT device (or any IGD).

3.2.3 UPnP (Universal Plug and Play)

UPnP is an extension of the "Device Plug and Play" by Microsoft, with the standard currently maintained by the Universal Plug and Play Forum. The goal is to provide the ability to transparently add devices to a network and allow them to communicate with the existing devices with little or no configuration [1]. The protocol is meant to be used by all types of devices, including computers, mobile devices, entertainment devices, and appliances. Communication is handled with standard transport protocols and is meant to be agnostic to the operating systems used on various devices. It is used extensively for media streaming and control, and can theoretically be used over any medium that supports the bandwidth needed for the protocols used. Protocols used include Extensible Markup Language (XML), Hypertext Transfer Protocol (HTTP), Extension of HTTP using UDP instead of TCP for transport (HTTPU), TCP/IP, etc. A short description of UPnP protocol uses and operation is below; additional protocols are supported via bridging. Operations specific to NAT traversal are handled with the UPnP-IGD protocol, which is described later in this section.

Summary of Communication Protocols and Uses

- HTTPU, Variant of HTTPU that makes use of IP multicast (HTTPMU), Simple Service Discovery Protocol (SSDP), Generic Event Notification Architecture (GENA) [Discovery]
- HTTP [Description, Events]
- Simple Object Access Protocol (SOAP) [Control]
- UDP/TCP
- IP

A network using UPnP has one or more control points, which are devices that handle the discovery and control of other devices on the network. There are several steps involved with UPnP.

Addressing When first connecting to a UPnP network, a device must obtain an IP address via its DHCP client or use Auto IP if it cannot find a DHCP server. The check for a DHCP server must be done periodically. The device also tests the address using an Address Resolution Probe (ARP). Each device may also include a DNS client to locate other devices by identifiers other than their IP address [2].

Discovery Devices on a UPnP network use the SSDP to communicate with the various control points (e.g. to notify them of its services). The "discovery" message sent by the device to the control point includes a URL where the control point can retrieve the XML encoded UPnP description of the device.

Description The UPnP description contains information such as model and serial numbers, manufacturer, vender-specific URLs and more. A listing of any embedded devices and services is also included.

Control A control point gets the UPnP description in XML for all the services offered by a device. Once this is done, it can send control messages to the URL for a particular service. These messages and the device responses are sent using SOAP.

Eventing Services send "event" messages (encoded in XML and GENA) when their state changes. The delivery of these messages is controlled using a

subscription model to allow for multiple control points. When a device initially "subscribes" to a service, it receives an event message containing all of the state variables in their initial state. This allows each subscriber to set up a model for the state of that service [2].

Presentation Some devices contain a URL for a presentation service, which allows the control point to view status information or control the device, depending on the particular service.

The usage of UPnP for NAT traversal is outlined in the description of UPnP IGD1 [53]. While the most recent standard is actually UPnP IGD2 [46], the routers used for the experiments in this thesis only support Version 1, so that is the focus of this description.

"IGD" stands for "Internet Gateway Device", which in this context is the NAT device. As with normal UPnP functionality, messages are exchanged using the XML format, and use the same data types defined in the base UPnP standards. The protocol works over UDP or TCP depending on the action. It uses Comma-Separated Values (CSV), lists, which can contain heterogeneous data (e.g. boolean, integer, string, etc) within the XML. Like other devices on a UPnP network, an IGD is controlled by a UPnP control point, in most cases the client node. The protocol can handle cases where the client node behind the NAT and the control point for the IGD are on the same device as well as cases where they have different IP addresses.

The protocol contains functionality for obtaining general information about port mappings, with functions such as GetGenericPortMappingEntry() and GetSpecificPortMappingEntry(). It also allows for the creation and deletion

of port mappings, with the functions AddPortMapping() and DeletePortMapping(). The mappings created by the service can conflict with mappings created by other entities, so an error code is provided in cases where a conflict is detected.

Information about the IGD itself and the mapped connections are tracked with UPnP state variables. A list of some of the more notable variables is shown below.

- ConnectionType
- UpTime
- NATEnabled
- PortMappingLeaseDuration (range 0-604800)
- PortMappingProtocol (UDP, TCP, or vendor-defined)
- InternalClient
- InternalPort
- ExternalIPAddress
- ExternalPort

Like with the other UPnP services, events are used to communicate changes in state variables, and actions are used to control the IGD. These actions include functionality such as setting a new protocol, enabling and disabling port mapping, changing the timeouts, and getting general status information. In terms of port mapping, unlike with NAT-PMP and PCP, an error is generated if the "ExternalPort" for the specified port is already mapped to an internal client, instead of like the former cases where a different port mapping is applied. One

item of note is the "NewLeaseDuration" action, which modifies the "PortMappingLeaseDuration" variable. The NAT-PMP specification states that these lease duration aspects are rarely used in UPnP [29], although the UPnP specification simply states that a default value must be used instead of zero. Unlike with protocols such as NAT-PMP and PCP, the specification does not require that the IGD retain port mappings after device resets. However, this can be done if desired by using the control point to rebuild the mappings.

The port-mapping itself is conceptually viewed as an 8-tuple:

```
<PortMappingEnabled, PortMappingLeaseDuration, RemoteHost,
 ExternalPort, InternalPort, PortMappingProtocol,
 InternalClient, PortMappingDescription>
```

Figure 3.7: UPnP Port Mapping Data

For general use, initially a "discover" request is sent to UDP port 1900 on the NAT device to retrieve an XML listing of the UPnP services available. Once the control URL for the needed service is acquired, the external facing address and port mapping is done via the UPnP methods AddPortMapping(), DeletePortMapping(), and GetExternalIPAddress(). XML Descriptions of the SOAP messages for these methods are shown in Figures 3.8, 3.9, and 3.10 below.

```
<?xml version="1.0"?>
<s:Envelope xmlns:s=
  "http://schemas.xmlsoap.org/soap/envelope/"
 s:encodingStyle=
  "http://schemas.xmlsoap.org/soap/encoding/">
  <s:Body>
    <u:AddPortMapping
      xmlns:u="urn:schemas-upnp-org:service
               :WANIPConnection:1">
      <NewRemoteHost></NewRemoteHost>
      <NewExternalPort>3478</NewExternalPort>
      <NewProtocol>UDP</NewProtocol>
      <NewInternalPort>3478</NewInternalPort>
      <NewInternalClient>192.168.1.123</NewInternalClient>
      <NewEnabled>1</NewEnabled>
      <NewPortMappingDescription>
        Some Description
      </NewPortMappingDescription>
      <NewLeaseDuration>120</NewLeaseDuration>
    </u:AddPortMapping>
  </s:Body>
</s:Envelope>
```

Figure 3.8: UPnP AddPortMapping() Method

```
<?xml version="1.0"?>
<s:Envelope xmlns:s=
   "http://schemas.xmlsoap.org/soap/envelope/"
 s:encodingStyle=
   "http://schemas.xmlsoap.org/soap/encoding/">
   <s:Body>
     <u:DeletePortMapping xmlns:u="urn:schemas-upnp-org
                                   :service
                                   :WANIPConnection:1">
       <NewRemoteHost></NewRemoteHost>
       <NewExternalPort>3478</NewExternalPort>
       <NewProtocol>UDP</NewProtocol>
     </u:DeletePortMapping>
   </s:Body>
</s:Envelope>
```

Figure 3.9: UPnP DeletePortMapping() Method

```
<?xml version="1.0"?>
<s:Envelope xmlns:s=
  "http://schemas.xmlsoap.org/soap/envelope/"
 s:encodingStyle=
  "http://schemas.xmlsoap.org/soap/encoding/">
  <s:Body>
    <u:GetExternalIPAddress xmlns:u="urn:schemas-upnp-org
                                     :service
                                     :WANIPConnection:1">
    </u:GetExternalIPAddress>
  </s:Body>
</s:Envelope>
```

Figure 3.10: UPnP GetExternalIPAddress() Method

3.2.4 STUN (Session Traversal Utilities for NAT)

Unlike the protocols discussed in the previous sections, STUN is not used to explicitly create an external facing address and port mapping. STUN is a NAT traversal and discovery protocol that does not depend on any particular functionality of the NAT devices used. It is a client/server system and is designed to be used on its own or as part of a larger set of components in the context of NAT traversal. STUN can be used by a node to discover its external facing IP address in order to allow communication with other nodes that may or may not be behind NATs [54]. It is a binary protocol and can be used over UDP, TCP, or TCP-over-Transport Layer Security (TLS). When operating over UDP, the STUN client must use re-transmission as a means of achieving delivery reliability. If used as the sole protocol in a TCP connection, no re-transmission logic is needed. STUN is known to work with full-cone NATs, restricted-cone NATs, and port-restricted-cone NATs. It does not work with symmetric NATs. However, according to [23], this type of configuration is becoming more rare due to its interference with several types of applications.

STUN supports two transaction types: request/response for two way communications, and "indications" for one-off messages. The STUN server is located on the outside of the NAT, while the clients may be behind one or more NATs. The message format is relatively extensible, having the potential to support many types of operations, although only one is supported by the base protocol. The header for a STUN packet contains a "method" (specifying which request or indication is desired), a class, and a transaction ID (used to match requests and responses). The basic structure of the header is shown below in Figure 3.11.

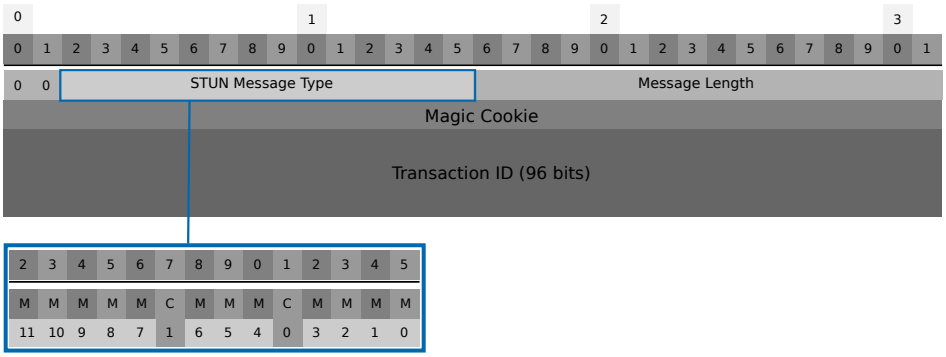


Figure 3.11: Common Message Header

A STUN message may also contain a number of attributes. The protocol defines a set of standard error codes that are included as attributes in STUN messages.

In the basic operation of STUN, the client, which knows the public address of the server, sends a binding request to the server. As the request passes through one or more NATs, the IP address and port of the client will be translated within the packets. When the STUN server receives the request, it sets the "MAPPED-ADDRESS" attribute in the binding response to the client's most

public facing address and sends it back to the client. An Exclusive OR (Logical Operator) (XOR) obfuscated version of the public address is also set in the "XOR-MAPPED-ADDRESS" attribute. This is done since some NAT devices or ALG's that may be in use re-write any instance of the IP address in the packet, making the "MAPPED-ADDRESS" data not always reliable. After this exchange, the client knows its external facing IP address and port. In order for two nodes to set up direct communication, they both go through this process to discover their most public facing address. After exchanging these addresses via a signalling mechanism, they can begin communicating directly (e.g. via a hole punching method). The structure of these attributes is shown below.

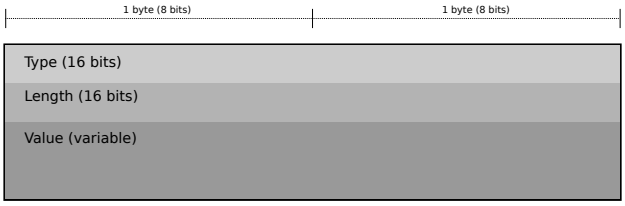


Figure 3.12: Common Attribute Header

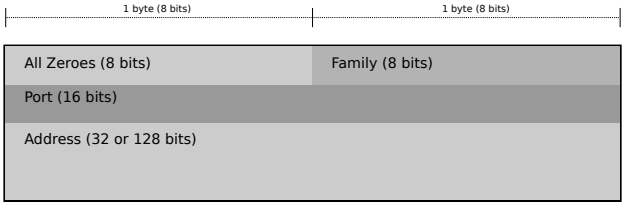


Figure 3.13: MAPPED-ADDRESS Attribute

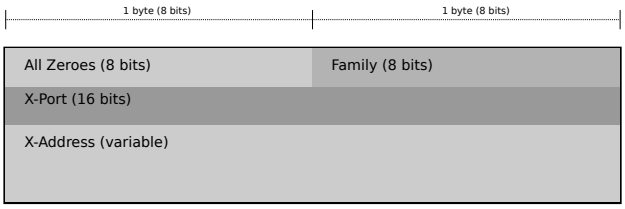


Figure 3.14: XOR-MAPPED-ADDRESS Attribute

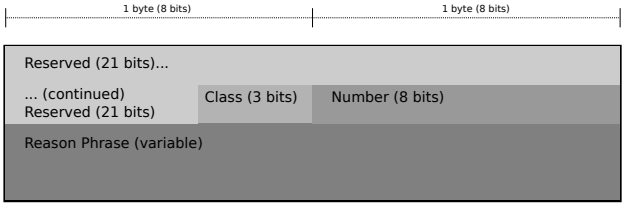


Figure 3.15: ERROR-CODE Attribute

A simple example of two nodes initiating communication with the help of STUN and UDP hole punching is described below and shown in Figure 3.16. This technique will not work with symmetric NATs as the external address and port seen by the server will be different than any usable by the other node.

1. Node 1 uses the STUN server on the public internet to obtain its external facing address and port.

2. Using a signalling mechanism (e.g. a messaging service, a rendezvous server, SIP, etc) Node 1 communicates its external facing address and port and desire to connect to Node 2.

3. Node 2 then uses the STUN server on the public internet to obtain its external facing address and port, and uses the signalling mechanism to communicate the information to Node 1.

4. Both nodes attempt to connect to each other over UDP using the received address and port information for the other node. With most NAT con-

figurations, the first message from each will be dropped, but subsequent messages will be successful as the NAT devices for both nodes will have a $\langle source, destination \rangle$ mapping for the other and therefore allow incoming connections from each respective node.

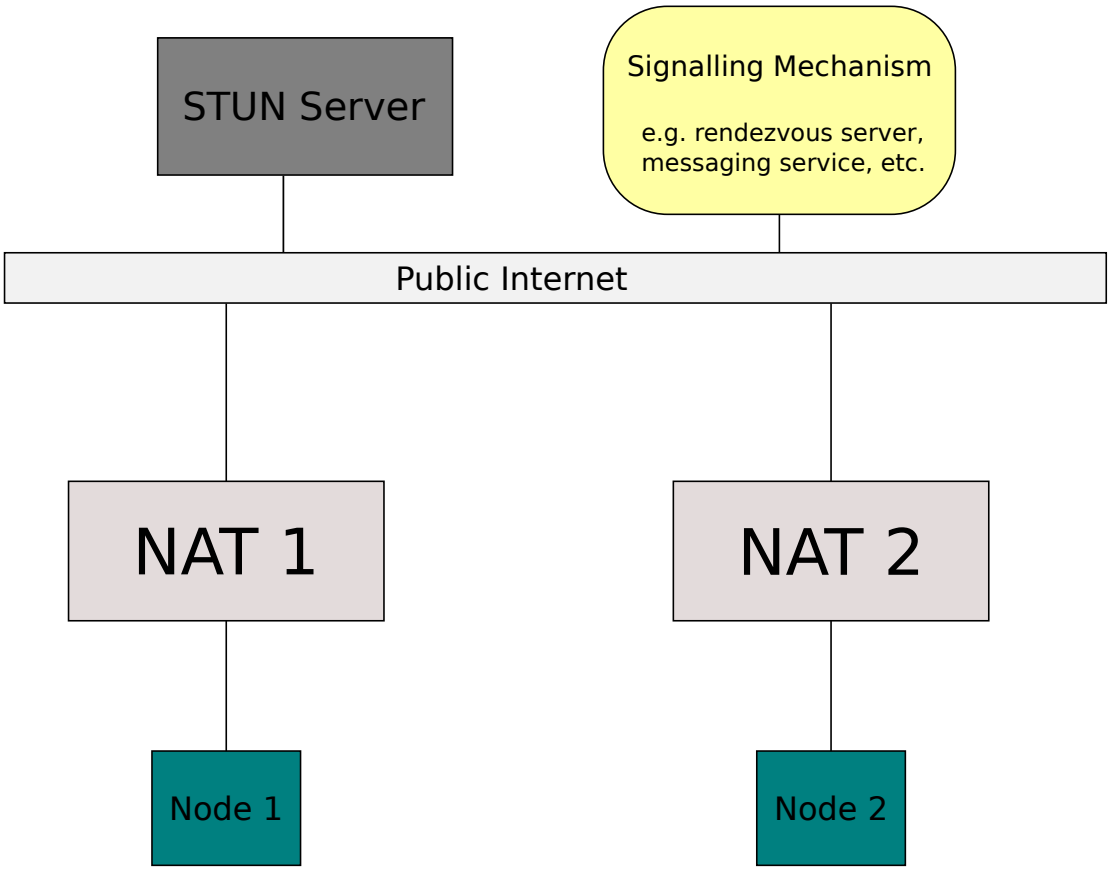


Figure 3.16: A simple network setup with a STUN server.

A more sophisticated use of STUN is employed by the Internet Connectivity Establishment (ICE) protocol [40]. Along with the TURN protocol (described in the next section), STUN is used for NAT discovery and connectivity checks. In some cases only STUN is used. For example, two nodes behind separate NATs can determine if there is a connection path between them using only STUN with

ICE extensions. They both prepare a list of candidates from both the local client addresses and the server reflexive addresses. One node takes on the controller role and sends an initial offer to the second using an Session Description Protocol (SDP) message, via an external server if both nodes are behind NATs. Since both nodes now have candidate lists for both their addresses and those of the node they are trying to contact, they begin connection checks using STUN binding requests. If any valid connections are found, those can then be used for direct communication.

In addition to allowing STUN clients to discover their external facing address, there are several optional STUN components. These include a fingerprint mechanism for use when STUN is multiplexed with other protocols, DNS discovery of other STUN servers, a mechanism to provide message integrity, and short and long term credential mechanisms [54]. These mechanisms are used by other protocols that make use of STUN, such as ICE.

The original incarnation of STUN was described as a complete NAT traversal solution, and was called Simple Traversal of UDP through NATs. It was also described as able to detect NAT configurations [41]. In some network configurations this version of STUN had problems working over TCP, leading to proposed solutions such as Simple Traversal of UDP Through NATs and TCP too (STUNT) and others discussed in [16] and [15]. STUNT attempts to achieve NAT connections via TCP by having the nodes predict their global addresses and use RAW sockets to read the full first SYN message. Some of the ideas present in STUNT have been incorporated into ICE over TCP (ICE-TCP) in order to allow ICE to work over TCP streams [27]. However, the success of these methods is still somewhat dependent on the involved NATs complying

with the requirements described in [45].

The ability to detect NAT configurations was omitted from the current version of STUN as not all NATs fit neatly into the categories recognized by STUN. While the current version of the STUN protocol is not a complete NAT traversal solution, it does provide a set of tools used for NAT discovery and traversal by other protocols such as ICE, SIP, and BEHAVE-NAT [54]. There is also research being done to extend the STUN protocol for more specialized uses. Some of these pursuits are detailed in Section 3.2.7.

Unlike NAT-PMP, PCP, or UPnP, STUN depends on an external server that is usually not part of the NAT device. This allows software applications to use STUN without being concerned about the configuration or number of the NAT devices on the network. When used as part of a larger system for NAT traversal such as ICE, knowledge about the architecture of the network is also not needed. While UPnP also involves a discovery mechanism, it is arguably more complex and depends on the NAT devices themselves. NAT-PMP and PCP do not have a discovery mechanism.

3.2.5 TURN (Traversal Using Relays around NAT)

While TURN was not evaluated for performance as part of this research, it is an extension of the STUN protocol and is described here for comparison.

General Operation TURN is an extension of STUN that allows nodes behind NATs to communicate regardless of the NAT architecture or configuration, as long as they are both able to contact the TURN server. It uses a client/server architecture, and the server is used to relay messages from the client to another

node [34]. However, as a result of having to relay messages between nodes, TURN is generally slower than STUN. In the base specification, TURN supports IPv4 addresses and the transport of messages from the client is supported over UDP, TCP, or TLS over TCP, while messages from the server to the contacted node, known as the "peer", are always sent over UDP. An extension to support TCP messages is described in [37] and another to support IPv6 addresses in [10]. Only a client initiating communication with another node needs to use this protocol; the contacted node simply sends and receives messages to and from the TURN server. The protocol offers several services to provide the relay functionality including allocations, permissions, and data channels. It also supports the reduction of IP fragmentation via packet size [34].

As TURN is an extension of STUN, its services supplement those offered by STUN by adding the following methods and attributes to the base STUN protocol.

Methods Allocate, Refresh, Send, Data, CreatePermission, ChannelBind

Attributes CHANNEL-NUMBER, LIFETIME, XOR-PEER-ADDRESS, DATA, XOR-RELAYED-ADDRESS, EVEN-PORT, REQUESTED-TRANSPORT, DONT-FRAGMENT, RESERVATION-TOKEN

TURN uses the concept of "allocations" to track relayed connection data, and these structures are the base association for the other concepts used. An allocation contains information such as the relayed transport address (used for server to node communication), the 5-tuple (client IP and port, server IP and port, and protocol), authentication information, and permissions, among others. Allocations are created via a STUN request/response sequence with additional

attributes. An allocation can also be refreshed by sending a request with a LIFETIME attribute of 0.

A "permission" is associated with an allocation, and each allocation can have multiple permissions associated with it. They contain both an IP address and a time-to-expiry, and allow the specified IP address outbound communication with the client. All permissions have an initial lifetime of 300 seconds, and can be created and refreshed either via CreatePermission requests or ChannelBind requests.

Once the allocations and permissions have been set up, data can be exchanged either though "Send" and "Data" indications, or using channels. The Send indication is meant to be used for data originating from the client and destined for the contacted peer. The Data indication is used for data originating from the peer and coming back to the client. Both of these indications use the Data attribute to store the actual data being communicated.

The other data exchange method is to use Channels, which require some initial set-up but involve less overhead for the individual messages. A channel is set up via a ChannelBind request from the client. Each binding is made up of a number for the channel, the transport address of the peer to contact, and the time-to-expiry. It is specific to an allocation. In addition to setting up channels, these requests also create or refresh a permission. Once the channel has been established, data is sent with ChannelData messages, which have the structure below.

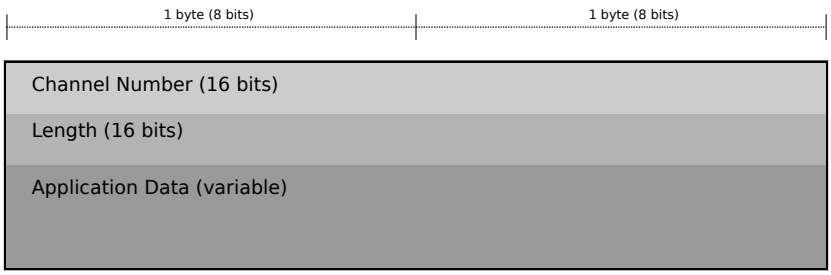


Figure 3.17: ChannelData Structure

An example of a basic exchange of data via a TURN server is shown below.

1. Client sends an Allocate request to the server.

2. Client receives an Allocate success response.

3. Client sends a Permission request to the server for a particular node, or "peer" with which to communicate.

4. Client receives a Permission success response.

5. Client sends a ChannelBind request to the server to establish a communication channel with the peer.

6. Client receives a ChannelBind success response.

7. The client now sends ChannelData messages to the peer and receives responses via the established channel.

8. The connection is refreshed by the client via a Refresh Request to the server.

9. The server responds with a Refresh success response and communication between the client and the peer continues.

While TURN works in all situations, since messages are relayed through a server, performance may suffer. This was also concluded by the authors of [36] during their review of NAT traversal methods. As an extension of STUN, TURN shares many of the same properties.

3.2.6 Summary of Comparison

NAT-PMP is the simplest of the protocols discussed, but as a result has the fewest features. It allows a client to explicitly map an external facing address and port using a user-specified external port number and a requested lifetime for the inbound mapping. In addition, it allows a client to obtain the external facing IP address from the NAT device using a separate request. PCP has a more complex message format and offers support for creating both inbound or outbound mappings, as well as a specialized message format for making announcements to clients. The lifetime component of the mapping request for NAT-PMP, PCP, and UPnP allows the client to reduce the number of keep-alive messages needed, as they can simply renew when the lifetime expires. This is especially useful with the outbound PEER mapping option offered by PCP. In addition, PCP includes multiple ways to restore the state of a NAT if it is powered off or reboots.

When compared to UPnP, NAT-PMP and PCP both have a more compact message format, since UPnP uses XML. UPnP also lacks a mechanism to recover from a NAT device reboot, although that can be mitigated by having the UPnP control points handle the recovery. NAT-PMP and PCP require a finite lifetime for the mapping while UPnP does not, although it can accept one if requested.

If a requested external facing port has already been mapped, NAT-PMP and PCP can assign an alternative port to the client. However, UPnP will either return a error or silently overwrite the existing mapping [29].

STUN is not designed to control the explicit mapping of ports, but it can be used to obtain an external facing address and port mapping via an outgoing binding request to a server outside of the NAT. Since TURN is essentially an extension of STUN that acts as a relay server, it allows communication in all situations, but potentially at the cost of performance.

If a node is behind a single NAT device, NAT-PMP, PCP, or UPnP would work equally well for simple address and port mapping. However, PCP includes several security-related enhancements as well as the PEER mapping option for outbound connections. These make it a better choice in situations where that type of mapping is required. The ability of NAT-PMP and PCP to automatically assign an alternate external facing port may make them a better choice in situations where the client does not know which ports may be available on the NAT device. On the other hand, the UPnP collection of protocols includes functionality other than NAT device port mapping, so applications that make use of other UPnP services (such as presentation device control) may want to take advantage of the accompanying port mapping. STUN and TURN are best for situations where the number or configuration of NAT devices is not known and the client needs to communicate with another node outside of its local NAT. The characteristics of each are summarized in Table 3.1 below.

Supported Capability	NAT-PMP	PCP	UPnP-IGD*	STUN	TURN
Control of NAT Device	Yes	Yes	Yes	No	No
Multiple NAT Layers	No	No	No	Yes	Yes
Requires External Server	No	No	No	Yes	Yes
Message Relaying	No	No	No	No	Yes
Mapping Type(s)	Inbound	Inbound, Outbound	Inbound	Auto (NAT Device)	Auto (NAT Device)
Mapping for other clients	No	Yes	Yes	No	No
Communication with Server	UDP	UDP	UDP, TCP	UDP, TCP, TCP-over-TLS	UDP, TCP, TCP-over-TLS
Mapping Protocols (NAT)	UDP, TCP	UDP, TCP, SCTP, DCCP	UDP, TCP	UDP, TCP	UDP, TCP
NAT Device Recovery	Seconds Since Start of Epoch	ANNOUNCE, Rapid Recovery	Via External UPnP Features	N/A	N/A

* These characteristics are based on the UPnP-IGD Specification Version 1 [53].

Table 3.1: Comparison of Protocol Characteristics

3.2.7 Specialized Traversal Methods

While the protocols mentioned in Section 3.2 are useful in most situations, research has been done to adapt these protocols to optimize NAT traversal in more specific situations or improve NAT traversal in general, especially in the realm of P2P applications. Many of these techniques either incorporate or are based on the original or current STUN protocol specification. Others involve modifications to the NAT device itself, or take a different approach entirely. While none these techniques were involved in the experiments for this thesis, they demonstrate some issues with the more standardized methods and propose some interesting solutions.

Techniques Based on STUN

In addition to being extended to create the TURN protocol, the flexibility and extensibility of the STUN protocol has led to its use for many applications, which is one of the reasons it was chosen as the basis for the algorithm described in Chapter 4. This section describes modifications to facilitate working in a P2P environment. While many of these are based on the legacy STUN specification [41], the work in still relevant in the context of the newer protocols

predominantly in use today.

One of these techniques makes use of the concept of user "superpeers", described in [51] by Wacker et al. With this technique STUN is used as a component of a larger process to allow communication between nodes that may or may not be behind a NAT. As a prerequisite, all of the nodes involved must be joined to an overlay network with a Distributed Hash Table (DHT) to use as storage for node addresses. The network also provides "multi hop" routing between nodes. First, all of the nodes obtain their NAT configuration using the legacy STUN protocol. This can be done either with external servers or by "superpeers" within the network, which are STUN servers formed at runtime from two nodes that are either behind a Full-Cone NAT or no NAT. Nodes on the network use the DHT to find a suitable STUN server. Once a node knows its NAT type, it chooses a suitable communication method based on that type and that of the node being contacted. The traversal methods chosen are either direct, hole punching, reversal, or relaying. The initial coordination for the communication setup is done via the overlay network.

The authors noted that they plan to continue the work and evaluate performance. Even though the current STUN specification no longer offers the ability to detect the NAT configuration, this system could be used in situations where the NAT configuration is known for at least some of the nodes, or where another method is used to learn that information. For example, NAT device information such as supported protocols, NAT mapping behavior, and NAT filtering behavior could be determined by methods such as those described in [21]. Another possibility would be to adapt the ICE protocol [40] to set up the traversal method. The authors also mention that without UDP support this

system will not work, but the current STUN specification works over TCP as well.

Another approach to adopting STUN to work in a P2P environment is called C_STUN, and is described in [57]. An external STUN server is required. The network is divided into areas, or "communities", each containing one or more NAT devices. The authors define a new STUN attribute called COMMUNITY, which is used to indicate to which area of the network a node belongs. This new attribute is included with normal binding requests and responses used by the client node to receive its external IP address and port.

The STUN server maintains a "watchlist" of nodes based on their communities. The first node in each list is called a "supernode" and is used to connect to other nodes in that community via UDP. With this setup two nodes within the same community can connect to each other via the supernodes. The list is managed dynamically and if a supernode is detected to no longer be "alive", the next node in the list becomes the supernode. The server also manages the list for efficiency by tracking requests and moving often requested nodes up the list. For nodes in different communities, hole punching is used to establish a direct connection between the nodes via the STUN server.

As with the previously discussed technique, this paper is based on the legacy STUN specification. However, this technique only uses the STUN functionality that is still available with current specification [54]. Therefore it would not need additional support if the current specification was used instead. If knowledge of a node's NAT type is desired to facilitate setting up a method of communication either via supernodes or hole punching, techniques such as those discussed for the previous research can be used. The current STUN specification's support

of TCP also opens up additional options. The authors note that it would be more efficient to use a DHT instead of a watchlist, as is used by the previously discussed technique.

Techniques with NAT Device Modifications

Another approach is to enhance the NAT devices themselves to support the required functionality. One example is a dynamic mapping method geared toward heterogeneous networks, which is described in [52]. These modifications allow the NAT devices to store the address mappings in a P2P storage system that is mirrored on other nodes in the network. A node can find the address of any other by contacting its NAT, which will return the address information. The architecture is comprised of NAT devices in the "core" network as well as NAT devices on the border of each subnetwork. The authors describe this method as advantageous since it does not require an external server, there is no single point of failure, and there are no redundant communication routes, which minimizes any effects on performance. It also supports communication over TCP. While the authors' criticisms of STUN are based on the legacy specification, they make several points which still apply to the current specification, such as the dependence on external servers and the inability to work through symmetric NATs. They also compare their method to other NAT traversal solutions, such as TURN, ICE, AVES, and 4+4.

The authors of [56] propose router support for a "UDP Switch", which aids in communication among devices behind NATs. Like many other techniques used for NAT traversal, this one requires the use of an external server; in this case a SIP Server. Each node publishes the address of its NAT via a REGISTER

command to the SIP Server. When a node wants to contact another, it obtains the address of the corresponding NAT device via the server. The authors also include a new SIP parameter "paddr" to the SIP INVITE message, which tells the SIP server to return the NAT address of invited node. A new content type, "application/x-udp-switch", is added to SIP INFO messages as well.

Their protocol is used for communication through the UDP Switch enhanced NAT devices. It consists of four commands, using a Session ID to manage the switch sessions and Direction Indentifier to indicate from which node a data message originated. As is the case with most scenarios involving NAT mappings, once a connection is established, keep-alive messages must be sent between the two nodes to maintain the connection.

In [50], the authors are attempting to solve a different, but related problem than most of the other literature discussed in this section. While the majority of the discussed techniques focus on traversing NATs, this technique is presented as an alternative to NATs all together. The authors describe and evaluate "4+4" as a solution to the problem of insufficient numbers of unique IPv4 addresses. The authors state that a big issue with IPv6 will be the transition, while 4+4 allows for an easy transition and will make traditional NAT architectures unnecessary.

The core idea is the 4+4 address, which is the concatenation of a public and a private IPv4 address. This idea seems similar to the one presented in [30] (discussed in the next section), but taken a step further by eliminating the concept of the NAT. The IP packets contain two fields each for the source and destination addresses. The data format is set up so that the outer headers will contain addresses understood by normal IPv4 routers, and as a result packets can be forwarded correctly. A network is divided into "realms", which are groups

of networks that use the same address block with unique addresses. These are further divided into public-address realms and private-address realms.

Modified NAT devices, known as "realm gateways" handle the translation and routing of messages. For devices inside a private-address realm, the 4+4 address consists of the public address of the realm gateway and a local address of the node. In terms of transitioning to 4+4, most NAT devices will only need to be modified to swap addresses in the 4+4 header and convert ICMP (v4) message headers. The authors discuss several advantages of 4+4 over IPv6, including the ease transitioning in an ad-hoc manner, backwards compatibility with IPv4, and the preservation of subnet isolation. The latter is particularly interesting as the concept of a NAT does not need to exist in a 4+4 network, as the realm gateways can use the 4+4 address to route a message anywhere.

The authors also evaluated an implementation testing several common protocols and existing network analysis tools, and determined that while there are some issues, 4+4 could be a viable alternative to IPv6 if one is needed. While this paper was published in 2003, and a need does not seem to have yet presented itself, the idea has been shown to be experimentally solid. The ideas could also be adapted to NAT traversal in current environments.

The NAT-f (NAT-free) protocol described in [49] does not require a special server, but does make use of a Dynamic DNS (DDNS) server for name resolution using wildcards. DDNS servers are already commonly deployed in networks. The protocol is implemented both in the home gateways (NAT devices) and the "external nodes" of a network. It works over both UDP and TCP since NAT devices map them separately. The process described in the literature is based on a node outside of a NAT trying to contact a node behind one, but if the

functionality in the external node is instead implemented as part of the home gateway, the technique will work even with both nodes located behind NATs. The technique requires each home gateway to register itself with a domain name on the DDNS.

There are several data structures involved in the use of this protocol. Each home gateway contains an Access Control Table (ACT) containing the name, internal IP address, and access control flag of all internal nodes. External nodes contain a Name Relation Table (NRT), which stores the names of internal nodes and their associated home gateways. In addition, external nodes contain a Virtual Address Translation Table (VAT), which maps the virtual IP address to the actual mapped address in a particular home gateway.

When an external node wants to contact a node behind a NAT, the first step is DNS name resolution, with the external node saving the information in its NRT. It then attempts to create an entry in the VAT if one does not exist by sending a NAT-f mapping request to the internal node's associated home gateway. The home gateway checks its ACT and if an entry corresponding to the requested node exists and allows external connections (via the access control flag), the gateway creates a NAT mapping and sends the appropriate information back to the external node that issued the request.

One notable aspect of this technique is that it works with Symmetric NAT configurations since the NAT-f process is triggered each time the address information changes. It also allows communication with devices that do not support NAT-f. According to the authors it does not significantly affect the performance of communication in the cases tested with their implementation in the FreeBSD kernel, especially relative to other NAT traversal techniques.

Other Techniques

Other research focuses on solutions that make use of an external server or a P2P network overlay to achieve NAT traversal. For example, the authors of [35] describe a system called Autonomous NAT Traversal. They cite security risks and added complexity as reasons for not using an external server, instead relying on the mapping behavior of most NAT devices.

The only prerequisite for this technique is that the node (Node A) trying to contact another node behind a NAT (Node B) must know the public address of the NAT device for Node B. Node B is configured to regularly send either an ICMP ECHO REQUEST or a UDP message to an unallocated IP address at a set time interval. Since the request came from behind the NAT, an external facing mapping is created and the NAT device will allow incoming responses to the message. Node A can then fake an ICMP TTL_EXPIRED reply, which is routed by the NAT to Node B due to the mapping. Node B can obtain the IP address of Node A from that reply and establish a UDP or TCP connection.

The authors provide three implementations of this technique, which were evaluated in a variety of network configurations by volunteers. They determined that the technique works almost all of the time in configurations where only one node is behind a NAT, but works very rarely if both nodes are behind a NAT. Despite this limitation, the authors note that most applications that utilize NAT traversal have several techniques at their disposal, and this one can be included as part of the collection and used when applicable. This could potentially be attempted along with other protocols in a similar manner to how the ICE protocol utilizes STUN and TURN, and only used when it is a viable choice for

the NAT configuration. The authors discuss the advantages and disadvantages of using ICMP messages verses UDP, as well as the various methods attempted for connection if both nodes are behind a NAT.

The authors of [30] describe a NAT traversal technique based on hole punching as part an a P2P eLearning system they are developing. The technique requires no external servers, and unlike many other techniques discussed, all of the nodes in the network are equal. However, it is only effective when each node is behind only one NAT device. Each node has a unique identifier consisting of either its global IP address and port; or if it is behind a NAT, both its external IP address and port and private IP address and port. When a node joins the network, it creates a mapping on its NAT device either using UPnP (if available), or manually. To join the network, a node must know the IP address for an existing member. Once part of the network, data is shared between the nodes.

A structured P2P NAT traversal technique known as SMBR (Selective-Message Buddy Relaying) is introduced in [55]. A DHT is used to keep a list of mappings, similar to other methods discussed previously such as [51], or the name relation table in [49]. The protocol has message types for several scenarios. Control messages are used to establish communication with other nodes. The connection process depends on the network configuration for each node, and in some cases a third node can be used to help initiate a connection between two nodes. Random selection is used when selecting helper, or "buddy" nodes.

The authors of [42] describe a method of using nodes as NAT proxies for connections in P2P games. This technique requires an external server, called

the "global tracker", which maintains global identifiers for each node. The system consists of two layers, Application Layer Framing (ALF) and Peer to Peer Abstraction Layer (PAL). The PAL layer is responsible for creating proxied connections when necessary.

The system first attempts NAT traversal via hole punching, similar to previously discussed techniques and to those used with STUNT. The hole punching is attempted using information from the global tracker, a "buddy server", and port prediction. It will fall back to creating a proxy connection using another node (or server if absolutely necessary) if the NAT hole punching is not successful. The proxy technique works for both UDP and TCP traffic. Proxy nodes are found based on a set of five nodes from a random selection. The authors point out the difficulties of a proxy node leaving the network and offer some mitigation techniques. Their evaluation concludes that the performance of this method is satisfactory, but can be improved by further research.

Much research in the area of NAT traversal focuses primarily on UDP, since a common use case is P2P applications such as Voice-over-IP (VoIP). In [12], the authors propose a NAT traversal technique designed to solve the NAT traversal problem for TCP, although the solution will potentially work for other transport protocols as well. It uses UDP hole punching and IPv4 tunnels to wrap any transport protocol (in this case TCP using IPv6 addresses) and establish a connection between two nodes. All of the traffic of the original protocol is packaged in UDP packets and the segments are sent through the tunnel. While it does not explicitly require an external server, it does require that some nodes not be behind a NAT, in this case to act in a rendezvous capacity when setting up a connection. The two types of nodes are the "rendezvous" nodes and the

"client" nodes. The authors state that an overlay network of rendezvous nodes is the basis of the system. The rendezvous nodes can be compared to the supernodes described in [57]. While the supernodes simply act as links between various nodes in a part of a network, like the rendezvous nodes they are essential for making the NAT traversal technique effective. The superpeers described in [51] are also similar in that they provide an essential service to the other nodes, in that case supplying their external address information.

A client node begins its connection to the network by logging in to a rendezvous node. This technique assumes that all clients have knowledge of addresses for stable rendezvous nodes. As with previously discussed techniques, once the login connection is established, keep-alive messages must be sent periodically in order to keep the connection active. The rendezvous node sends the client node an IPv6 address that was chosen for it.

The technique depends on a virtual network interface created by the client node for the tunnel, from and to which normal network traffic is forwarded. The rendevzous nodes keep a table of all active client nodes, as well as tables of other rendezvous nodes it is logged into and those logged into it. A timeout value in the table is used to remove inactive sessions and avoid wasting memory. Because rendezvous nodes are in contact with each other and maintain lists of client nodes, they are able to assist in the UDP hole punching needed for the connections. Experimentation showed that there is little effect on performance with no modifications to existing software or NAT devices required. For future work the authors look to include a DHT to improve the performance of client nodes looking for rendezvous nodes. This is one of the few pieces of literature reviewed that explicitly stated its goal to work with NATs even after IPv6 is in

wide use. Many papers, particularly ones published before 2005, assume that once IPv6 is in wide use there will be little use for NATs.

3.3 Related Work

To our knowledge there is little research in terms of comparing these protocols in terms of performance. Comparisons based on other factors are described in the previous sections.

As part of their research for creating guidelines for managing SIP-based services, the authors of [11] evaluated the STUN, UPnP, and IPFreedom protocols for both the number of messages sent and time based performance. The former two protocols are discussed in Sections 3.2.4 and 3.2.3, respectively. Like STUN, the IPFreedom protocol uses a server on the outside of the NAT. The server is used to send signal messages via TCP tunneling, while media related data is sent with UDP.

The Vocal software was used as a SIP proxy, and Ethereal was used to obtain network traffic data. In terms of clients, the experiment tested Grandstream Budgetone, Windows XP Messenger, and Wave3. The SIP messages used for analysis were REGISTER, INVITE, and BYE. STUN was found to be the most efficient, generating the fewest extra messages (although still a significant amount) and having the smallest performance impact. UPnP and IPFreedom both generated many more extra messages and affected the performance in several of the "public to private" communication scenarios. The authors point out that while STUN is the most efficient, it does not work in all NAT situations.

As part of their evaluation for their SPM peer communication system, the

authors of [36] evaluated the efficiency of NAT traversal in their system, which uses ICE. The comparison was done against the iperf benchmark using UDP communication over routers that were manually configured to forward the necessary ports. Their experiment spanned four sites and eleven network topology setups. In all cases the nodes were able to connect using SPM and NAT hole punching, and were within 87% network efficiency in terms of successful traffic.

3.4 Objectives

The objective of this experiment is to determine which of the following methods (if any) are the best in terms of performance for obtaining an external address and port mapping through a NAT: UPnP, NAT-PMP, PCP, and STUN.

3.5 Motivation

In terms of packet formats, NAT-PMP, PCP, and STUN are fundamentally different than UPnP. The first three protocols use an efficient, well defined binary packet format and have maximum message sizes. This allows for less data transmitted over the network, as well as potentially easier processing by the client and server. They are also transmitted over UDP, which has less overhead than TCP since the connection state is not tracked.

UPnP control messages are transmitted via HTTP requests and responses over TCP. The requests themselves are XML, and have no defined maximum size. While this does not necessarily mean the messages will take longer to process, it is a possibility.

3.6 Tools

For this experiment the following equipment and software was used:

- Clients running with Java 1.7 on Slackware Linux 14.1
- STUN Server running with Java 1.7 on Windows 8.1
- ASUS RT-N16 router with ASUSWRT-MERLIN firmware, using miniupnpd for UPnP, NAT-PMP, and PCP

Since UPnP, NAT-PMP, and PCP are used for actually creating a port mapping, the client interfaces directly with the NAT device for the test. When using STUN, the automated functionality of the NAT device is used to create the port mapping. A STUN server outside of the NAT completes the process by providing the client with its external facing address and port.

3.7 Experimental Methods

3.7.1 Assumptions

- The network configuration is constant for each run for all of the protocols tested.
- The NAT device is under a similar processing load for each run for all of the protocols tested.

3.7.2 Design

There are two parts to this experiment. When testing UPnP, it was observed that if the process was repeated within thirty seconds of finishing a previous iteration, the resultant time was significantly smaller, even when requesting a mapping for a different port. When the client pauses for thirty seconds or more, the times are more stable. Although the client removes the mapping and closes the socket used for the connection, the operating system may keep resources open for reuse at a lower level. It is also possible that the UPnP server caches the slot for a specified period of time. Initial tests with the other protocols showed much smaller times that seemed to be unaffected by the lack of a pause. However, for the sake of consistency, the clients for all of the protocols pause for thirty seconds after completing an iteration of the test before beginning the next one.

Figure 3.18 shows the network architecture used for these experiments. In all cases except STUN, the client interacted directly with the NAT device to create the external address and port mappings. For STUN, the external STUN server was used to obtain information about the external facing address and port mapping automatically created by the NAT device.

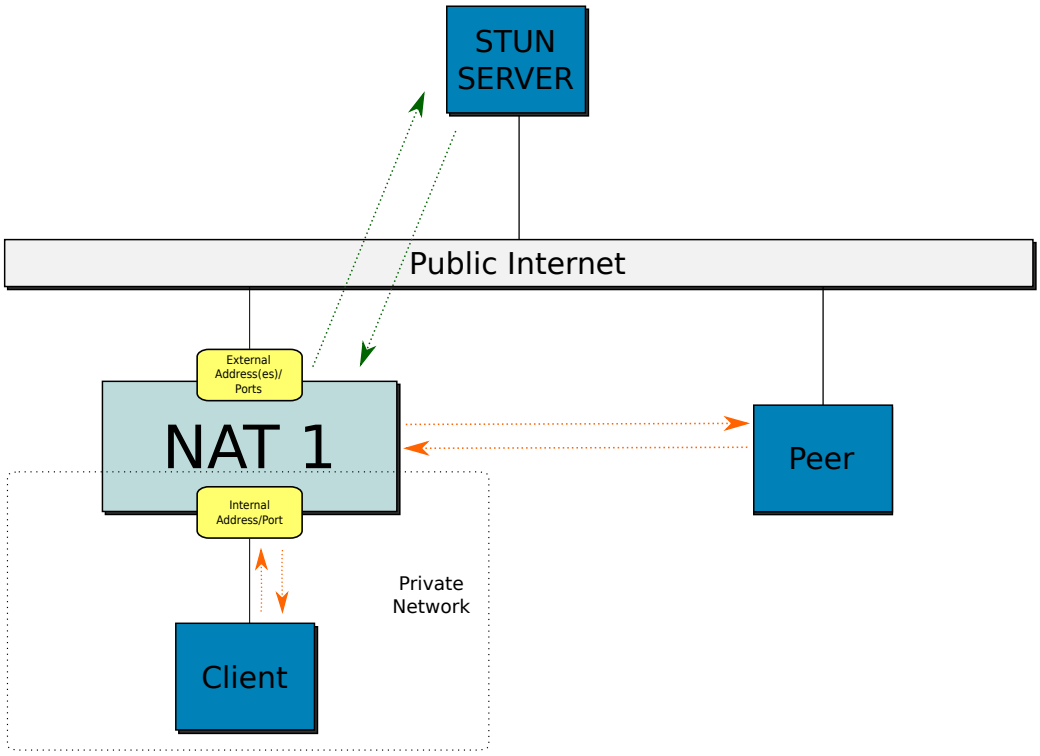


Figure 3.18: Protocol Test Experiment Architecture

Local Tests

The first part of the experiment involves all of the protocols and is as follows:

1. Client sends a request to the NAT device or server to create the mapping.

2. Client receives a response. With PCP and STUN this is the only interaction with the NAT device or server required to create the mapping.

3. In the case of NAT-PMP, the client makes another request to get the external address, and with UPnP an extra request is needed to get both the address and port for verification.

4. Client sends a request to a pre-determined peer address outside of the

NATs local network that includes its new external facing address and port.

5. Client receives a response from the peer to confirm that the NAT address and port mapping is active. In the case of STUN this simulates a UDP hole punching scenario where the external address and port of the other node is already known.

6. Client sends a request to the server to destroy the mapping (for all protocols except STUN).

The experiment was performed 1000 times for each protocol.

Geographically Dispersed Tests

The second part involves only STUN. Since UPnP, NAT-PMP, and PCP directly interact with the NAT device to actually create the external address and port mapping, their only logical use is from the machine directly connected to the NAT device. However, STUN is meant to be used in a variety of network configurations. In order to get a broader view of the performance, the STUN test was also done on a machine in a different location to test the effect of network latency on the performance of the protocol. The STUN client was located on a machine in Austria, while the STUN server and the peer used for the test ping were located in Melbourne, FL.

3.7.3 Measurement

The timing is done on the scale of milliseconds, using Java's System.currentTimeMilliseconds() method. Each client records the time di-

rectly before sending the initial request to the NAT device or server and directly after receiving a response. They once again record the time directly before sending the request to the external peer and directly after receiving a response. In both cases the two times are subtracted to find the time interval.

3.7.4 Threats to Validity

The primary validity concern with this experiment is that the results are implementation dependent. However, the objective is primarily to get a general idea of the performance for each. For UPnP, NAT-PMP, and PCP, the widely used MiniUPnP daemon acts as the server and is part of the router firmware. The STUN server is custom built for the experiment, but processes the requests in a straightforward manner.

3.8 Analysis of Results

The mean, median, and mode of the response times for each protocol are shown below. The response times were as expected, with UPnP being slightly larger possibly due to the use of TCP.

3.9 Discussion and Conclusions

As expected, all of the protocols provide very quick response times. NAT-PMP and PCP are comparable, with STUN having a slightly slower average time. However, this may be due to needing to traverse to an external server and back through the NAT device instead of only using the NAT device. UPnP was

Protocol	Mean Response Time	Median Response Time	Mode Response Time
UPnP	78	57	43
NAT-PMP	3	3	3
PCP	4	4	4
STUN (local)	5	3	3
STUN (distant)	209	186	172

(a) NAT Mapping Only

Protocol	Mean Response Time	Median Response Time	Mode Response Time
UPnP	88	68	65
NAT-PMP	13	11	10
PCP	14	12	11
STUN (local)	16	11	10
STUN (distant)	424	387	373

(b) Including Peer Ping

Table 3.2: Protocol Response Times (ms)

slower, but this is most likely due to its use of the TCP protocol for handling address and port mapping requests. NAT-PMP, PCP, and STUN use UDP for request and response traffic. STUN was also tested with a geographically distant server to explore the effects of network latency on the protocol's performance. The STUN server and "peer" for pinging were located in Melbourne, FL. The STUN client was run from Austria. This resulted in longer response times when making binding requests, but was still very fast.

The performance of all of these protocols when creating an external address and port mapping is comparable. When deciding on which protocol to use, it appears to be best to instead focus on the features provided by each (discussed in previous sections).

Chapter 4

NAT UDP Keep-Alive Interval Optimization

4.1 Introduction

There are numerous situations where a client application behind a NAT device needs to keep an external facing address and port mapping active, even when there is no data to be sent or received for an extended period of time. This is especially true in the case of P2P and VoIP applications, where the client depends on other nodes knowing their address and port so that they can be contacted. Several methods exist within various protocols to handle the sending and reception of keep-alive messages.

While some protocols have recommended keep-alive interval times, research has been done to calculate the "optimium" keep-alive interval for a given client's environment. Ideally a minimum number of keep-alive messages will be sent, both to reduce network traffic and to reduce the processing load of the clients

and servers involved in the communication. The sections below explore existing methods for sending and receiving keep-alive messages, describe existing research to calculate the keep-alive interval, and introduce a new algorithm for calculating a keep-alive interval while offering comparisons to existing methods.

4.2 Background

There are currently several techniques used by applications when sending "keep-alive" messages, which in this context are messages whose purpose is solely to keep the NAT external facing address and port mappings active for the client node. Many NAT devices will terminate a mapping if it becomes inactive after a certain amount of time, which will vary based on the device and configuration. While many of these methods include recommended values for various protocols, they all offer a way for the client to provide their own preferred keep-alive interval value or at least note that it should be configurable.

In their analysis of VoIP systems energy efficiency, the authors of [5] discuss the necessity of NAT keep-alive messages, as well as their being one of the major drains of power or "waste", along with signaling and media relaying. They discuss various methods used by applications to maintain NAT external facing address and port mappings, such as SIP NOTIFY requests. It is also noted that while these messages are expensive, without knowledge of the network architecture there may not be a better way to keep the NAT mapping alive. In their experiments, the authors used three network configurations, the first two of which used UDP; the first with NAT traversal and the second without. In the second configuration, due to the absence of keep-alive message traffic,

the server could handle twice as many users as the first configuration. In [17], the authors discuss various origins of energy consumption in mobile devices, including keep-alive messages. They also discuss how many protocols can be wrapped and sent over UDP, so longer UDP mapping lifetimes are becoming more important. The authors used a Nokia 6630 phone and measured both the transient current and the cumulative electric charge while using the phone on various networks. They were able to measure the energy consumption of a single keep-alive message, and concluded that keep-alive messages were a significant factor in the energy consumption of the device. They suggest that network operators can aid in preserving the battery life of devices using the network by setting the NAT and firewall configurations to have reasonable timeout values.

The authors of [13] use a simplified version of the STUN protocol along with "superpeers", STUN servers, and "timeout" servers to assess various attributes of the network configurations of nodes on a P2P network. They collected data over a month long period, using a sample of 3500 peer nodes. Their experiments determined that 90% of the nodes tested were behind some type of NAT or firewall. They also found that out of their sample, 62% of the timeout values were between two and two and a half minutes, and 25% were between one minute and two minutes. A significant number of nodes were also found to be behind a NAT configuration that makes P2P communication difficult. They concluded that there is a growing number of nodes behind NATs and firewalls that are participating in P2P activity, but, as stated earlier, much of the infrasture is still not P2P friendly.

In [31], the authors discuss keep-alive messages as a challenge to their new NetServ node architecture and propose a "Nat KeepAlive Responder" module

to handle the messages. The module would be set up on NetServ routers and respond to keep-alive requests on behalf of the SIP server by spoofing the server's IP address in the response. This would at least alleviate the stress on the server from many users constantly issuing keep-alive requests. This idea is similar to one presented in [5] where the SIP user agents are embedded in routers. However, with that solution the purpose is to bypass the need for keep-alive messages all together by taking advantage of the router's known public address.

The author of the patent [19] describes a system to manage keep-alive messages for wireless devices based on one or more servers and "network boundary components", or NAT devices. The motivation appears to be saving battery power in the mobile devices by minimizing their involvement in the keep-alive process. Two implementations are described. In both cases, keep-alive messages are sent from the server on a set interval to the NAT device. In the first implementation, the NAT device forwards the keep-alive message to the client, and the client only contacts the NAT device if it does not receive an expected message. The NAT device can then resend the message or reset the session depending on the configuration. The second implementation involves an enhanced NAT mechanism and does not require any communication from the client. In that case the packets are not transmitted past the NAT device, and it handles the session renewal without contacting the client. This is achieved via a Time-To-Live (TTL) attribute on the keep-alive packet that is calculated to only be large enough to reach the appropriate NAT device. Since the client is not involved, keep-alive messages can be sent more frequently, creating a more "robust" connection.

There are various methods of sending keep-alive messages depending on the

protocols used by the application. In [26], both a Carriage Return and Line Feed (CRLF) (for TCP) and a STUN (for UDP) keep-alive technique are discussed in the context of supporting SIP messages. The core of the technique is a ping from the client to the server to keep the NAT binding active. Instead of attempting to find an interval to use for sending the keep-alive messages, default intervals are suggested. The authors also state that clients or servers may want to do their own interpolation of the interval. With the CRLF technique, the client sends double CRLF messages at 10 second intervals and the server responds with a single CRLF response. The STUN method consists of Binding Requests and Binding Responses, with messages from the client sent at random intervals between 24 and 29 seconds. In lieu of the default values, the client can supply their own timeout interval via the Flow-Timer header in a SIP registration message.

Another keep-alive technique using SIP messsages is described in [22]. Unlike the previously discussed method, this includes a new SIP parameter, "keep". This parameter indicates a willingness to receive keep-alive messages and provides a recommended interval value. Keep-alive messages can either be associated with a SIP registration or with a dialog. In the first case, the messages are sent until the registration is ended. In the latter case, the lifetime is either negotiated up front or lasts until the dialog ends. These keep-alive messages can either be CRLF or STUN, depending on the transport protocol used.

The "keep" SIP parameter used to define the keep-alive messages in this context contains a place for a recommended frequency for the messages (interval value), measured in seconds. The interval between each message should be randomly distributed between 80% and 100% of the recommended value. How-

ever, this recommended value is not required, and the frequency determination can fall back to the methods discussed in [26]. The "keep" parameter is more versatile than using the previously mentioned Flow-Timer header, but in the case where they are both used, the specification states that the same value be used for both.

The subject of keep-alive messages is also discussed in [40]. The authors discuss various keep-alive message formats based on the peer type such as STUN indications, Real-time Transport Protocol (RTP) No-ops, error messages, or some other means. The recommendation is that the frequency should be configurable, but have a default of 15 seconds. There is also a recommendation to have the frequency set to the largest value permitted by the network setup. In [25], as part of an outline of behaviors that NAT devices should exhibit when dealing with UDP traffic, the authors state that for general use, an external address and port mapping lifetime must last at least two minutes, and a lifetime of more than five minutes is recommended. As seen by the recommended keep-alive intervals from other publications and the results from [13], many NAT devices have shorter mapping lifetimes by default.

In [33], the authors discuss various methods for sending keep-alive messages for applications using RTP. These include empty transport packets, STUN indications, and RTP messages with invalid or empty data fields that are ignored by the receiver. However, the authors point out that RTP implementations do not always ignore messages as they should per the standard, so the latter method may cause issues. The recommended method is to multiplex RTP packets with RTP Control Protocol (RTCP) packets, as this reduces the number of ports needed and in some cases the keep-alive functionality can be handled

with RTCP. In terms of the keep-alive interval, the recommendation is 15 seconds for UDP and for TCP the recommended interval is 7200 seconds, with the understanding that the values should be configurable.

4.3 Related Work

4.3.1 Introduction

In addition to the different methods of handling keep-alive messages, research has been done to find the optimum interval to use for sending keep-alive messages. This is due to the drawbacks involved with keep-alive messages, such as the resources used to process the messages on the server and considerations such as battery life on the client.

Many factors add to the difficulty of calculating a keep-alive inteval for a particular network configuration. In addition to the different types of possible NAT configurations discussed in Section 2.1.1, the actual external address and port mapping lifetimes for different NAT devices can vary considerably. The study described in [24] shows the results from testing the behavior of several routers and firmware versions, including the NAT behavior. The experiments were performed by having each test node connected both through the NAT device and through a "management link" to help coordinate message sending and data collection for measurements.

For mapping over UDP connections, the authors tested the mapping lifetime in the following cases: when only one message is sent from the client with no reply, when the server responds with multiple messages, and when there are mul-

tiple back-and-forth messages between the client and server. They determined that many routers modify or renew the mapping lifetime for a connection based on both incoming messages and outgoing messages. In most cases the lifetime resulting from the inbound server messages is longer than the initial lifetime from the single client ping, although in other cases it was shorter. When there are multiple back-and-forth messages, some routers extend the lifetime even more upon renewal while most renew it by the same length as they did with the first server response. The actual length of the lifetime varies by router. The authors state that due to the wide range of behaviors and the large percentage of routers that do not adhere to Internet Engineering Task Force (IETF) recommendations, there is not a set of devices that can be targeted by developers as being "better" behaved. These variations in behaviors could be due to a combination of different default settings and the NAT device internally handling UDP connections using various states, such as UNREPLIED and ASSURED as described in [3].

These inconsistencies make determining a usable keep-alive interval difficult, especially for mobile devices that may change networks frequently. The techniques discussed in this section strive to address these issues. The new technique described in Section 4.4 builds on these existing ideas to determine an optimum keep-alive interval in a variety of situations.

4.3.2 Necessity of Keep-Alive Messages

Keep-Alive messages are a necessity in many situations, due in part to the variations with NAT mapping behavior discussed in the previous section. Connecting

to another node on a network through one or more NAT devices can be problematic and time consuming, especially for P2P applications, as is described in Section 2.2.

The common use case for keep-alive messages are situations where the architecture of the network is either not known, or where the client is on a mobile device that changes networks frequently. In these cases connecting to another node can be expensive in terms of both network traffic and time. The general approach in these situations is to use a variation of ICE or a less formalized set of steps to attempt a connection to the other node, which can result in wasted messages and lost time during the attempted connection.

When an application needs to exchange data with another node at a slower rate than the NAT mapping lifetime allows, these long connection attempts become a problem. For example, if an application experiences periods of time where it only sends messages once every few minutes, but the mapping lifetime of a NAT device along the network path is only 30 seconds, the node sending the message will need to re-establish the connection to the recipient node each time it sends a message.

In this case, sending a keep-alive message through the connection to prevent a timeout once it has been established both prevents a node from being inaccessible and allows application data to flow through the connection when necessary. Whether or not the cost of maintaining the keep-alive message traffic outweighs the inconvenience of continually re-establishing the connection depends on the application.

In situations where a node only interacts with a single NAT device, one of the previously discussed protocols for explicitly establishing mappings directly

with the NAT device is a better solution (NAT-PMP, PCP, or UPnP). With the functionality offered by these protocols, the node can specify the lifetime of the mapping and therefore send renewal requests at a rate more fitting for the situation instead of having to work around the automatic mapping lifetime of the NAT device.

4.3.3 Existing Keep-Alive Optimization Techniques

One area in which this research is particularly active is in P2P applications. The authors of [39] describe and evaluate three algorithms to help extend the length of keep-alive messages in a P2P network based on the amount of time each node has been in the network. In the context of their research they are using keep-alive messages to determine whether or not nodes are still part of the network, however the ideas can be used to help with address and port mappings in NATs in P2P networks, potentially both by the server and the client. They cite research stating that nodes which have been on the network for a longer period of time are more likely to remain on the network ([47], [48], [9]).

The first algorithm is the Probabilistic Keep-Alive Algorithm. It uses a fixed value for the interval, but each of the node's connections to other nodes is handled independently. For each connection, after each period of the interval ends, a keep-alive message may or may not be sent based on the probability that the node is still online, which in turn is based on the time the node has been online and the time it was last observed. This results in fewer keep-alive messages on the network overall. The second algorithm is the Predictive Keep-Alive Algorithm. It gradually increases the keep-alive interval for each

connection based on the amount of time the associated node is on the network. The third algorithm, the Probabilistic Keep-Alive Algorithm With a Budget, keeps a "budget" of bandwidth to use for keep-alive messages. The node sets the keep-alive interval for each connection from that bandwidth allotment based on the time each connected node has been on the network, with nodes that have been on the network longer getting less bandwidth.

The authors evaluated these algorithms using real P2P data from RedHat9 and LegalTorrents. Overall these methods risk increasing the maximum delay of failure detection due to fewer keep-alive messages, but decrease the mean and median delay. These methods could also be used for calculating the keep-alive interval for normal use between two nodes, especially in situations where one of the nodes involved is part of a complex NAT configuration where the needed keep-alive value could change based on external factors such as network traffic.

Two techniques for optimizing the keep-alive interval are described in [38] and are expanded on in the patents described later in this section. The first is the iGlance algorithm, which was used as part of the iGlance VoIP application. The original source for the iGlance algorithm could not be located. With this method two connections are used: one for normal traffic that uses a known "safe" keep-alive value, and a test connection to evaluate potential new keep-alive values. For each keep-alive message sent on the test connection, if a response is received, the interval between messages is doubled, and the normal connection keep-alive interval is set to the old interval used by the test connection. If no response is received, the interval between keep-alive messages is halved. This allows the live stream to always use a safe interval value and at the same time reduce the amount of network traffic resulting from keep-alive messages.

The authors improve on the basic iGlance technique by checking both the normal and test connections for failures and introducing a binary search element. As with iGlance, a normal and a test connection are used. Instead of only sending a keep-alive message on the test connection during the test interval, the message is sent on the normal connection as well. Testing both connections for failure allows the application to determine if there is another network issue or if the receiver is no longer on the network. While a failure on only the test connection indicates a NAT mapping timeout, a failure on both connections potentially indicates one of the situations mentioned previously.

When there is a successful response on the test connection, that interval becomes the lower bound for a binary search. When there is a failure only on the test connection, the current interval becomes the upper bound for the search. The keep-alive interval for the test connection is then set to the midpoint between the lower and newly found upper bound $\frac{lower+upper}{2}$. The search process is repeated until the optimium value is found, with a maximum of $2[log_2 t]$ iterations. Because this was devised in the context of P2P applications interacting, the authors also added a "gossip" feature, where a node will alert others if it detects that another node is not responding. Using the binary search method has the added benefit of a more accurate result for the optimum interval.

In [32], the authors propose a technique that uses extensions to the STUN protocol to determine the lifetime of a NAT mapping as part of a larger NAT discovery method. The client uses two source ports. First, the client sends a normal Binding Request from the first port. After a specified amount of time T, on the second port it sends another Binding Request with a RESPONSE-PORT attribute, both creating a new binding and instructing the server to deliver the

response to the first port. If the client receives a response on the first port, it knows the binding to the first port is still active. Otherwise the binding has expired. For each change of T, the client repeats the process of sending the Binding Requests from both ports. Like with the technique just discussed [38], the optimium time interval is determined with a binary search of the T values, from the point where a response is received to a point where one is not. The number of iterations or recommended precision are not specified and left up to the implementer. A downside of this technique is inconsistent results due to NAT device reboots or network traffic.

The authors of the patent [20] use a method similar to the ones previously discussed and that seems to be based on [38] and [39]. Their system involves a server, one or more clients, and one or more "network devices", which are defined as devices such as NATs or firewalls. A client maintains two connections to the server, one for normal data communication, and one for testing the keep-alive interval value. The keep-alive interval starts out with a known "safe" value, and is incremented by a pre-defined amount for the test connection. If a response is received on that connection, the interval is increased once again and the process is repeated. If no acknowledgment is received, the data connection is checked for connectivity. If a response has been received on that connection, the test connection keep-alive interval is decremented. The amount of the decrement is suggested to be the midpoint between the last known "safe" value and the new upper bound, similar to iGlance or possibly suggesting a binary search method. If there has been no response on the data connection, the system assumes a network failure has occurred.

This process repeats until the interval for the test connection is less than

or equal to the interval for the data connection. When that is the case the "optimium" keep-alive interval value has been found. There are a few aspects that differentiate this method from the others discussed. The first is that after the optimium value has been found, it is uploaded to the server, which then distributes it to other clients. Depending on the situation, the final keep-alive value could be calculated based on several clients, although an aggregation method for doing so is not described. This method also contains provisions to handle intervals for individual network devices (if they can be uniquely identified) as well as the situation where a known interval value ceases to work. Uploading the keep-alive interval value to the server and distributing it to clients may result in issues if there are NAT devices on the network which change the mapping lifetime based on the amount network traffic, or if the network topology changes. The authors note that one of the advantages of sharing the keep-alive interval with other clients is battery power conservation for clients running on portable devices, which is a concern they share with the authors of [5], which is discussed in Section 4.2.

To our knowledge, there is not much mention of optimizing keep-alive intervals using a single channel in the literature. Existing methods use a secondary channel for testing the keep-alive timeout value and perform normal data transmission over the primary channel. However, there are some published ideas that involve adapting keep-alive behavior based on network traffic for various scenarios.

For example, the patent [8] describes a technique for adaptable keep-alive intervals. Similar to other keep-alive optimization attempts, one of the goals is to reduce the number of keep-alive messages and cut down on network traf-

fic. In this context the technique will be used to monitor traffic on "Enterprise Extenders", which are devices that allow network traffic based on Systems Network Architecture (SNA) to be transmitted over an IP network. When a long period of inactivity is detected, the system can double the keep-alive interval up to a set maximum value. The authors note that alternative adjustments to the interval could be made instead. Conversely, the timer interval is set to the normal value when normal network traffic resumes. In terms of the normal keep-alive message flow, if there is no response, the keep-alive message is sent repeatedly until a pre-determined maximum retry count is reached, in which case the connection is terminated.

Another approach is taken in RFC 3519 [18], which describes a method of IP-in-IP tunneling to allow Mobile IP usage through a NAT device. The tunnel extensions contain a "keep-alive interval" setting. While no dynamic adjustment is done, the RFC states that the keep-alive message should only be sent if there has been no network traffic through the connection for a specified amount of time. This prevents unneccessary network traffic.

The patent described in [4] is focused on adapting the transmission of keep-alive messages from a client while being connected to a Virtual Private Network (VPN) over UDP. The focus is on mobile phones, but the authors note that the technique could be applied to any type of device. The core of the technique is a configurable timeout value. The mobile device (client) will enter a "power saving" mode and stop connection roaming scans and keep-alive messages if no traffic is detected during the specified length of time. When traffic is again detected, the device re-establishes a connection on the highest priority connection and resumes generation of keep-alive traffic. This is achieved via a "Power

Conservation Logic" component, which includes a power timeout timer and a re-registration module on the mobile device. If the NAT mapping has expired during the period of inactivity, the the mobile device communicates with the VPN server to associate the new IP address and port mapping with the existing VPN session.

4.4 "STUN Calc Keep-Alive" Overview

4.4.1 Overview

This section describes a new technique for determining the optimum keep-alive interval for use with the NAT devices on a network, and the next discusses the similarities and differences to the techniques discussed previously. Like the technique in [32], this is an extension of the STUN protocol. It consists of four new methods and one attribute. Responsibilities for the process are split among three logical entities: the client, the server controller, and the server message wheel. A brief description of that breakup is below.

Client The client is always in one of three states: SEARCH, RUN, or REFINE. It initiates the process by sending a request to the server. In the event of a timeout, it alerts the server to stop the current session and calculates a new interval value. If the client is configured in "single channel" mode, it will also alert the server to "reset" the next scheduled message for a session when other traffic is detected on the connection.

Server Controller The server controller processes the requests from the client and maintains a hash table with relevant information for each session.

Each session represents a NAT address and port binding. The server controller is responsible for adding and removing sessions from the message wheel.

Server Message Wheel The message wheel is a separate thread on the server. It maintains an ordered list of sessions for which to send keep-alive messages based on a priority queue, and is responsible for sending the messages out to the clients for each session.

The client begins the process in the SEARCH state, generating a new session ID and sending a STUN_CALC_KA_REQUEST message to the server. The server then creates a new session for the message wheel. Under normal conditions a client will send an explicit STUN_CALC_KA_STOP_REQUEST message to the server to end a session.

The message wheel sends out keep-alive messages for each session based on a priority queue. The sessions are organized based on a "NextSendTime" value which contains the scheduled send time of the next message. If there is a delay before the next message, the wheel will sleep for the appropriate amount of time, but may be interrupted by any new sessions. If the wheel is interrupted during a wait, it adds the current session back into the priority queue, pops the top item in the queue (which may or may not be the same session) and restarts the wait. After sending a message, the wheel will update the keep-alive interval (T) based on the current iteration in the session (K), the original constant delta (ΔT), and the current value of the divisor (Z). It may also update Z and the *direction* of the search based on the parameters of the session. In addition, the wheel handles scheduling the next message and updating the appropriate

session information.

In the SEARCH state, the client waits for STUN_CALC_KA_RESPONSE messages from the server. Unlike most of the other keep-alive optimization techniques described, the client does not respond to the successful reception of a message. When there is a timeout while waiting for a message, the client will send a STUN_CALC_KA_STOP_REQUEST message to the server, revert back to the last known good time interval, and increase the time interval based on K, ΔT, and Z. The client will then start a new session based on the new parameters. Z is used as a cutoff point for the adjustment and is increased based on the Z_{Mult} parameter each time it is used. Once Z reaches a pre-defined value (e.g. 1000, signifying an adequate precision), the client will switch to the RUN state. Alternatively, the client will end the session and switch to the RUN state if ΔT_{Min} is reached. If the client receives a message with a non-matching session ID (most likely from an old session), it will immediately send a STUN_CALC_KA_STOP_REQUEST message to the server to terminate that session.

When in the RUN state, the client sends a STUN_CALC_KA_REQUEST message to the server to begin a session and waits for a timeout, similar to the SEARCH state. However, when a timeout occurs, instead of calculating a new time interval value T, the client will increment the failure count, and determine whether or not to switch to the REFINE state based on the current failure rate of the messages.

The REFINE state is a simplified version of the SEARCH state behavior that can either increment or decrement the time interval. The client will begin with the current interval T, and decrement the value (using the BACKWARD

direction) by $K \times \frac{\Delta T}{z}$, where K is the number of timeouts experienced so far. Once a valid keep-alive value is found, the REFINE state will switch to the FORWARD direction, adjust Z, and increment the time interval until it experiences a timeout. At that point the time interval is set to the last known good time, and the client enters the RUN state.

The REFINE state was added due to some inconsistencies discovered when testing with the Linksys WRT-54G router. Even though one or two pings will be successful for a particular interval, the next one may not be if the interval value is "borderline" on the mapping lifetime value used by the NAT device. While not documented, it is possible that the router adjusts the NAT mapping lifetime based on other traffic. This state is also useful for mobile devices that may change networks frequently and need to quickly re-adjust the keep-alive value for a running application. This allows for a backwards adjustment without restarting in the SEARCH state. The ΔT and Z_{Mult} parameters for the REFINE state are independent of those for the SEARCH state, and ΔT should be significantly smaller.

This technique also provides the ability for the client to "build up" to the current keep-alive time interval in both the RUN state and the REFINE state by specifying the appropriate parameters in the KEEP_ALIVE_TIMER_DATA STUN attribute when initiating a session. This allows the system to utilize functionality on routers that use a longer address and port mapping lifetime value for UDP connections that have had several messages travel through the NAT device instead of just one (such as routers that use "assured" UDP sessions).

Single Channel Operation The normal usage of the STUN Calc Keep-Alive optimization technique is on a dedicated connection, much like the other keep-alive optimization techniques discussed. However, it can also be run on the same connection as normal application data. While the time taken to determine an appropriate keep-alive interval will be longer due to the interference of the other traffic, as long as there are a few pauses longer than the NAT mapping lifetime, an interval will be found. If the channel is particularly noisy, Z and Z_{Max} can be adjusted to limit the number of adjustments of Z to help find a value more quickly.

While operating on the same channel as application data, the client SEARCH state works much as it does when using its own connection. However, when an unrecognized message is received, a STUN_CALC_KA_RESET_REQUEST message is sent to the server, resetting the timer for the next keep-alive message scheduled to be sent to the client. There are three configurable settings to limit the number of reset messages sent: the maximum number of reset tries, the minimum amount of time before sending another reset message, and the delay before sending a message. These are described in more detail along with the pseudo-code for this functionality.

While this technique is able to find a keep-alive interval, it is not optimal since despite the minimum wait time and send delay, there is a potential for a large number of RESET messages to be sent, depending on the pattern of the other network traffic. While it was not tested for this thesis, a potentially more optimal single-channel implementation is described in Section 4.4.2.

4.4.2 Algorithm Details

In order to support the keep-alive optimization process, four new methods and one new attribute are added to the STUN protocol.

KEEP_ALIVE_TIMER_DATA Attribute

The new attribute is called KEEP_ALIVE_TIMER_DATA and has the structure shown in Figure 4.1 below.

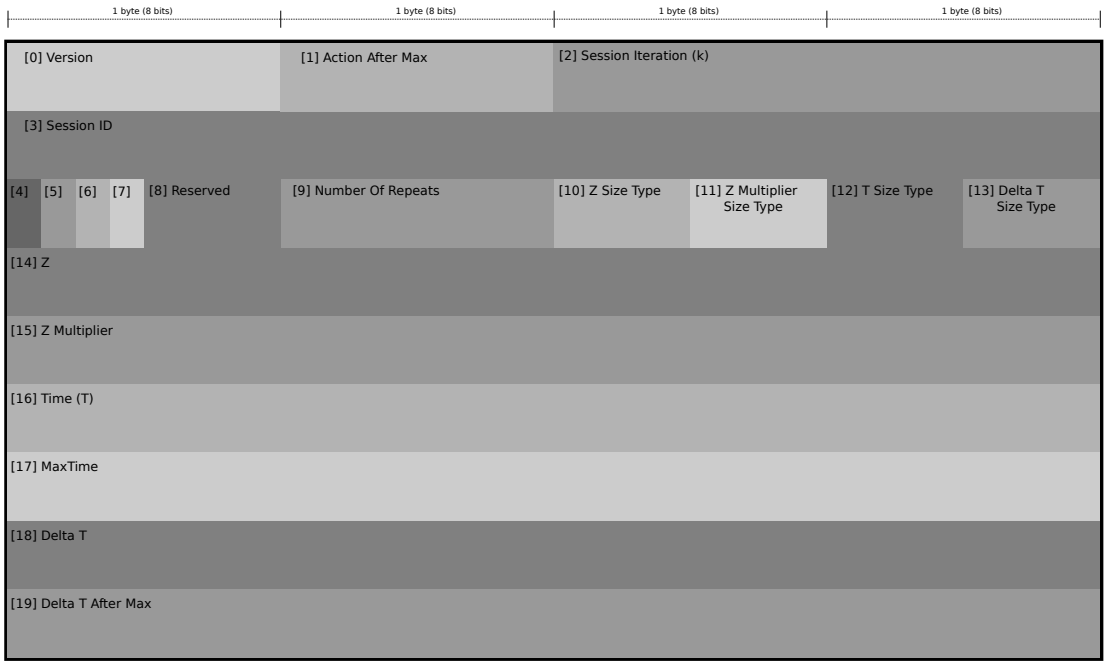


Figure 4.1: KEEP_ALIVE_TIMER_DATA Structure

[0] Version (1 byte) The version identifier for the protocol.

[1] Action After Maximum Time ($ActionAfterT_{Max}$**)** (1 byte) Specifies what action to take after "Maximum Time" is reached (if it is set). The possible values are 0 for MAINTAIN_CURRENT, 1 for STAY_CONSTANT, 2 for

CHANGE_TO_FORWARD, and 3 for CHANGE_TO_BACKWARD. Only the latter three are specified directly by the client.

[2] **Session Iteration (K)** (2 bytes) Signifies the current message iteration for the session. It is used by the message wheel on the server to update the "Interval Time" after each iteration of a session, and by the client to re-calculate the Interval Time after a timeout. This may not be a true indicator of the overall iteration, as the value may be reset or held constant depending on the session parameters.

[3] **Session ID** (4 bytes) An unique identifier for the keep-alive session. A client and server may take part in a large number of sessions before an optimium keep-alive value is found. Each time the client experiences a timeout while in the SEARCH or REFINE states or changes state it will start a new session.

[4] **Uses Starter Message** (1 bit) Determines whether or not a message will be sent to the client immediately after a request to begin a new session is received by the server. In normal usage this will always be set in order to achieve the best precision.

[5] **Server Adjusts Z** (1 bit) Indicates whether or not the server message wheel should adjust the value of Z. If set, the wheel will adjust Z before the next "Interval Time" calculation if the next value is one that has already been tested. This is used to perform a binary search. When this value is set, it implies a constant K value for the session.

[6] **Increment Type** (1 bit) Determines the type of increment to use when

adjusting the "Interval Time". It is set to 0 for LINEAR and 1 for GEOMETRIC.

[7] **Direction** (1 bit) Determines the direction of the "Interval Time" change for this session. It is set to 0 for FORWARD and 1 for BACKWARD.

[8] **Reserved** (4 bits) Reserved for future use. Currently used to align the data on a 4 byte boundary.

[9] **Repeat Count** (1 byte) Specifies the number of times to repeat each "Interval Time" value before making an adjustment.

[10,11,12,13] (4 bits each) These specify the size type of Z, Z_{Mult}, T and T_{Max}, and ΔT and $\Delta T After T_{Max}$, respectively. The value n is specified for a size of 2^n bytes. Valid values are 0 for a size of 1 byte, 1 for 2 bytes, 2 for 4 bytes, 3 for 8 bytes, and 4 for 16 bytes.

[14] **Z** (size depends on [10]) Specifies the Z value. This is used to adjust the "Interval Delta".

[15] **Z Multiplier (Z_{Mult})** (size depends on [11]) Specifies the multiplier for Z. This is used to adjust Z and the "Interval Delta" when the "Server Adjusts Z" flag is set.

[16] **Interval Time (T)** (size depends on [12]) Stores the current value of the keep-alive interval. This can be adjusted by the client or server depending on the state and the other parameters.

[17] **Maximum Time (T_{Max})** (size depends on [12]) Specifies when the "build-up" process is complete. When "Interval Time" reaches this value, the

server message wheel will change actions if appropriate.

[18] Interval Delta (ΔT) (size depends on[13]) Stores the current delta for the time. This is modified locally by the client after a timeout as a new basis for incrementing the keep-alive interval. It can also be modifed locally by the server if the "Server Adjusts Z" flag is set. The value specified in this data structure should be the original. The client and server will determine the current value based on Z.

[19] Interval Delta After Maximum Time ($\Delta TAfterT_{Max}$) (size depends on [13]) Specifies the delta to use after "Maximum Time" is reached. It can be modified locally by the server if the "Server Adjusts Z" flag is set, but like ΔT, only the original should be included in this data structure unless the client is in the REFINE state.

New STUN Methods

The new methods all make use of this new attribute.

STUN_CALC_KA_REQUEST Used by the client to initiate a keep-alive optimization session. The server determines whether or not to increment the time for each response in the session based on the Interval Delta value, and may make adjustments based on the parameters of the included KEEP_ALIVE_TIMER_DATA attribute described above.

STUN_CALC_KA_RESPONSE Used by the server to "ping" the client during a keep-alive optimization session. Unlike the STUN Binding Response, instead of sending this response once, the server will continue to send this

response message at intervals determined by the parameters of the keep-alive optimization session. The messages will continue to be sent until the client issues a stop request.

STUN_CALC_KA_STOP_REQUEST Used by the client to signal the server to end a keep-alive optimization session. These messages prevent the server from sending stray messages from a previous session to a client. Since the optimium interval is determined based on keeping the mapping alive with the keep-alive message traffic, extra messages can skew the calculation.

STUN_CALC_KA_RESET_REQUEST Used by the client to signal the server to reset the scheduled send time of the next message for a keep-alive optimization session. These messages are sent only when the client is configured in "single channel" mode, and are sent when other traffic is detected on the same connection as the keep-alive session messages.

The state changes for the client are shown in Figure 4.2 below.

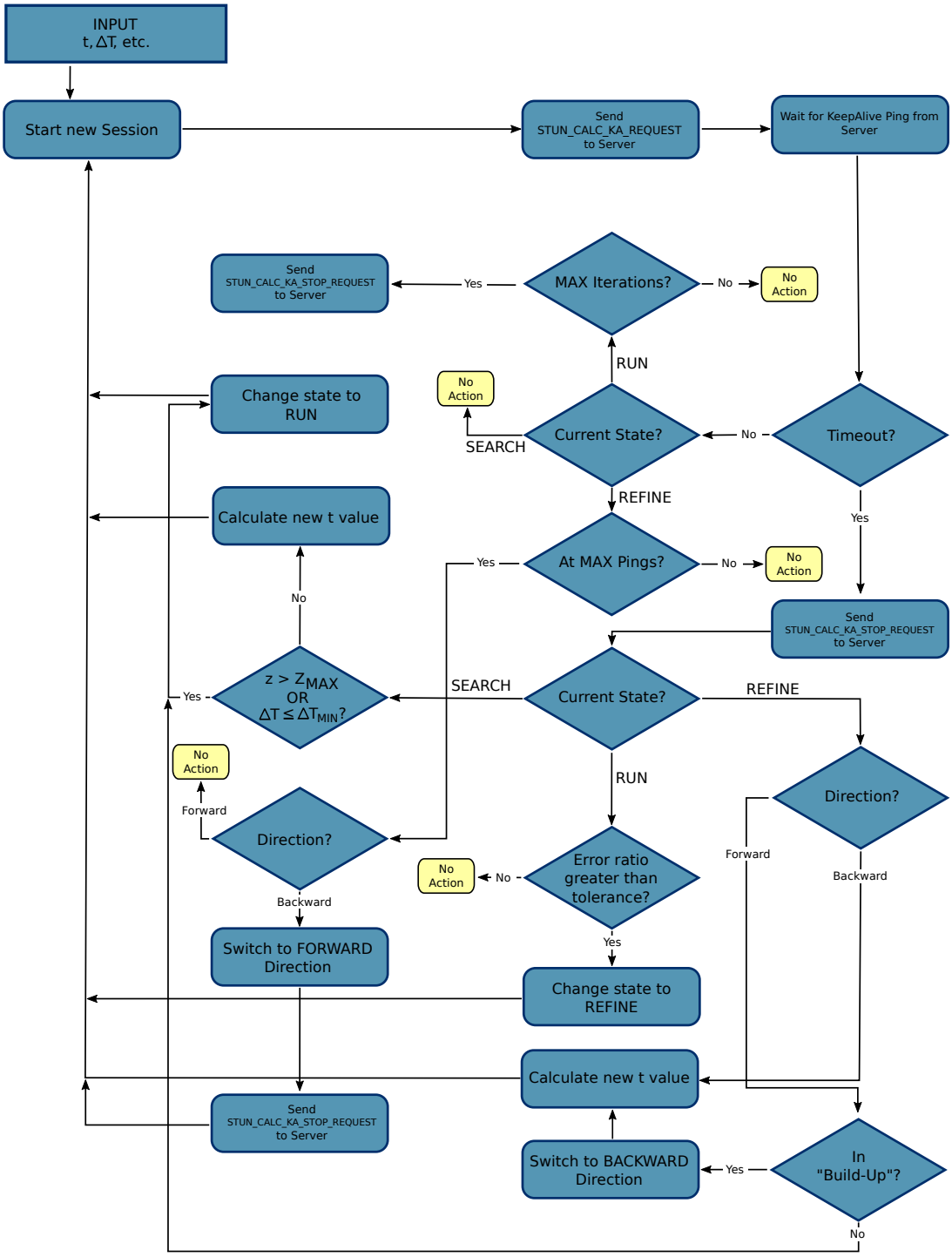


Figure 4.2: Keep-Alive Client State Change Behavior

After a client sends a STUN_CALC_KA_REQUEST, if $UsesStarterMessage$ is TRUE, the server sends the first STUN_CALC_KA_RESPONSE immediately, regardless of the value of T. This is done to establish the timeout interval as accurately as possible. Depending on the network configuration, the initial request may take several seconds to reach the server. If the server then delays for a period of time T before sending the first response, the timeout interval calculation would be off by that number of seconds. The immediate message sent by the server mitigates this issue by establishing the session with a start time known to the server.

The data structures and select pseudo-code for the operation of the Server Controller, the Server Message Wheel, and the state-based Client are described in the following sections. More complete pseudo-code descriptions and examples are shown in Appendices A and B and referenced where appropriate.

Server Controller

When the server receives a request for a new STUN_CALC_KA session, it creates a session based on the request parameters and adds the session to the wheel as shown in Algorithm 1. When it receives a RESET or STOP message, it interrupts the wheel thread and allows the wheel to handle the action. Each session is identified by a $SessionKey$ triple $\langle ClientAddress, ClientPort, ID \rangle$.

The $Session$ data structure is shared by both the Controller and the Message Wheel on the server. In addition to a $SessionKey$ object, the structure contains many items based on information from The KEEP_ALIVE_TIMER_DATA data structure shown in Figure 4.1. Those items include $NextAction$ (from $ActionAfterT_{Max}$), K, $UsesStarterMessage$, $ServerAdjustsZ$, $IncrementType$,

Direction, $MaxRepeats$ (from *RepeatCount*), Z, Z_{Mult}, T, T_{Max}, ΔT, and $\Delta TAfterT_{Max}$.

Additional items used to track the session state are shown below.

Variable	Description
$NextSendTime$	The send time of the next scheduled message
$InitialK$	The K value when the session started
$ClientK$ (K_l)	The K value to send to the client
$IsFirstSessionMessage$	TRUE if the current queued message is the first of the session
$Original\Delta T$	ΔT value when the session started
$Original\Delta TAfterT_{Max}$	$\Delta TAfterT_{Max}$ value when the session started
$T_{Max}WasReached$	TRUE when T_{Max} has been reached.
$InnerRepeatCount$	Tracks inner repeats when $MaxRepeats > 0$ Reset when $MaxRepeats$ is reached
$OuterRepeatCount$	Tracks outer repeats when $MaxRepeats > 0$

Table 4.1: Server Session Variables

```
1 when STUN_CALC_KA_REQUEST(KeepAliveTimerData) do
2 |   if Session is New then
3 |   |   Add to Session Hash Table
4
5 when STUN_CALC_KA_STOP_REQUEST(ID)
6 |   Remove Session from Session Hash Table
7
8 when STUN_CALC_KA_RESET_REQUEST(ID)
9 |   Reset Timer for Next Session Message
```

Algorithm 1: Server Controller Behavior

Server Message Wheel

The wheel contains several controller methods to handle the MessageQueue, which is the priority queue used for scheduling messages, and the SessionHashTable, containing data for all of the sessions. The latter data structure is shared with the Server Controller thread. An overview of the methods is shown below.

After obtaining a session from the priority queue, the wheel first sends the message after waiting the appropriate amount of time.

```
1 when Wheel Session s do
2 |   if s.NextSendTime > SystemTime() then
3 |   |   pause the thread until the correct time
4 |   if thread is interrupted then
5 |   |   MessageQueue.add(s)
6 |   |   s ← MessageQueue.pop()
7 |   |   CONTINUE
8 |   send to(ClientAddress, STUN_CALC_KA_RESPONSE)
```

The wheel then checks whether or not the value of T needs to be updated on this iteration. If it does, the adjustment is calculated according to the "CalcNextTime" procedure described in Appendix A.2. Once the appropriate adjustments are made, the "NextSendTime" is set for the session, and the local variables are updated. If $ServerAdjustsZ$ is set, K is kept constant to allow a binary search.

The Wheel then checks whether or not T_{Max} was reached after the adjustment. If so, it updates the session variables and direction as described in the "AdjustAfterMaxTime" procedure detailed in Appendix A.2. After all calculations are done and the session has been updated, the wheel adds it back into the priority queue.

```
1  if s.UsesStarterMessage AND s.IsFirstSessionMessage then
2  |  ShouldCalcNextTime ← FALSE
3  else
4  |  ShouldCalcNextTime ← TRUE
5  if s.IsFirstSessionMessage = TRUE AND s.ServerAdjustsZ = FALSE then
   |  // Initial adjustment for ΔT if ServerAdjustsZ is FALSE.
6  |  if s.T_Max WasReached = TRUE then
7  |  |  s.ΔTAfterT_Max ← s.OriginalΔTAfterT_Max / s.Z
8  |  else
9  |  |  s.ΔT ← s.OriginalΔT / s.Z
10
11 if s.T ≠ s.T_Max AND ShouldCalcNextTime = TRUE then
12 |  CalcNextTime( s )
13
14 s.NextSendTime ← s.T + SystemTime()
15 if s.ServerAdjustsZ = FALSE and ShouldCalcNextTime = TRUE then
16 |  s.K ← s.K + 1
17
18 if s.T_Max > 0 AND s.T ≥ s.T_Max AND ShouldCalcNextTime = TRUE then
19 |  AdjustAfterMaxTime( s )
20
21 MessageQueue.add( s )
```

Each session is stored with both "inner" and "outer" iteration counts to allow coordination with the RUN and REFINE states of the client. The "CalcNextTime" procedure will only make adjustments (other than incrementing counts) when the $InnerRepeatCount$ variable is equal to the $MaxRepeats$ parameter. This allows the same T value to be repeated for an arbitrary number of iterations.

When the adjustment criteria is met, the Server Message Wheel will update T, and if $ServerAdjustsZ$ is set, also update Z and the appropriate ΔT value.

Client (Dual Channel)

The client initiates a new session, both when first starting the keep-alive optimization process and after each timeout. The input parameters for client initialization are shown below.

Input: $ServerAddress$, $SearchParams$, $RunParams$, $RefineParams$, T, $State$, $UsesStarterMessage$

1 **SearchParams:** $\langle\ \Delta T,\ \Delta T_{Min},\ Z,\ Z_{Max},\ Z_{Mult},\ IncrementType\ \rangle$

2 **RunParams:** $\langle\ FailureTolerance,\ RepeatCount\ \rangle$

3 **RefineParams:** $\langle\ \Delta T,\ Z,\ Z_{Mult},\ PingsPerAttempt\ \rangle$

Algorithm 2: Client Initialization Parameters

Additional details about the sets of parameters for each state are described in the tables below.

Parameter	Description
ΔT	The base delta value for modifying the interval T
ΔT_{Min}	Optional. The client will stop the search when ΔT reaches this value Ignored when set to 0
Z	Used as the divisor for ΔT after a timeout
Z_{Max}	Optional. The client will stop the search when Z reaches this value Ignored when set to 0
Z_{Mult}	Multiplier for Z after a timeout
$IncrementType$	Specifies type of multiplication (LINEAR or GEOMETRIC)

Table 4.2: Search State Parameters

Parameter	Description
$FailureTolerance$	Acceptable ratio of failures to successes
$RepeatCount$	The number of runs before exiting the state When set to 0, the client will run indefinitely

Table 4.3: Run State Parameters

Parameter	Description
ΔT	The base delta value for modifying the interval T
Z	Used as the divisor for ΔT
Z_{Mult}	Multiplier for Z
$PingsPerAttempt$	The number of pings to try for each T value

Table 4.4: Refine State Parameters

The client uses several local variables to help track the status of the keep-alive interval optimizaton. The core variables are described in Table 4.5 below, and expanded on in the text that follows.

T_{Max} is not used in the SEARCH state, but because of the "build up" process, the RUN and REFINE states initially set T_{Max} to T, and set T to $SearchParams.\Delta T$. This allows the client to build up to the actual T value using the smaller value of the delta. When T reaches T_{Max}, the client sets $BuildingUpToTime$ to FALSE, and treats the calculations accordingly.

As part of the initialization, the client also establishes the $BaseMessageCount$ based on whether or not the $UsesStarterMessage$ parameter is set. This value (set to either 0 or 1) is used as a base for the client to determine whether or not any T values have been successful for a particular session.

Variable	Description
T	The current value of the keep-alive interval
T_{Max}	Maxinum known value for T Used in REFINE and RUN states for "build-up"
$LastGoodT$	The last known T value that did not result in a timeout
Ω_K	The K value to use when multiplying based on $IncrementType$ (LINEAR $\leftarrow$K, GEOMETRIC $\leftarrow 2^K$)
$Direction$	The current search direction (FORWARD or BACKWARD)
$TimeoutCount$	The total number of timeouts
$BaseMessageCount$	The number of messages to skip before calculations start (0 or 1)
$SessionMessageCount$	The number of messages received for the session
$TotalMessageCount$	The total number of messages received
$RunCount$	The number of messages received while in the RUN state
$RefineCount$	The number of messages received while in the REFINE state
$RefineTimeoutCount$	The number of timeouts for the current T value in the REFINE state
$RefineMinValueFound$	TRUE of a valid T value has been found in the REFINE state
$StartNewSession$	TRUE if a message should be sent to the server
$BuildingUpToTime$	TRUE if in T has not yet reached T_{Max}
$ExitOnNextMessage$	TRUE if the state should change after the next iteration. Used with the ΔT_{Min} parameter

Table 4.5: Client Local Variable Summary

The "ClientLoopStart" functionality of the client determines what needs to be done both directly after initialization and after the last server message or timeout event is processed. The client first checks to see whether or not

the state should be changed based on the current values of $\frac{SearchParams.\Delta T}{Z}$ and $SearchParams.\Delta T_{Min}$. If those exit criteria are met, the process will continue for one more iteration to allow the current T value to be checked. It will also check if the condition $Z > Z_{Max}$ has been met and will change the state immediately if that is the case.

When sending a message, this procedure packages the current keep-alive state data into a KEEP_ALIVE_TIMER_DATA data structure (shown in Figure 4.1) and sends it to the server to initiate a new session. With the test implementation this functionality was built as the main program loop. A high-level summary of this functionality is show below.

```
1  procedure ClientLoopStart do
2      if ExitOnNextMessage = TRUE then
3          T ← LastGoodT
4          Switch to RUN State
5
6      if IN SEARCH State then
7          if SearchParams.ΔT / Z ≤ SearchParams.ΔT_Min then
8              ExitOnNextMessage ← TRUE
9          if SearchParams.Z_Max > 0 AND Z > SearchParams.Z_Max then
10             T ← LastGoodT
11             Switch to RUN State
12
13     if StartNewSession = TRUE then
14         Generate New Session ID
15         SessionMessageCount ← 0
16         sendto(ServerAddress, STUN_CALC_KA_REQUEST)
17
18     TimeOut ← CalculateNextTimeout()
19     alarm timeout( TimeOut )
20
```

If $StartNewSession$ is TRUE, then the client will send a session start request to the server. Otherwise the client simply sets the new timeout period and waits for the next message from the server. In each case the time to wait for a reply from the server is calculated in the "CalculateNextTimeout" procedure

described in Appendix B.2.

When the client receives a message from the server, it first verifies that the message is a STUN message of the correct type. In dual channel mode (normal operation), it simply discards unrecognized messages. In single channel mode, it handles them as described in Section 4.4.2.

If $UsesStarterMessage$ is true, the first message is discarded as well, as its only purpose is to create the external address and port mapping on the NAT device. Otherwise the client processes the message and adjusts its variables accordingly.

```
1  when STUN_CALC_KA_RESPONSE(CalcKaResponse r) from ServerAddress do
2      if not correct message type then
3          CONTINUE
4      if not correct (ID) then
5          sendto(ServerAddress, STUN_CALC_KA_STOP_REQUEST)
6          CONTINUE
7
8      SessionMessageCount ← SessionMessageCount + 1
9      TotalMessageCount ← TotalMessageCount + 1
10     K ← r.K
11
12     if UsesStarterMessage = TRUE AND SessionMessageCount = 1 then
           // Take no action.  Message still counts towards total.
```

For the SEARCH state, the client simply updates its local variables based on state information from the server. In the RUN state, the client will check if the session is currently "building up" to T_{Max}, update the status accordingly, and also send a STOP request if $RunParams.RepeatCount$ has been reached.

```
1  if State = SEARCH then
2  |  Update Z, LastGoodT, and T from Server
3
4  if State = RUN then
5  |  Update Z, LastGoodT, and T from Server
6  |  if BuildingUpToTime = TRUE then
7  |  |  if T ≥ T_Max then
8  |  |  |  BuildingUpToTime ← FALSE
9  |  |  |  Increment RunCount
10 |  else
11 |  |  Increment RunCount
12 |  |  if RunParams.RepeatCount > 0 AND RunCount ≥ RunParams.RepeatCount
   |  |  then
13 |  |  |  sendto(ServerAddress, STUN_CALC_KA_STOP_REQUEST)
14 |  |  |  Exit Optmization Process
```

For the REFINE state, the client first checks to see if it is currently "building up" to T_{Max}, much like for the RUN state. It then checks whether or not the current repeat iteration has reached $RefineParams.PingsPerAttempt$. If so, it will set $LastGoodT$ and either switch to the FORWARD direction (if currently moving BACKWARD), or take no action. In the REFINE state, Z is only adjusted locally on the client. Regardless of the local Z value, the client will always send a Z value of 1 to the server. Since Z is only adjusted once throughout this process, there is no ΔT_{Min} parameter for the REFINE state. Unlike the SEARCH state, the REFINE state always resets K to 0, regardless of the value of $SearchParams.IncrementType$.

```
1  if State = REFINE then
2      Update Direction From Server
3      if BuildingUpToTime = TRUE then
4          Update T From Server
5          if T ≥ T_Max then
6              BuildingUpToTime ← FALSE
7              Increment RefineCount
8      else
9          Increment RefineCount
10         if RefineCount ≥ RefineParams.PingsPerAttempt then
11             Reset RefineCount
12             Update T and LastGoodT From Server
13             if Direction = BACKWARD then
14                 Reset RefineTimeoutCount
15                 Reset K
16                 RefineMinValueFound ← TRUE
17
18                 Z ← Z × RefineParams.Z_Mult
19                 T ← T + (RefineParams.ΔT / Z)
20                 Switch to FORWARD Direction
21
22                 sendto(ServerAddress, STUN_CALC_KA_STOP_REQUEST)
23                 StartNewSession ← TRUE
24                 goto ClientLoopStart
25         else
26             if T ≠ T_Server then
27                 Reset RefineCount
28                 Update T From Server
```

If a new session has not been started, the client will then resume listening for messages.

```
1 StartNewSession ← FALSE
2 goto ClientLoopStart
```

When experiencing a timeout, in all cases the first step for the client is to send a STOP request to the server in order to terminate the current session.

```
1 when TIMEOUT do
2 |   sendto(ServerAddress, STUN_CALC_KA_STOP_REQUEST)
3 |   Increment TimeoutCount
```

When in the SEARCH state, the client will update Z to synchronize the value with the server, and set $ServerAdjustsZ$ if it is not already set to allow the server to help with the search and avoid unnecessary timeouts.

```
1 if State = SEARCH then
2 |   Z ← Z × SearchParams.Z_Mult
3 |   if ServerAdjustsZ = TRUE then
4 |   |   if SessionMessageCount > BaseMessageCount then
5 |   |   |   Z ← Z × SearchParams.Z_Mult
6 |   else
7 |   |   ServerAdjustsZ ← TRUE
8 |   T ← LastGoodT + Ω_K × SearchParams.ΔT / Z
```

Timeout handling in the RUN state involves checking whether or not the total timeout count is within the set failure tolerance. If not, the client will exit to the REFINE state. If there is a build up sequence involved, the client also updates T to T_{Max} to give the REFINE state the appropriate starting value.

```
1 if State = RUN then
2 |   if TimeoutCount / (TotalMessageCount+1) > RunParams.FailureTolerance then
3 |   |   if BuildingUpToTime = TRUE AND T < T_Max then
4 |   |   |   T ← T_Max
5 |   |   Switch to REFINE State
```

When in the REFINE state, the client is attempting to stabilize on a value as soon as possible. If there is a timeout while moving BACKWARD, T is decremented to allow the search for a valid T value to continue. If moving FORWARD, the client checks whether or not the session is still "building up" to T_{Max} and whether or not a good value for T has been found. If currently building up and no T value has been found, the current T value becomes the starting

point of the search and the direction switches to BACKWARD. Otherwise, a minimum good value for T has been found, so the client falls back to that value and exits to the RUN state.

```
1  if State = REFINE then
2  |  Increment RefineTimeoutCount
3  |  Reset RefineCount
4  |  Reset K
5  |  if Direction = BACKWARD then
6  |  |  Keep BACKWARD Direction
7  |  |  T ← T − RefineTimeoutCount × RefineParams.ΔT / Z
8  |  |  LastGoodT ← T
9  |  else if Direction = FORWARD then
10 |  |  if BuildingUpToTime = TRUE AND RefineMinValueFound = FALSE then
11 |  |  |  Switch to BACKWARD Direction
12 |  |  |  T ← T_Max
13 |  |  |  T ← T − RefineTimeoutCount × RefineParams.ΔT / Z
14 |  |  |  LastGoodT ← T
15 |  |  else
16 |  |  |  T ← LastGoodT
17 |  |  |  Switch to RUN State
```

In all cases the client will then start a new session.

```
1  StartNewSession ← TRUE
2  goto ClientLoopStart
```

Client (Single Channel)

In single channel mode, instead of ignoring unrecognized traffic, the client sends RESET messages to tell the server to reset the timer on the next message for the current keep-alive session. This allows the client to determine (to some extent) the NAT binding lifetime of the NAT device as long as there are some pauses in traffic on the connection that are longer than that lifetime.

Three configurable parameters guide the client's behavior: MAX_RESET_TRIES, MIN_RESET_INTERVAL, and RESET_DELAY. The parameters were set to the values **10**, **5000 ms**, and **2000 ms** respectively for the

experiments described in this paper.

The client keeps a count of the number of RESET messages sent, and uses RESET_DELAY to attempt to minimize the number of messages sent. Once MAX_RESET_TRIES has been reached, the client reacts in much the same way as it does during a timeout in the SEARCH state. Z is first adjusted, and then T. This allows the client to try out a smaller interval and possibly get a better T value due to the smaller increment. The RESET message count is reset every time MAX_RESET_TRIES is reached.

The MIN_RESET_INTERVAL setting is the amount of time the client waits before sending another RESET message after one has been sent. If this value has not been reached, the client will reset the timer on the next scheduled RESET messsage. This helps mitigate the extra traffic created by the RESET messages, while still minimizing extra keep-alive messages from the server.

As mentioned earlier, this implementation may not be optimal due to the potentially large number of RESET messages generated. An alternate implementation is described in the next section.

```
1  when other network traffic detected do
2      if ResetCount < MAX_RESET_TRIES then
3          if SystemTime() − LastResetAttempt > MIN_RESET_INTERVAL then
4              SendReset()
5              LastResetAttempt ← SystemTime()
6              ResetCount ← ResetCount + 1
7          else
               // ResetCount is incremented after the scheduled RESET is sent.
8              if reset is scheduled then
9                  ResetScheduledReset( RESET_DELAY )
10             else
11                 ScheduleReset( RESET_DELAY )
12     else
13         sendto(ServerAddress, STUN_CALC_KA_STOP_REQUEST(ID))
14         K ← K + 1
15         Z ← Z × SearchParams.Z_Mult
16         switch IncrementType do
17             case LINEAR
18                 Ω_K ← K
19             case GEOMETRIC
20                 Ω_K ← 2^K
21         if ServerAdjustsZ = TRUE then
22             Z ← Z × Z_Mult
23         T ← LastGoodT + Ω_K × SearchParams.ΔT / Z
24         ResetCount ← 0
25
26         StartNewSession ← TRUE
27         goto ClientLoopStart
28     CONTINUE
```

Algorithm 3: Client Single Channel Handling Behavior

Client (Single Channel Alternate Implementation)

While this variation of the single channel client was not implemented and tested for this thesis, it is in theory a more optimal solution for determining an appropriate keep-alive interval value while running on the same channel as normal application data.

This technique requires an addition to the KEEP_ALIVE_TIMER_DATA structure described in Section 4.4.2. One of the reserved bits can instead be used as a *SingleMessage* flag. When set, the server would only send one keep-

alive message (two if the $UsesStarterMessage$ flag is also set) to the client before ending the session based on the parameters of the keep-alive session request.

Instead of sending a RESET message to the server when non-keep-alive traffic is received, the client simply then ignores the next response from the server. In that case when the server message is received, the client initiates another session using the unchanged T value. Otherwise, the client updates the current value of T based on the maximum length of time so far with no traffic, and then initiates a new session. The client also tracks the amount of time between each piece of traffic received. When a timeout is experienced, aside from resetting the new timestamps and adjusting K, Z and T are updated the same way as with the other client implementations.

Pseudo-code for this alternate implementation for the portion of the client that handles incoming traffic is shown below. This version does not show handling of "starter" messages, or adjustments to the total and session message counts.

```
1  IncrementType ← INPUT: IncrementType (LINEAR or GEOMETRIC)
2  T ← INPUT: Starting Time Interval
   // The timestamp of the previous message received
3  LastReceivedTimeStamp ← 0
   // The timestamp of the message just received
4  NewReceivedTimeStamp ← 0
   // The last time interval that had no traffic
5  NewT ← 0
   // If TRUE, the next message from the server does not trigger a change in T
6  IgnoreNextServerMessage ← FALSE
   // The last known good value of T.  Used the same as for dual channel mode
7  LastGoodT ← T
   // The number of iterations for non-ignored server communication
8  K ← (1 OR 0) based on LINEAR or GEOMETRIC IncrementType
   // Indicates whether or not to send a new session request
9  StartNewSession ← TRUE
10 when message received do
11     NewReceivedTimeStamp ← SystemTime()
12     if LastReceivedTimeStamp > 0 then
13         NewT ← NewReceivedTimeStamp − LastReceivedTimeStamp
14     LastReceivedTimeStamp ← NewReceivedTimeStamp
       // Update T if the time interval with no traffic is the longest so far
15     if NewT > T then
16         T ← NewT
17         LastGoodT ← T
       // By default there is no new session (e.g.  for other traffic and
          incorrect session ID's)
18     StartNewSession ← FALSE
19     if message is non-keep-alive traffic then
20         IgnoreNextServerMessage ← TRUE
21     else
22         if correct (ID) then
23             if IgnoreNextServerMessage = TRUE then
24                 IgnoreNextServerMessage ← FALSE
25             else
26                 LastGoodT ← T
27                 if ServerAdjustsZ = TRUE then
28                     Z ← Z × Z_Mult
29                 switch IncrementType do
30                     case LINEAR
31                         Ω_K ← K
32                     case GEOMETRIC
33                         Ω_K ← 2^K
34                     T ← LastGoodT + Ω_K × SearchParams.ΔT / Z
35                 if ServerAdjustsZ = FALSE then
36                     K ← K + 1
37             StartNewSession ← TRUE
       // Resume listening for messages, first starting a new session if
          appropriate
38     goto ClientLoopStart with SingleMessage flag set
```

Algorithm 4: Client Single Channel Handling Behavior (Alternate)

4.5 Comparison to Related Work

While the STUN Calc Keep-Alive technique has much common ground with existing research in the area of keep-alive intervals, there are a number of differences. The first is that both a STUN client and a STUN server are heavily involved in the interval optimization process. In the existing techniques discussed, there was primarily one agent controlling the process. In the case of the STUN enhancement, it was the client [32]. Many of the other algorithms are meant for P2P environments and therefore simply have a node go through the process for each connection to the other nodes. For the STUN Calc Keep-Alive technique, once the client initiates the process, it only needs to contact the server in the event of a timeout based on its state. The NAT mapping is kept active by the server's messages to the client. This technique is intended for use with UDP, but could be adapted to work with TCP in the future.

Much of the "heavy lifting", such as keeping a timer of when to send keep-alive messages and actually sending the keep-alive messages is done by the server. This is advantageous as it saves processing power and battery life on the client, which may be running on an embedded or mobile device with limits on both. The other techniques rely on the client responding to each ping from the server to signify that the connection is still alive.

Another difference of STUN Calc Keep-Alive is the state based management of the client. The client switches states based on feedback from the server. It begins in the SEARCH state, which has the same goals as the previously discussed techniques. Once an "optimium" keep-alive interval has been found, it switches to the RUN state. It also includes the REFINE state to make

adjustments to the keep-alive interval when needed by searching backwards. These states are discussed in greater detail in Section 4.4.1.

The STUN Calc Keep-Alive technique can also be used incrementally when appropriate in the SEARCH state. The previously mentioned techniques find the maximum timeout value by either adding a constant value to the current keep-alive interval or by doubling the current interval value. With this technique, the growth of the interval is determined based on the number of successful responses for the current "session" (K) as well as a pre-defined constant delta value (ΔT). STUN Calc Keep-Alive also allows for a "buffer" segment of time to account for network latency. When a client discovers that a NAT address and port mapping linked to a session has timed out while in the SEARCH state, the time interval T is re-adjusted as follows: $T \leftarrow LastGoodTime + \Omega_K \times \frac{\Delta T}{Z}$, where Ω_K is based on the number of successful iterations for the session (K or 2^K), and Z is a variable that helps determine the cutoff point in the search. The server can also be instructed to adjust Z itself in order to perform a binary search using this formula without constant timeouts from the client.

Another unique feature is the ability to keep the reliability of the keep-alive interval within a certain threshold. This is achieved using the RUN and REFINE states. With the exception of [8], in the techniques discussed if there is a timeout it is assumed that there is either a network problem or that the NAT mapping has been lost. During the RUN state of the STUN Calc Keep-Alive technique, the server sends keep-alive messages to the client at a constant interval. If the client experiences a timeout, it increments an internal failure count. It then uses the following method to determine whether or not it should change to the REFINE state: $\frac{RunFailureCount}{TotalMessageCount+1}$, where $TotalMessageCount$ is

the number of successful responses for the current session. This allows for drops of UDP packets that sometimes occur and for establishing an acceptable rate of failure for the keep-alive interval. Like with the SEARCH state, the RUN state allows for an adjustable "buffer" to avoid needing to recalculate the keep-alive interval when there are minor changes in the mapping time, related to either network traffic levels or adjustments made by the NAT device itself.

In terms of the existing STUN extension described in [32], the primary difference is the distribution of responsibility. That technique uses a custom attribute and normal STUN binding requests and responses for data gathering, with the client doing most of the work. The STUN Calc Keep-Alive technique uses additional STUN methods in addition to a new STUN attribute. While this does add some complexity, as already discussed it also shifts some of the processing responsibility to the server. Using the server to slowly increase the interval without creating a new mapping for each timeout test also allows the STUN Calc Keep-Alive technique to take advantage of situations where the NAT device uses a longer lifetime for established UDP traffic, and allows the client to automatically "build up" to a certain interval time without needing to re-enter the SEARCH state; a feature not present in the other techniques.

Unlike the techniques described in [38], [20], and [8], the STUN Calc Keep-Alive technique is meant to be used on an individual client basis. It does not include any way for the server to send the optimum keep-alive interval value to other clients, as each session (one per NAT mapping) is handled individually. The latter two techniques also seem to involve stopping the process once the interval is found. The STUN Calc Keep-Alive technique is meant to run continually in the background of the client and the server to re-adjust the interval

as needed.

The single channel keep-alive interval determination features of the STUN Calc Keep-Alive technique have different goals than existing keep-alive techniques that operate on or monitor the same channel as the data. While the techniques described in Section 4.3.3 enable, disable, or adjust the frequency of keep-alive messages based on the amount of current network traffic, the technique presented in this chapter aims to find the largest NAT mapping timeout value despite the presence of other network traffic. The version that was implemented and tested for this thesis takes a more active role by sending RESET messages, while the alternate proposed implementation is more passive. The goal for the former is for the value to be as close as possible to what would be found if the technique were run on its own connection without any interference. The latter shares the goal of eventually finding the maximum NAT mapping lifetime, but in a more passive manner. It works to simply find the largest keep-alive interval needed at the current time until larger windows of time with no traffic become available.

4.6 Objectives

The objectives of these experiments are to evaluate the STUN Calc Keep-Alive method described in the previous section for determining an optimum keep-alive interval. The goals are to evaluate the following:

- The "optimality" of the results of running the SEARCH state of the technique compared to others when run on a dedicated channel.

- The ability of the technique using the REFINE state to settle on a stable keep-alive interval even on routers with inconsistent behavior.
- The ability of the technique to determine a reliable keep-alive interval in a single channel situation where other data is being multiplexed with the keep-alive messages.

4.7 Theoretical Analysis

4.7.1 Linear Increment

This formula is used to first perform a linear increment of the ΔT value to find the upper bound on the address and port mapping lifetime, and then perform a binary search to determine the actual timeout value.

The formula used to find the upper bound is as follows:

$$T_{Upper} = \sum_{k=1}^{n} k \times \frac{\Delta T}{z}$$

The constant can be moved out of the summation:

$$T_{Upper} = \frac{\Delta T}{z} \times \sum_{k=1}^{n} k$$

It can then be moved to the other side of the equation, and the summation simplified.

$$\begin{aligned} \frac{T_{Upper} \times z}{\Delta T} &= \sum_{k=1}^{n} k \\ &= \frac{n(n+1)}{2} \end{aligned}$$

$$\frac{2 \times T_{Upper} \times z}{\Delta T} = n^2 + n$$

The equation can be solved for n using the quadratic formula:

$$0 = n^2 + n - \frac{2 \times T_{Upper} \times z}{\Delta T}$$

$$\begin{aligned} n &= \frac{-1 + \sqrt{(1)^2 - 4 \times 1 \times (-\frac{2 \times T_{Upper} \times z}{\Delta T})}}{2 \times 1} \\ &= \frac{-1 + \sqrt{1 + \frac{4 \times 2 \times T_{Upper} \times z}{\Delta T}}}{2} \end{aligned}$$

The positive root is the number of messages needed to find the upper bound. For the binary search, z always starts out as 1, so the formula can be further simplified:

$$n = \frac{-1 + \sqrt{1 + \frac{8 \times T_{Upper}}{\Delta T}}}{2}$$

Once the upper bound has been found, to perform a binary search z is multiplied by 2 each iteration. The result of z being multiplied by 2 is represented with 2^n, where n is the number of iterations.

$$\Delta T = 2^n$$

$$log_2 \Delta T = n$$

Thus the total maximum number of steps required to find the keep-alive interval are:

$$n = log_2 \Delta T + \frac{-1 + \sqrt{1 + \frac{8 \times T_{Upper}}{\Delta T}}}{2}$$

4.7.2 Geometric Increment

This formula is used to perform a geometric increment of the ΔT value to find the upper bound on the address and port mapping lifetime. Once the upper bound is found, a binary search is performed like with the Linear Increment.

The formula used to find the upper bound is as follows:

$$T_{Upper} = \sum_{k=0}^{n} 2^k \times \frac{\Delta T}{z}$$

Like with the linear increment, the constant can be moved out of the summation:

$$T_{Upper} = \frac{\Delta T}{z} \times \sum_{k=0}^{n} 2^k$$

It can then be moved to the other side of the equation, and the summation simplified.

$$\begin{aligned} \frac{T_{Upper} \times z}{\Delta T} &= \sum_{k=0}^{n} 2^k \\ &= \frac{1 - 2^{n+1}}{1 - 2} \\ &= \frac{1 - 2^{n+1}}{-1} \end{aligned}$$

$$\frac{-1 \times T_{Upper} \times z}{\Delta T} = 1 - 2^{n+1}$$

$$\frac{-1 \times T_{Upper} \times z}{\Delta T} - 1 = -2^{n+1}$$

$$\frac{1 \times T_{Upper} \times z}{\Delta T} + 1 = 2^{n+1}$$

Take the natural log of both sides to eliminate the exponent.

$$\ln\left(\frac{T_{Upper} \times z}{\Delta T} + 1\right) = \ln\left(2^{n+1}\right)$$
$$= (n+1) \times \ln(2)$$

$$\frac{\ln\left(\frac{T_{Upper} \times z}{\Delta T} + 1\right)}{\ln(2)} = n + 1$$

$$\frac{\ln\left(\frac{T_{Upper} \times z}{\Delta T} + 1\right)}{\ln(2)} - 1 = n$$

For the binary search, z always starts out as 1, so the formula can be further simplified:

$$\frac{\ln\left(\frac{T_{Upper}}{\Delta T} + 1\right)}{\ln(2)} - 1 = n$$

Once the upper bound has been found, to perform a binary search z is multiplied by 2 each iteration, following the same process described for the Linear Increment in Section 4.7.1. This results in the following maximum number of steps:

$$n = log_2 \Delta T + \frac{\ln\left(\frac{T_{Upper}}{\Delta T} + 1\right)}{\ln(2)} - 1$$

4.7.3 Discussion

The geometric increment method can potentially find the upper limit of the NAT mapping lifetime with fewer messages, depending on the difference between the lifetime and the initial ΔT value. In situations where there is no knowledge of the NAT mapping lifetime or a small ΔT value is used, this is the better option.

The other keep-alive optimization techniques discussed use a basic geometric increment (doubling the initial value of T for each iteration).

Since the other techniques described involve two messages for each iteration, a ping and a reply, in the worst case scenario the total number of messages exchanged will be double the result of the equations above. Since the client only responds in the event of a timeout when using the STUN Calc Keep-Alive SEARCH state, there is the potential for fewer overall messages as not all pings will include a response.

4.8 Tools

For these experiments the following equipment and software was used:

- Wireshark
- STUN Client running with Java 1.7 on Slackware Linux 14.1
- STUN Server running with Java 1.7 on Windows 8.1
- Linksys WRT-54G router with standard firmware
- ASUS RT-N16 with ASUSWRT-MERLIN firmware

The Linksys WRT-54G router was inconsistent in terms of the time that a NAT mapping was kept alive, which provided the motivation for the addition of the REFINE state to the algorithm. Running the algorithm multiple times resulted in slightly different values, and in some cases the interval found would only work for one or two iterations. It also did not have a configurable UDP lifetime value. This is in contrast to the ASUS RT-N16 router, which was much more consistent in its NAT mapping lifetime values. While the RT-N16 router

had an option to configure both the UNREPLIED and ASSURED UDP lifetime values, regardless of the setting it always functioned with the UNREPLIED value at 30 seconds. This prevented any comparison tests being performed using a different mapping lifetime on this router.

The ASUS RT-N16 router also had "NAT Acceleration" (also known as CTF, or cut-through-forwarding) enabled by default. This appeared to use information in addition to the normal UNREPLIED and ASSURED UDP timeout settings and adapt the timeout based on other parameters. Because it gave inconsistent results, it was disabled for these experiments to allow a valid comparison between the keep-alive optimization algorithms.

4.9 Experimental Methods

4.9.1 Assumptions

While the algorithm itself is adaptive to changing lifetime values, for the purposes of the experiment the following conditions were satisfied.

- The UDP mapping lifetime value of the NAT devices on the route between the client and the server do not change during the course of each run, although the "ASSURED" timeout value may be different than the "UNREPLIED" timeout value.
- The configuration of the network between the client and the server does not change during the course of each run.
- The client and server are both connected to the network constantly for each run.
- The routers both adjust the NAT mapping lifetime based on incoming as well as outgoing traffic for existing connections.

4.9.2 Design

Several sets of experiments were run to evaluate the STUN Calc Keep-Alive technique and compare it to existing methods. For the experiments involving the REFINE and RUN states on the Linksys WRT-54G router, the repeat count was set to five. Initial testing with that router indicated that if the keep-alive messages are returned five times with the same timeout interval during the RUN state, then that interval would work consistently.

Optimality of Keep-Alive Interval

For the first experiment two other keep-alive optimization techniques were implemented for comparison. The first is a heavily client-based method described in [32]. It uses normal STUN binding requests and responses, and two ports on the client to make the appropriate calculations using a binary search to narrow down the address and port mapping lifetime. The second method utilizes the ability of the modified server for the STUN Calc Keep-Alive method to use a delay in sending responses. It sends a new request after each timeout, and like the previous method uses a binary search to determine the mapping lifetime. This method is similar to those described in Section 4.3.3.

In each case, one thread on the client ran the appropriate keep-alive algorithm on a connection to the server, while a second thread ran a "constant" keep-alive session on another connection. The second thread updated its keep-alive interval based on new findings from the first. The purpose of this was to simulate the normal data channel for an application and evaluate any effect from the keep-alive technique in use on the number of needed keep-alive mes-

sages over a period of time. The STUN Calc Keep-Alive algorithm was run using the geometric increment method to better match the search path taken by the other two algorithms. The test was run 400 times with the UDP mapping lifetime value set to 30 seconds. For all of the algorithms tested, an initial value of 5000 ms was used, with the STUN Calc Keep-Alive algorithm having a matching ΔT. This value was chosen to allow a good fit for a variety of NAT mapping lifetime values, as common mapping lifetimes for UDP range from 15 seconds to two minutes.

REFINE State

The second test evaluated the ability of the technique's REFINE state to find a stable value. The technique was run with the client and server on opposing sides of the Linksys WRT-54G router, which appears to be somewhat inconsistent in terms of NAT mapping lifetimes, with variations of over 100 milliseconds between sessions. The UDP mapping lifetime of this router is 90 seconds.

The algorithm was started in the REFINE state at a value known to be too large (90100 ms) and run 35 times to verify that the REFINE state was always able to stabilize on a value. The REFINE state ΔT value used was 100 ms, while the SEARCH ΔT was 50000 ms in order to minimize the "build up" time. The ΔT value is small to in order to allow small adjustments to the T value. On a router with more consistent behavior, ΔT could be set even smaller.

Single Channel

The third set of tests evaluated the performance of the STUN Calc Keep-Alive algorithm when used in a single channel situation. For this test the server

generated traffic and sent it to the client from the same port being used by the keep-alive algorithm. Tests were run both using a random traffic generator, and with four sets of real traffic data: two from Direct Democracy P2P (software) (DDP2P) and two from Skype. The traffic data was gathered using Wireshark.

Two additional tests were run setting Z_{Mult} and Z_{Max} to limit the number of times that the algorithm timed out before accepting the last known good time. These tests were done using the first DDP2P data set, which included a significant amount of network traffic noise. Each test was run 82 times using a UDP mapping lifetime of 30 seconds. The number of tests is due to time constraints experienced while running the experiments.

The tests were run with the geometric increment type and MAX_RESET_TRIES set to **10**, MIN_RESET_INTERVAL set to **5000 ms**, and RESET_DELAY set to **2000 ms**. These values were chosen based on initial tests to attempt to minimize the number of RESET messages while still finding a reasonable T value. The 2000 ms RESET_DELAY prevents extraneous messages from being sent for busy traffic sets, while retaining some precision by sending a RESET message when there is a gap in traffic.

4.9.3 Measurement

The test configuration of the STUN client keeps count of all incoming messages, sent messages, timeouts, and the duration of the session. When $UsesStarterMessage$ is set, in order to accurately gage the timeout, once the message wheel on the server queues up a new session, it sends the first message immediately to establish the beginning of the interval.

For the first set of experiments, the total number of sent and received messages, the number of timeouts, and the total time taken to determine the keep-alive interval were analyzed for each of the three optimization algorithms. The mean, median, and mode of the final keep-alive interval value found were also compared.

Measurement for the second experiment testing the REFINE state involved verifying that the algorithm was able to find a stable keep-alive value on the WRT-54G router using the REFINE state. The final keep-alive interval found and message counts were measured.

For the third set of experiments, the number of RESET messages, the keep-alive interval values found, and the time taken to gather them were the primary data points measured.

In addition, a "time unreachable" metric was also gathered. To find this value, the client kept track of all T values that resulted in a timeout, in each instance adding the new value to a sum total. At the end of the process, the client multiplied the final value of T by the number of timeouts, and subtracted that from the sum total of timeout T values. The resulting value is the total time that the client was "unreachable" from outside of the NAT. The formula is shown below:

$$T_{Unreachable} = \sum T_{TimedOut} - TimeoutCount \times T$$

With the test implementation, the timeout for the client is set to include a buffer due to the possibility of a delayed RESET message causing a server response message to arrive after a timeout. In this situation the client would

miss a message when in actuality it successfully travelled through the NAT. Since RESET_DELAY was set to 2000 ms for these experiments, the timeout buffer value was 2010 ms. The adjusted formula for "time unreachable" is as follows:

$$T_{Unreachable} = \sum T_{TimedOut} - TimeoutCount \times T + TimeoutCount \times TIMEOUT\_BUFFER$$

4.9.4 Threats to Validity

The primary threat to the validity of these results is that only two routers were used in the experiments, and they have known differing behaviors. The lack of variety in NAT lifetime mapping values is also a limiting factor. As discussed in earlier sections, there are a variety of different NAT mapping behaviors displayed depending on the router and firmware, and the NAT mapping behavior of many routers is not well documented.

In addition, these experiments were done with only one router acting as a NAT device between the client and server, whereas there could potentially be several NAT devices on the network route depending on the network configuration.

4.10 Analysis of Results

Optimality of Keep-Alive Interval

Metric	Measure	STUN Calc KA	Server Binary Search	Client Binary Search
Timeout Interval	Mean	29991.41	29989.58	29986.82
	Median	29992	29990	29987
	Mode	29992	29990	29990
Session Duration	Mean	450831.59	615215.02	572275.17
	Median	455097.5	505192	540283.5

Table 4.6: Keep-Alive Connection Thread Time Intervals (ms)

Metric	Measure	STUN Calc KA	Server Binary Search	Client Binary Search
Received	Mean	16.55	15.70	34.03
	Median	17	14	34
	Mode	17	14	33
Sent	Mean	8.22	19.80	38.66
	Median	8	18	38
	Mode	8	17	38
Timeouts	Mean	3.24	4.11	4.60
	Median	3	4	5
	Mode	3	4	5

Table 4.7: Keep-Alive Connection Thread Message Counts and Timeouts

While in theory STUN Calc Keep-Alive is more accurate than techniques that rely on a back-and-forth client ping, in terms of this experiment the tested algorithms all seem very close. The STUN Calc Keep-Alive algorithm did find the highest average keep-alive interval, however the difference between the final keep-alive interval compared to the other algorithms was smaller than the total

range of values found. The STUN Calc Keep-Alive algorithm appears to have a slightly shorter session duration overall, and the Server Binary Search included some significantly longer sessions that brought up the mean.

The Client Binary Search both sent and received the largest number of messages due to its need to establish a new binding for each timing attempt. A notable difference among the algorithms is the number of sent messages. Since in this case the search path taken by the STUN Calc Keep-Alive algorithm involved a small number of timeouts, only a small number of sent messages were needed since the client only communicates with the server to initiate or stop a session and when it experiences a timeout. Had the search path resulted in the final keep-alive interval value being closer to the original "min" value used for the search, therefore causing the client to experience more timeouts, the difference in sent messages between the STUN Calc Keep-Alive and the Server Binary Search algorithms would be smaller.

Metric	Measure	STUN Calc KA	Server Binary Search	Client Binary Search
Received	Mean	19.33	24.45	23.82
	Median	20	22	24
	Mode	20	22	24

Table 4.8: Constant Connection Thread Message Counts

There did not seem to be a significant difference in the number of messages received by the "constant" runner thread accompanying the search thread. This is most likely because while they took different search paths depending on the exact NAT mapping lifetime that the router gave their sessions, they were all doing a binary search.

REFINE State

A small series of tests was performed to verify that the client was able to determine a stable keep-alive interval value using the REFINE state 100% of the time. Once the REFINE state finished running, the new keep-alive value was verified with the RUN state in each case.

Metric	Measure	Time
Timeout Interval	Mean	88334.29
	Median	88550
	Mode	88550
Session Duration	Mean	2899027.63
	Median	2843284

Table 4.9: REFINE State Time Intervals (ms)

Metric	Measure	Count
Received	Mean	36.43
	Median	36
	Mode	36
Sent	Mean	15.43
	Median	16
	Mode	16
Timeouts	Mean	6.71
	Median	7
	Mode	7

Table 4.10: REFINE State Message Counts and Timeouts

Single Channel

The STUN client performed well with the single channel capability enabled for determining the keep-alive interval. As expected, despite the delays in sending RESET messages, the network traffic increases significantly when attempting to determine the keep-alive interval, especially with noisy data sets. The algorithm was able to converge on a keep-alive interval value in all cases in spite of the network traffic noise.

Measure	DDP2P 1	DDP2P 2	Skype 1	Skype 2	Random
Mean	29925.44	29958.48	29990.51	29990.38	29922.00
Median	29992	29990	29990	29992	29988
Mode	29994	29990	29990	29992	29988

Table 4.11: Single Channel Time Intervals (ms) Found

Metric	Measure	DDP2P 1	DDP2P 2	Skype 1	Skype 2	Random
Without TO Buffer	Mean	695.63	10086.27	10022.14	10022.47	10284.67
	Median	12	10023	10023	10020	10031
	Mode	8	10023	10023	10020	10031
With TO Buffer	Mean	4655.00	16587.75	16548.43	16722.47	12298.22
	Median	4020	16053	16053	16053	12044
	Mode	4020	16053	16053	18060	12045

Table 4.12: Single Channel Time Unreachable (ms)

In Table 4.12 above, "TO" signifies "timeout".

Using the RESET messages, the algorithm was able to find a keep-alive value similar to those found using a dedicated channel. However, the traffic noise did

affect the final keep-alive interval values found using several of the data sets.

The "time unreachable" metric is comparable among most of the traffic patterns with the exception of "DDP2P 1". For the rest of the patterns, the large value appears to be due to a timeout with an attempted keep-alive interval of 40000 ms. The actual NAT mapping lifetime is around 30000 ms, and the large difference adds significantly to the time during which the client is unreachable. With the "DDP2P 1" data set, during the majority of the tests the client reached MAX_RESET_TRIES before there was a lapse in traffic large enough for the 40000 ms attempt to time out, so only smaller intervals were attempted and able to result in a timeout.

The algorithm was able to determine similar keep-alive interval values in various network traffic conditions. The "random" traffic generator and the "DDP2P 2" data set included somewhat large gaps in traffic, while the remaining sessions contained significantly more noise in addition to some gaps of various lengths. Since the traffic used for each test was different, a comparison of the session duration among the different conditions is not appropriate.

Aside from the number of RESET messages, the other message counts remained within a reasonable range (in relation to the dual channel mode results) while operating within the different sets of traffic conditions. The "sent" messages in Table 4.13 consist of both the START and STOP messages sent to the server.

Metric	Measure	DDP2P 1	DDP2P 2	Skype 1	Skype 2	Random
Received	Mean	18.53	18.37	18.19	18.44	18.11
	Median	19	18	18	19	18
	Mode	19	18	18	19	19
Sent	Mean	8.22	8.23	8.30	8.32	9.06
	Median	8	8	8	8	9
	Mode	8	8	8	8	8
Timeouts	Mean	2.23	3.23	3.25	3.33	3.56
	Median	2	3	3	3	3
	Mode	2	3	3	3	3
Reset	Mean	40.01	32.05	54.16	45.01	37.78
	Median	40	33	55	45	37
	Mode	40	33	55	45	33

Table 4.13: Single Channel Message Counts

Measure	No Z_{Max}	$Z_{Max} = 16$	$Z_{Max} = 64$
Mean	29925.44	28024.69	29323.38
Median	29992	29375	29843
Mode	29994	29375	29843

Table 4.14: Single Channel (with Z_{Max}) Time Intervals (ms) Found

Measure	No Z_{Max}	$Z_{Max} = 16$	$Z_{Max} = 64$
Mean	877032.59	645251.52	775621.19
Median	903705	715699	788761

Table 4.15: Single Channel (with Z_{Max}) Session Durations (ms)

Using the "DDP2P 1" data set, additional tests were run with two different

Z_{Max} values. These cutoff points allowed the algorithm to return with a keep-alive interval more quickly, but at the cost of precision. This test is only relevant for this implementation, as the alternate implementation described in Section 4.4.2 is more passive and has little reason to end the search prematurely as it adjusts its activity along with the network traffic level.

4.11 Discussion and Conclusions

While the STUN Calc Keep-Alive technique was not shown to be more accurate during these experiments, when running in the SEARCH state it has been shown to provide comparable results to traditional keep-alive interval calculation techniques. Testing the technique using a client and server with a high network latency between them could possibly better illustrate any improved optimality. The number of sent messages for the STUN Calc Keep-Alive algorithm included both the START and STOP messages. This number could potentially be further reduced in some situations by having a single message that both stops the existing session and begins a new one. The current implementation uses separate messages in order to stop the existing session as quickly as possible.

The REFINE state was shown to stabilize on a keep-alive value, even with a starting time several seconds too large. The primary imagined uses of the client in the REFINE state are to maintain a keep-alive connection on routers that are inconsistent (e.g. those that adjust NAT mapping lifetimes based on external factors), or to more quickly find a new keep-alive value if a device changes networks or is otherwise confronted with a new NAT mapping lifetime with which the current keep-alive interval will not work.

While the experiment to evaluate the "Single Channel" mode of the algorithm used incoming network traffic from the server as the "noise", in practice the technique could be used with outgoing traffic as well, as an application would be aware of any outgoing traffic. The values of MAX_RESET_TRIES, MIN_RESET_INTERVAL, and RESET_DELAY chosen for the experiment were meant to strike a balance between precision and a reasonable number of RESET messages. However, the number of RESET messages turned out higher than ideal. A larger value for MIN_RESET_INTERVAL might help alleviate the issue. The original incarnation of this mode exited the search after the first time MAX_RESET_TRIES was reached. This may be desirable if a keep-alive interval is needed quickly, or if the connection has a large amount of traffic with few gaps. It can be achieved with the current algorithm by setting Z_{Max} to a value less than Z, so the $Z > Z_{Max}$ condition is met on the first occurance. Despite the larger than expected number of RESET messages, the "Single Channel" mode has been shown to be a viable option for determining a keep-alive interval, despite being sub-optimal.

The use case for this mode is narrower than that for the normal "Dual Channel" mode, but may be useful depending on the application and the NAT device. For example, many applications that utilize UDP use a single socket for multiple purposes and multiplex the data. Even if the various communication channels have different destination locations, some NAT devices may maintain a single NAT mapping lifetime for that set of connections based on the source address and port. It could also be the case that a client needs to communicate with a server that is also multiplexing data, with both using the same endpoints for multiple tasks.

Chapter 5

Conclusion

Due to the the limitations of the IPv4 internet protocol concerning the number of possible unique addresses, a technology called Network Address Translation (NAT) has been used to provide a gateway between the Internet and private intranets. This technology introduces a number of challenges to P2P protocols; challenges that will continue after the introduction of IPv6 due to the large amount of legacy equipment and knowledge. The contributions of this thesis are as follows:

- An overview of general information about NAT devices and how external facing address and port mappings are created and maintained for use with various protocols (e.g. UDP and TCP).
- Comparison of features and a timing evaluation of several protocols, including NAT-PMP, PCP, UPnP, and STUN for use creating or maintaining NAT mappings.
- A survey of existing methods to calculate an optimal keep-alive interval

value for a given network configuration.

- Proposing the STUN Calc Keep-Alive method for calculating a keep-alive interval value, including its theoretical and experimental comparison with existing techniques.

Obtaining the External Address One of the goals of this thesis was to evaluate the different methods available for creating an external address and port mapping on a NAT device. The NAT-PMP, PCP, UPnP, and STUN protocols were all discussed and weighed according to their advantages and disadvantages.

The protocols were also tested for performance when obtaining an external address and port mapping.

NAT UDP Keep-Alive Interval Optimization Method We introduce a flexible keep-alive interval optimization method for use over UDP. One of the distinguishing features is that the server sends the keep-alive messages, and the client only responds in the event of a timeout. With most existing methods, the client responds to each ping from the server to verify that the connection is still active. Depending on the number of timeouts, the method presented in this thesis results in fewer messages from the client overall, thus resulting in less network traffic and potential power savings for the client.

Having the server send keep-alive messages with its own timer instead of relying on responses from the client also allows for more consistent results, and possibly more precise results in situations where the NAT device uses a different NAT mapping lifetime for "assured" UDP sessions than for new sessions.

The T_{Max} and $ActionAfterT_{Max}$ elements of the KEEP_ALIVE_TIMER_DATA attribute structure allow a client to re-use a keep-alive interval that takes advantage of NAT devices utilizing a separate lifetime for connections in the aforementioned "assured" UDP state. Both the RUN and REFINE states of the client can use these values along with the specified ΔT to "build up" to the actual interval value needed.

The technique's REFINE state allows the client to adapt to a NAT device with an inconsistent address and port mapping lifetime or to a change in networks without having to restart the search. When the algorithm is in the RUN state, and the number of timeouts reaches a preset failure threshold, it moves to the REFINE state and begins to search backwards. This is potentially useful especially for mobile devices that change networks frequently. For devices that remain on the same network for longer periods of time, the REFINE state allows for reasonably quick adjustments of the keep-alive interval, as some NAT devices change the mapping lifetime depending on external factors such as traffic load.

This thesis also presented a variation of the STUN Calc Keep-Alive technique that works using the same connection as application data. This is achieved using RESET messages which are sent to the server to "reset" the timer for the next message in a keep-alive optimization session based on the amount of unrelated network traffic as well as multiple parameters. While the implementation tested is not optimal due to the potentially large number of RESET messages generated, an alternate implementation that takes a more passive approach is described as well.

Future Work In terms of comparing NAT traversal techniques, newer methods such as ICE could be evaluated in more detail and compared to the methods discussed for creating the address and port mapping explicitly (UPnP, PCP, and NAT-PMP), in addition to those discussed that utilize the NAT device to automatically obtain their external address and port mapping in order to initiate communication with a peer via another method (e.g. STUN). The protocols could also be evaluated when working with TCP connection mappings.

The STUN Calc Keep-Alive technique has the potential for additional functionality. For example, when the $ServerAdjustsZ$ flag is set, the server wheel assumes that K will remain constant in order to facilitate a binary search. If one wishes to have the server modify Z while also incrementing K as usual, a flag could be added to the KEEP_ALIVE_TIMER_DATA attribute structure in place of one of the reserved bits. Work could also be done to utilize the "build up" functionality while the client is in the SEARCH state.

For the "Single Channel" mode, one could research more into reducing the number of RESET messages sent, perhaps by more intelligently estimating the delay before the next keep-alive message from the server and dynamically adjusting the MIN_RESET_INTERVAL or RESET_DELAY values. In addition, the alternate implementation for the Single Channel mode could be implemented and tested. In theory that is a better solution as it does not generate extra traffic with RESET messages, yet it allows the client to eventually find the NAT mapping lifetime.

In addition, all variations of the STUN Calc Keep-Alive technique could be tested on various routers to verify that it works in as many situations as possible. NAT devices vary greatly in their behavior for maintaining the address and port mappings for UDP connections, so testing the algorithm on a variety of devices is advantageous.

Concluding Remarks Traversing NATs presents a challenge for many pieces of software, especially P2P applications, as NAT devices make it difficult to determine one's own external facing address in order to communicate with others. Fortunately, many options exist to mitigate these issues. This thesis explores some of those options, including protocols for explicitly creating and maintaining an external facing address and port mapping, as well as other protocols that are used to obtain one's external facing address and set up communication via another method (e.g. UDP hole punching). The STUN Calc Keep-Alive technique is also introduced, meant to be an adaptive algorithm that can respond to changes in networks or even NAT mapping lifetime changes within the same device. The REFINE state allows adjustments to be made to the keep-alive interval when a timeout is encountered without restarting the search for a new interval, while the RUN state simply maintains a keep-alive session with regularly timed messages. Given the variation and unpredictability in mapping lifetime behaviors among NAT devices, and the increasing number of mobile devices and P2P applications in use, the ability to generate keep-alive messages at an efficient interval will most likely remain a need for the foreseeable future.

Appendix A

STUN Calc Keep-Alive Pseudo-Code (Server)

This chapter contains more detailed pseudo-code for the various procedures and other pieces of functionality described in Section 4.4.2 for the Server.

A.1 Server Controller

```
1  when STUN_CALC_KA_REQUEST(KeepAliveTimerData ka) from ClientAddress do
2      INPUT: ⟨ ActionAfterT_Max, K, ID, UsesStarterMessage, ServerAdjustsZ,
       IncrementType, Direction, RepeatCount, Z, Z_Mult, T, T_Max, ΔT, ΔTAfterT_Max ⟩
3
4      if ID not in SessionHashTable then
           // Create a session based on the request parameters
5          Session s = new Session( ka )
           // Initialize other state variables.
6          s.K_l ← s.K
7          s.InitialK ← s.K
8          s.IsFirstSessionMessage ← TRUE
9          s.OriginalΔT ← s.ΔT
10         s.OriginalΔTAfterT_Max ← s.ΔTAfterT_Max
11         s.T_MaxWasReached ← FALSE
12         s.InnerRepeatCount ← 0
13         s.OuterRepeatCount ← 0
14
15         SessionHashTable.add(⟨ClientAddress, ClientPort, ID⟩, Session)
16
           // The Wheel will reference the SessionHashTable using
              ⟨ClientAddress, ClientPort, ID⟩ for the relevant information.
17         Wheel.add( s )
18
19     InterruptWheel()
20 when STUN_CALC_KA_STOP_REQUEST(ID) from ClientAddress do
21     if ID in SessionHashTable then
22         SessionHashTable.remove(ID)
23         Wheel.remove(ID)
24         InterruptWheel()
25 when STUN_CALC_KA_RESET_REQUEST(ID) from ClientAddress do
26     if ID in SessionHashTable then
27         Wheel.reset(ID)
28         InterruptWheel()
```

Algorithm 5: Server Controller Behavior

A.2 Server Message Wheel

```
   Input: ID, Session s
1  procedure add() do
2  |  if s.UsesStarterMessage = TRUE then
   |  |  // The first send time is immediate to establish the address and port
   |  |     binding.
3  |  |  s.NextSendTime ← SystemTime()
   |  // Otherwise it is set up in the main wheel loop.
4  |  MessageQueue.add( s )
   Input: ID
5  procedure remove() do
6  |  MessageQueue.remove(ID)
   Input: ID
7  procedure reset() do
8  |  Session s ← SessionHashTable.get(ID)
9  |  s.NextSendTime ← SystemTime() + s.T
```

Algorithm 6: Server Wheel Behavior (Session Management)

```
1  when Wheel Session s do
2      SendRealKToClient ← TRUE
3      if s.NextSendTime > SystemTime() then
4          pause the thread until the correct time unless interrupted with a new session
5      if thread is interrupted then
6          MessageQueue.add(s)
7          s ← MessageQueue.pop()
8          CONTINUE
9      if session has been removed then
10         CONTINUE
11     send to(ClientAddress, STUN_CALC_KA_RESPONSE(ID, K_l, T, ΔT, Direction,
       Z, ServerAdjustsZ))
12
13     if s.UsesStarterMessage = TRUE AND s.IsFirstSessionMessage = TRUE then
14         ShouldCalcNextTime ← FALSE
15     else
16         ShouldCalcNextTime ← TRUE
17     if s.IsFirstSessionMessage = TRUE AND s.ServerAdjustsZ = FALSE then
           // Initial adjustment for ΔT if ServerAdjustsZ is FALSE.
18         if s.T_Max WasReached = TRUE then
19             s.ΔTAfterT_Max ← s.OriginalΔTAfterT_Max / s.Z
20         else
21             s.ΔT ← s.OriginalΔT / s.Z
22
23     if s.T ≠ s.T_Max AND ShouldCalcNextTime = TRUE then
24         CalcNextTime( s, SendRealKToClient )
25
26     s.NextSendTime ← s.T + SystemTime()
27     if s.ServerAdjustsZ = FALSE and ShouldCalcNextTime = TRUE then
28         s.K ← s.K + 1
29
30     if SendRealKToClient = TRUE then
31         K_l ← s.K
32
33     if s.T_Max > 0 AND s.T ≥ s.T_Max AND ShouldCalcNextTime = TRUE then
34         AdjustAfterMaxTime( s )
35
36     MessageQueue.add( s )
```

Algorithm 7: Server Wheel Behavior (Message Management)

```
1  procedure CalcNextTime(Session s, Boolean SendRealKToClient) do
2      if s.MaxRepeatCount > 0 AND s.InnerRepeatCount < s.MaxRepeatCount AND
       s.NextAction = MAINTAIN_CURRENT then
3          s.InnerRepeatCount ← s.InnerRepeatCount + 1
4          SendRealKToClient ← FALSE
5      else
6          if s.MaxRepeatCount > 0 then
7              K_l ← s.OuterRepeatCount
8              SendRealKToClient ← FALSE
9          else
10             K_l ← s.K
11         s.InnerRepeatCount ← 0
12         s.OuterRepeatCount ← s.OuterRepeatCount + 1
           // Adjust Z using the appropriate ΔT value.
13         if s.ServerAdjustsZ = TRUE then
14             s.OuterRepeatCount ← 1
15             K_l ← s.InitialK
16             SendRealKToClient ← FALSE
17             s.Z ← s.Z × s.Z_Mult
18             if s.T_Max WasReached = TRUE then
19                 s.ΔTAfterT_Max ← s.OriginalΔTAfterT_Max / s.Z
20             else
21                 s.ΔT ← s.OriginalΔT / s.Z
           // Select the ΔT value for the new calculation for T.
22         if s.T_Max WasReached = TRUE then
23             ΔT_l ← s.ΔTAfterT_Max
24         else
25             ΔT_l ← s.ΔT
26         if ΔT_l > 0 then
27             switch s.IncrementType do
28                 case LINEAR
29                     IntervalChange ← K_l × ΔT_l
30                 case GEOMETRIC
31                     IntervalChange ← 2^K_l × ΔT_l
               // Modify T with the new value based on the session direction.
32             switch s.Direction do
33                 case FORWARD
34                     s.T ← s.T + IntervalChange
35                 case BACKWARD
36                     s.T ← s.T − IntervalChange
```

Algorithm 8: CalcNextTime Procedure

```
1  procedure AdjustAfterMaxTime(Session s) do
2      s.T_Max WasReached ← TRUE
3      s.OuterRepeatCount ← 0
4      s.T ← s.MaxT
5      s.NextSendTime ← SystemTime() + s.T
       // Reset max time.
6      s.T_Max ← 0
7      SendRealKToClient ← FALSE
8      K_l ← s.OuterRepeatCount
9      switch s.NextAction do
10         case MAINTAIN_CURRENT
               // No change.
11         case STAY_CONSTANT
12             s.K ← GetBaseK( s )
13         case CHANGE_TO_FORWARD
14             s.K ← GetBaseK( s )
15             s.Direction ← FORWARD
16         case CHANGE_TO_BACKWARD
17             s.K ← GetBaseK( s )
18             s.Direction ← BACKWARD
19     s.NextAction ← MAINTAIN_CURRENT
```

Algorithm 9: AdjustAfterMaxTime Procedure

```
1  Function GetBaseK(Session s) : Number is
2      switch s.IncrementType do
3          case LINEAR
4              K_Base ← 1
5          case GEOMETRIC
6              K_Base ← 0
7      RETURN K_Base
```

Algorithm 10: GetBaseK Function

Appendix B

STUN Calc Keep-Alive Pseudo-Code (Client)

This chapter contains more detailed pseudo-code for the various procedures and other pieces of functionality described in Section 4.4.2 for the Client.

B.1 Initialization

```
Input: ServerAddress, SearchParams, RunParams, RefineParams, T, State,
       UsesStarterMessage
1  INPUT: SearchParams ⟨ ΔT, ΔT_Min, Z, Z_Max, Z_Mult, IncrementType ⟩
2  INPUT: RunParams ⟨ FailureTolerance, RepeatCount ⟩
3  INPUT: RefineParams ⟨ ΔT, Z, Z_Mult, PingsPerAttempt ⟩
4  procedure Initialize() do
5      t ← 0                                          // Identifier for the session ID
6      ID ← ID_t                                                  // The new session ID
7      T ← T                                          // The keep-alive interval value
8      switch State do
9          case SEARCH
10             Z ← SearchParams.Z
11             T_Max ← 0 // Not used in this state
12             ActionAfterT_Max ← STAY_CONSTANT
13             Direction ← FORWARD
14             RepeatCount ← 0
               // For binary search, this changes after the first timeout
15             ServerAdjustsZ ← FALSE
               // Used only for the current "single channel" implementation
16             ResetCount ← 0
17         case RUN
18             Z ← 1
19             T_Max ← T
20             T ← SearchParams.ΔT
21             ActionAfterT_Max ← STAY_CONSTANT
22             Direction ← FORWARD
23             RepeatCount ← RunParams.RepeatCount
24             ServerAdjustsZ ← FALSE // Not used in this state
               // Used to keep count for termination if a max number of runs is
                  specified
25             RunCount ← 0
26         case REFINE
27             Z ← RefineParams.Z
28             T_Max ← T
29             T ← SearchParams.ΔT
30             ActionAfterT_Max ← CHANGE_TO_BACKWARD
31             Direction ← FORWARD
32             ServerAdjustsZ ← FALSE // Not used in this state
33             RepeatCount ← RefineParams.PingsPerAttempt
               // Tracks the number of "pings" for the current T value
34             RefineCount ← 0
               // Used as a multiplier when calculating a new T value
35             RefineTimeoutCount ← 0
               // Set to TRUE once a good T value is found
36             RefineMinValueFound ← FALSE
```

Algorithm 11: Client Initialization (Part One)

```
1    ...
     // Used to help calculate the next timeout interval in the REFINE state
     // Set to TRUE after the maximum number of pings is reached
2    RefinePingsReached ← FALSE
     // Indicates if the time changed after RepeatCount was reached
3    TimeChanged ← FALSE
     // Indicates if a ‘‘build up’’ cycle was just completed
4    JustCompletedBuildUpCycle ← FALSE
5
     // The number of session messages to ignore before acting
6    if UsesStarterMessage then
7        BaseMessageCount ← 1
8    else
9        BaseMessageCount ← 0
10
     // The starting iteration value
11   switch SearchParams.IncrementType do
12       case LINEAR
13           K ← 1
14       case GEOMETRIC
15           K ← 0
16
     // The increment method to use for SEARCH and ‘‘build up’’
17   IncrementType ← SearchParams.IncrementType
     // The last known good T value
18   LastGoodT ← 0
     // The total number of timeouts
19   TimeoutCount ← 0
     // The number of received messages for the session
20   SessionMessageCount ← 0
     // The total number of received messages.
21   TotalMessageCount ← 0
     // The default is to always start in "build up" mode
22   BuildingUpToTime ← TRUE
     // Exit the optimization process after the next iteration
23   ExitOnNextMessage ← FALSE
     // Used to avoid an infinite loop if last attempted T value times out
24   LastAttemptStarterSkipped ← FALSE
25
26   StartNewSession ← TRUE
27   goto ClientLoopStart
```

Algorithm 12: Client Initialization (Part Two)

B.2 Main Loop

```
1  procedure ClientLoopStart do
2      if State changed then
           // In this instance Initialize() should avoid calling this procedure
              again, as it will simply resume after Initialize() finishes
3          Initialize()
4
5      if ExitOnNextMessage = TRUE then
6          if SessionMessageCount > BaseMessageCount OR
           LastAttemptStarterSkipped = TRUE then
7              sendto(ServerAddress, STUN_CALC_KA_STOP_REQUEST(ID))
8              T ← LastGoodT
9              State ← RUN
10             BREAK
11         else
               // This flag prevents an infinite loop from occuring if the last
                  T value times out.
12             LastAttemptStarterSkipped ← TRUE
```

Algorithm 13: ClientLoopStart (Part One)

```
1   ...
    // The calls to build the STUN_CALC_KA_REQUEST objects omit the SizeType
       in this description.  The ordering of the arguments otherwise
       corresponds to those in Figure 4.1.
2   if StartNewSession = TRUE then
3       t ← t + 1
4       ID ← ID_t
5       SessionMessageCount ← 0
6   switch State do
7       case SEARCH
            // As an alternative to checking Z_Max, check for exit based on
               SearchParams.ΔT_Min
8           if SearchParams.ΔT / Z ≤ SearchParams.ΔT_Min then
9               ExitOnNextMessage ← TRUE
10
11          if SearchParams.Z_Max > 0 AND Z > SearchParams.Z_Max then
12              T ← LastGoodT
13              State ← RUN
14              BREAK
15          BuildingUpToTime ← FALSE
16          if StartNewSession = TRUE then
17
18              request ← STUN_CALC_KA_REQUEST(Version, STAY_CONSTANT,
                    K, ID, UsesStarterMessage, ServerAdjustsZ, IncrementType,
                    FORWARD, 0, Z, SearchParams.Z_Mult, T, 0, SearchParams.ΔT, 0)
19
20              sendto(ServerAddress, request)
21              StartNewSession ← FALSE
22          TimeOut ← CalculateNextTimeout()
23          alarm timeout( TimeOut )
24      case RUN
25          if StartNewSession = TRUE then
26              BuildingUpToTime ← TRUE
27              T_Max ← T
28              T ← SearchParams.ΔT
29
30              request ← STUN_CALC_KA_REQUEST(Version, STAY_CONSTANT,
                    K, ID, UsesStarterMessage, ServerAdjustsZ, IncrementType,
                    FORWARD, RunParams.RepeatCount, Z, 1, T, T_Max,
                    SearchParams.ΔT, 0)
31
32              sendto(ServerAddress, request)
33              StartNewSession ← FALSE
34          TimeOut ← CalculateNextTimeout()
35          alarm timeout( TimeOut )
```

Algorithm 14: ClientLoopStart (Part Two)

```
1   ...
2   case REFINE
3       if StartNewSession = TRUE then
4           BuildingUpToTime ← TRUE
5           T_Max ← T
6           T ← SearchParams.ΔT
7           if FirstRefineTime = TRUE then
                // The current interval value already timed out in the RUN
                   state.
8               T_Max ← T_Max − RefineParams.ΔT / Z
9               FirstRefineTime ← FALSE
10          if ActionAfterT_Max = STAY_CONSTANT then
11              ActionAfterT_Max ← CHANGE_TO_BACKWARD
12
13          request ← STUN_CALC_KA_REQUEST(Version, ActionAfterT_Max, K, ID,
                UsesStarterMessage, ServerAdjustsZ, IncrementType, FORWARD,
                RefineParams.PingsPerAttempt, 1, 1, T, T_Max, SearchParams.ΔT,
                RefineParams.ΔT)
14
15          sendto(ServerAddress, request)
16          StartNewSession ← FALSE
17      TimeOut ← CalculateNextTimeout()
18      alarm timeout( TimeOut )
```

Algorithm 15: ClientLoopStart (Part Three)

```
1  procedure CalculateNextTimeout() do
2      switch State do
3          case SEARCH
               // The client sets the time for the start of the session, so in
                  this case T is known.
4              if SessionMessageCount ≤ BaseMessageCount then
5                  TimeOut ← T + TIMEOUT_BUFFER
               // Otherwise the next value of T must be calculated.
6              else
7                  switch IncrementType do
8                      case LINEAR
9                          Ω_K ← K
10                     case GEOMETRIC
11                         Ω_K ← 2^K
12                 if ServerAdjustsZ = TRUE then
13                     Z_l ← Z × SearchParams.Z_Mult
14                 else
15                     Z_l ← Z
16                 TimeOut ← T + Ω_K × SearchParams.ΔT / Z_l + TIMEOUT_BUFFER
17         case RUN
18             if SessionMessageCount ≤ BaseMessageCount then
19                 TimeOut ← T + TIMEOUT_BUFFER
20             else
21                 switch IncrementType do
22                     K_l ← SessionMessageCount − BaseMessageCount − 1
23                     case LINEAR
24                         Ω_{K_l} ← K_l
25                     case GEOMETRIC
26                         Ω_{K_l} ← 2^{k_l}
                   // If currently ‘‘building up’’, a ΔT is being used and Ω_{k_l}
                      is significant.
27                 if BuildingUpToTime = TRUE then
28                     TimeOut ← T + Ω_{K_l} × SearchParams.ΔT / Z + TIMEOUT_BUFFER
29                     if TimeOut − TIMEOUT_BUFFER > T_Max then
30                         TimeOut ← T_Max + TIMEOUT_BUFFER
                   // Otherwise the Ω_{K_l} is not needed since the time is
                      constant.
31                 else
32                     TimeOut ← T + TIMEOUT_BUFFER
```

Algorithm 16: CalculateNextTimeout Procedure (Part One)

```
1   ...
    // While this is the structure used for the test implementation,
       depending on the application a simpler implementation could be used.
2   case REFINE
3       if SessionMessageCount ≤ BaseMessageCount then
4           TimeOut ← T + TIMEOUT_BUFFER
5       else
            // When ''building up'', the SEARCH state ΔT is used.
6           if BuildingUpToTime = TRUE then
7               switch IncrementType do
8                   K_l ← SessionMessageCount − BaseMessageCount − 1
9                   case LINEAR
10                      Ω_{K_l} ← K_l
11                  case GEOMETRIC
12                      Ω_{K_l} ← 2^{k_l}
13              TimeOut ← T + Ω_{K_l} × SearchParams.ΔT + TIMEOUT_BUFFER
14              if TimeOut − TIMEOUT_BUFFER > T_Max then
15                  TimeOut ← T_Max + TIMEOUT_BUFFER
                // Otherwise the REFINE state ΔT is used.
16              else
17                  if (RefineCount < RefineParams.PingsPerAttempt - 1 AND
                    RefinePingsReached = FALSE) OR JustCompletedBuildUpCycle =
                    TRUE then
18                      TimeOut ← T + TIMEOUT_BUFFER
19                  else if RefinePingsReached = TRUE then
20                      RefinePingsReached ← FALSE
21                      if TimeChanged = FALSE then
22                          switch IncrementType do
23                              case LINEAR
24                                  Ω_K ← K
25                              case GEOMETRIC
26                                  Ω_K ← 2^k
27                          TimeOut ←
                            T + Ω_K × RefineParams.ΔT + TIMEOUT_BUFFER
28                      else
29                          TimeChanged ← FALSE
30                          TimeOut ← T + TIMEOUT_BUFFER
31                  else
32                      switch IncrementType do
33                          case LINEAR
34                              Ω_{K+1} ← K + 1
35                          case GEOMETRIC
36                              Ω_{K+1} ← 2^{k+1}
37                      TimeOut ← T+Ω_{K+1}×RefineParams.ΔT+TIMEOUT_BUFFER
```

Algorithm 17: CalculateNextTimeout Procedure (Part Two)

B.3 Message Handling

```
1  when STUN_CALC_KA_RESPONSE(CalcKaResponse r ⟨ ID, K, T, ΔT, Direction, Z,
   ServerAdjustsZ ⟩) from ServerAddress do
2      if not correct message type then
3          CONTINUE
4      if not correct (ID) then
           // This is an old session, so alert the server to terminate it.
5          sendto(ServerAddress, STUN_CALC_KA_STOP_REQUEST(ID))
6          CONTINUE
7
8      TotalMessageCount ← TotalMessageCount + 1
9      SessionMessageCount ← SessionMessageCount + 1
10     K ← r.K
11
12     if SessionMessageCount ≤ BaseMessageCount AND T ≠ 0 then
           // Take no action.  Message still counts towards total.
13     else
14         switch State do
15             case SEARCH OR RUN
16                 Z ← r.Z
17                 LastGoodT ← r.T
18                 T ← LastGoodT
19                 if State = RUN then
20                     if BuildingUpToTime = TRUE then
21                         if T ≥ T_Max then
22                             BuildingUpToTime ← FALSE
23                             RunCount ← RunCount + 1
24                     else
25                         RunCount ← RunCount + 1
26                         if RunParams.RepeatCount > 0 AND
                           RunCount ≥ RunParams.RepeatCount then
27                             sendto(ServerAddress,
                               STUN_CALC_KA_STOP_REQUEST(ID))
28                             EXIT
```

Algorithm 18: Incoming Message Handling (Part One)

```
1        ... case REFINE
2            Direction ← r.Direction
3            if BuildingUpToTime = TRUE then
4                T ← r.T
5                if T ≥ T_Max then
6                    JustCompletedBuildUpCycle ← TRUE
7                    BuildingUpToTime ← FALSE
8                    RefineCount ← RefineCount + 1
9            else
10               RefineCount ← RefineCount + 1
11               if RefineCount ≥ RefineParams.PingsPerAttempt then
12                   if T ≠ r.T then
13                       TimeChanged ← TRUE
14                   JustCompletedBuildUpCycle ← FALSE
15                   RefinePingsReached ← TRUE
16                   RefineCount ← 0
17                   LastGoodT ← r.T
18                   T ← LastGoodT
19                   if Direction = BACKWARD then
20                       RefineMinValueFound ← TRUE
21                       RefineTimeoutCount ← 0
22                       Z ← Z × RefineParams.Z_Mult
23                       T ← T + RefineParams.ΔT / Z
24                       K ← 0
25                       Direction ← FORWARD
26                       ActionAfterT_Max ← CHANGE_TO_FORWARD
27                       sendto(ServerAddress,
                         STUN_CALC_KA_STOP_REQUEST(ID))
28                       StartNewSession ← TRUE
29                       goto ClientLoopStart
30               else
31                   if T ≠ r.T then
                         // Server updated the time, so reset local iteration
                            count.
32                       RefineCount ← 0
33                       T ← r.T
34
35       StartNewSession ← FALSE
36       goto ClientLoopStart
```

Algorithm 19: Incoming Message Handling (Part Two)

B.4 Timeout Handling

```
1  when TIMEOUT do
2      sendto(ServerAddress, STUN_CALC_KA_STOP_REQUEST(ID))
3      TimeoutCount ← TimeoutCount + 1
4      switch State do
5          case SEARCH
6              Z ← Z × SearchParams.Z_Mult
7              if ServerAdjustsZ = TRUE then
8                  if SessionMessageCount > BaseMessageCount then
9                      Z ← Z × SearchParams.Z_Mult
10             else
11                 ServerAdjustsZ ← TRUE
12             switch IncrementType do
13                 case LINEAR
14                     Ω_K ← K
15                 case GEOMETRIC
16                     Ω_K ← 2^K
17             T ← LastGoodT + Ω_K × SearchParams.ΔT / Z
18         case RUN
19             if TimeoutCount / (TotalMessageCount+1) > RunParams.FailureTolerance then
20                 if BuildingUpToTime = TRUE AND T < T_Max then
                       // Set T to the maximum so that REFINE can start from that
                          point.
21                     T ← T_Max
22                 FirstRefineTime ← TRUE
23                 State ← REFINE
```

Algorithm 20: Timeout Handling (Part One)

```
1          ...
2          case REFINE
3              RefineTimeoutCount ← RefineTimeoutCount + 1
4              RefineCount ← 0
5              K ← 0
6              if Direction = BACKWARD then
7                  Direction ← FORWARD
8                  ActionAfterT_Max ← CHANGE_TO_BACKWARD
9                  T ← T − RefineTimeoutCount × RefineParams.ΔT / Z
10                 LastGoodT ← T
11             else if Direction = FORWARD then
12                 if BuildingUpToTime = TRUE AND RefineMinValueFound =
                   FALSE then
13                     Direction ← FORWARD
14                     ActionAfterT_Max ← CHANGE_TO_BACKWARD
15                     T ← T_Max
16                     T ← T − RefineTimeoutCount × RefineParams.ΔT / Z
17                     LastGoodT ← T
18                 else
19                     T ← LastGoodT
20                     State ← RUN
21
22     StartNewSession ← TRUE
23     goto ClientLoopStart
```

Algorithm 21: Timeout Handling (Part Two)

Bibliography

[1] Understanding universal plug and play. White paper, Microsoft, June 2000.

[2] UPnP device architecture v1.1. Technical report, UPnP Forum, October 2008.

[3] Oskar Andreasson. "UDP Connection State". `http://www.iptables.info/en/connection-state.html#UDPCONNECTIONS`, 2008. [Online; Accessed: 22-Mar-2015].

[4] Stephen W. Babin. Method of pausing keep-alive messages and roaming for virtual private networks on handheld devices to save battery power, March 2010. U.S. Classification 370/318, 455/574; International Classification H04B1/38; Cooperative Classification H04L29/12471, H04L67/145, H04L69/28, H04L61/2553, Y02B60/50; European Classification H04L61/25A6A, H04L29/12A4A6A, H04L29/06T.

[5] Salman Abdul Baset, Joshua Reich, Jan Janak, Pavel Kasparek, Vishal Misra, Dan Rubenstein, and Henning Schulzrinnne. How green is IP-telephony? In *Proceedings of the first ACM SIGCOMM workshop on Green networking*, pages 77–84. ACM, 2010.

[6] Mohamed Boucadair, Reinaldo Penno, and Dan Wing. Universal plug and play (UPnP) internet gateway device - port control protocol interworking function (IGD-PCP IWF). RFC 6970, RFC Editor, July 2013.

[7] Mohamed Boucadair, Paul Selkirk, Reinaldo Penno, Dan Wing, and Stuart Cheshire. Port control protocol (PCP). Request for Comments 6887, RFC Editor, April 2013.

[8] Patrick G. Brown and Randall T. Kunkel. Adaptible keepalive for enterprise extenders, October 2007.

[9] Fabian E. Bustamante and Yi Qiao. Friendships that last: Peer lifespan and its role in P2p protocols. In *Web content caching and distribution*, pages 233–246. Springer, 2004.

[10] Gonzalo Camarillo, Oscar Novo, and Simon Perreault. Traversal using relays around NAT (TURN) extension for IPv6. Request for Comments 6156, RFC Editor, April 2011.

[11] Samir Chatterjee, Bengisu Tulu, Tarun Abhichandani, and Haiqing Li. SIP-based enterprise converged networks for voice/video-over-IP: implementation and evaluation of components. *Selected Areas in Communications, IEEE Journal on*, 23(10):19211933, 2005.

[12] E.P. Duarte, K.V. Cardoso, M.O.M.C. de Mello, and J.G.G. Borges. Transparent communications for applications behind NAT/firewall over any transport protocol. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 935–940, December 2011.

[13] L. DAcunto, J. A. Pouwelse, and H. J. Sips. A measurement of NAT and firewall characteristics in peer-to-peer systems. In *Proc. 15-th ASCI Conference*, volume 5031, pages 1–5. Advanced School for Computing and Imaging (ASCI), 2009.

[14] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-Peer Communication Across Network Address Translators. *CoRR*, abs/cs/0603074, March 2006. arXiv: cs/0603074.

[15] Saikat Guha and Paul Francis. Characterization and measurement of TCP traversal through NATs and firewalls. In *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, IMC '05, page 1818, Berkeley, CA, USA, 2005. USENIX Association.

[16] Saikat Guha, Yutaka Takeda, and Paul Francis. NUTSS: A SIP-based approach to UDP and TCP network connectivity. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, FDNA '04, page 4348, New York, NY, USA, 2004. ACM.

[17] H. Haverinen, J. Siren, and P. Eronen. Energy Consumption of Always-On Applications in WCDMA Networks. In *Vehicular Technology Conference, 2007. VTC2007-Spring. IEEE 65th*, pages 964–968, April 2007.

[18] Henrik Levkowetz and Sami Vaarala. Mobile IP Traversal of Network Address Translation (NAT) Devices. Request for Comments 3519, RFC Editor, April 2003.

[19] Shai Herzog. Cost reduction of NAT connection state keep-alive, October 2013. U.S. Classification 370/395.2, 370/401, 370/465; International Classification H04L12/28; Cooperative Classification H04W52/0229, H04W92/02, H04W76/045, H04W48/08, H04L61/25, H04W28/06, H04L29/1233, H04L69/28, Y02B60/50.

[20] Shai Herzog, Rashid Qureshi, Jorge Raastroem, Xuemei Bao, Rajeev Bansal, Qian Zhang, and Scott Michael Bragg. Determining an efficient keep-alive interval for a network connection, February 2013. U.S. Classification 709/228, 370/241, 370/251, 370/401, 370/465, 709/224, 709/203; International Classification G06F15/16; Cooperative Classification H04L65/1066, H04L69/28, H04L69/163, H04L67/2842, H04L69/16, H04L67/14, H04L67/145, H04L67/141, H04L67/28.

[21] S. Holzapfel, M. Wander, A Wacker, L. Schwittmann, and T. Weis. A new protocol to determine the NAT characteristics of a host. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1651–1658, May 2011.

[22] Christer Holmberg Homberh, Christer and Ericsson. Indication of support for keep-alive. Request for Comments 6223, RFC Editor, April 2011.

[23] Geoff Huston. Anatomy: A look inside network address translators. *The Internet Protocol Journal*, 7(3):232, 2004.

[24] Seppo Htnen, Aki Nyrhinen, Lars Eggert, Stephen Strowes, Pasi Sarolahti, and Markku Kojo. An experimental study of home gateway characteristics. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 260–266. ACM, 2010.

[25] Cullen Jennings and Francois Audet. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. Request for Comments 4787, RFC Editor, January 2007.

[26] Cullen Jennings and Rohan Mahy. Managing client initiated connections in the session initiation protocol (SIP). Request for Comments 5626, RFC Editor, October 2009.

[27] Jonathan Rosenberg, Ari Keranen, Bruce B. Lowekamp, and Adam Roach. TCP Candidates with Interactive Connectivity Establishment (ICE). Request for Comments 6544, RFC Editor, March 2012.

[28] Dan Kegel, Pyda Srisuresh, and Bryan Ford. State of peer-to-peer (P2P) communication across network address translators (NATs). Request for Comments 5128, RFC Editor, March 2008.

[29] Marc Krochmal and Stuart Cheshire. NAT port mapping protocol (NAT-PMP). Request for Comments 6886, RFC Editor, April 2013.

[30] K. Kuramochi, T. Kawamura, and K. Sugahara. NAT traversal for pure P2P e-learning system. In *Third International Conference on Internet and Web Applications and Services, 2008. ICIW '08*, pages 358–363, June 2008.

[31] Jae Woo Lee, Roberto Francescangeli, Jan Janak, Suman Srinivasan, Salman A. Baset, Henning Schulzrinne, Zoran Despotovic, and Wolfgang Kellerer. NetSerV: active networking 2.0. In *Communications Workshops (ICC), 2011 IEEE International Conference on*, pages 1–6. IEEE, 2011.

[32] Bruce B. Lowekamp and Derek C. MacDonald. NAT behavior discovery using STUN. Request for Comments 5780, RFC Editor, May 2010.

[33] X. Marjou, A. Sollaud, and France Telecom Orange. Application mechanism for keeping alive the NAT mappings associated with RTP/RTP control protocol (RTCP) flows. Technical report, RFC 6263, June, 2011.

[34] Philip Matthews, Rohan Mahy, and Jonathan Rosenberg. Traversal using relays around NAT (TURN): relay extensions to session traversal utilities for NAT (STUN). Request for Comments 5766, RFC Editor, April 2010.

[35] A. Muller, N. Evans, C. Grothoff, and S. Kamkar. Autonomous NAT traversal. In *2010 IEEE Tenth International Conference on Peer-to-Peer Computing (P2P)*, pages 1–4, August 2010.

[36] V. Oliveira, A. Pina, and T. Sa. Simple peer messaging for remote user domains interconnection. In *2012 International Conference on High Performance Computing and Simulation (HPCS)*, pages 315–321, July 2012.

[37] Simon Perreault and Jonathan Rosenberg. Traversal using relays around NAT (TURN) extensions for TCP allocations. Request for Comments 6202, RFC Editor, November 2010.

[38] Richard Price and Peter Tino. Adapting to NAT timeout values in p2p overlay networks. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–6. IEEE, 2010.

[39] Richard Price, Peter Tio, and Georgios Theodoropoulos. Still alive: Extending keep-alive intervals in p2p overlay networks. *Mobile Networks and Applications*, 17(3):378–394, 2012.

[40] Jonathan Rosenberg. Interactive connectivity establishment (ICE): A methodology for network address translator (NAT) traversal for offer/answer protocols. Request for Comments 5245, RFC Editor, April 2010.

[41] Jonathan Rosenberg, Rohan Mahy, Christian Huitema, and Joel Weinberger. STUN - simple traversal of UDP through network address translators. Request for Comments 3489, RFC Editor, March 2003.

[42] D. Seah, Wai Kay Leong, Qingwei Yang, B. Leong, and A. Razeen. Peer NAT proxies for peer-to-peer games. In *2009 8th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6, November 2009.

[43] Pyda Srisuresh and Matt Holdrege. IP network address translator (NAT) terminology and considerations. Request for Comments 2663, RFC Editor, August 1999.

[44] Pyda Srisuresh and Matt Holdrege. Protocol complications with the IP network address translator. Request for Comments 3027, RFC Editor, January 2001.

[45] Pyda Srisuresh, Senthil Sivakumar, Kaushik Biswas, Bryan Ford, and Saikat Guha. Behavioral Requirements for TCP. Request for Comments 5382, RFC Editor, October 2008.

[46] Barbara Stark, Matthew Schmitz, Mark Baugher, Warrier Ulhas, Prakash Iyer, Victor Lortz, Cathy Chan, Mika Saaran, Erwan Nedellec, Fabrice Fontaine, and Dongshin Jung. WANIPConnection:2 service - standardized DCP. Technical report, UPnP Forum, September 2010.

[47] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202. ACM, 2006.

[48] Daniel Stutzbach, Reza Rejaie, and Subhabrata Sen. Characterizing unstructured overlay topologies in modern P2p file-sharing systems. *Networking, IEEE/ACM Transactions on*, 16(2):267–280, 2008.

[49] H. Suzuki, Y. Goto, and A. Watanabe. External dynamic mapping method for NAT traversal. In *International Symposium on Communications and Information Technologies, 2007. ISCIT '07*, pages 723–728, October 2007.

[50] Zoltn Turnyi, Andrs Valk, and Andrew T. Campbell. 4+4: An architecture for evolving the internet address space back toward transparency. *SIGCOMM Comput. Commun. Rev.*, 33(5):4354, October 2003.

[51] A. Wacker, G. Schiele, S. Holzapfel, and T. Weis. A NAT traversal mechanism for peer-to-peer networks. In *Eighth International Conference on Peer-to-Peer Computing , 2008. P2P '08*, pages 81–83, September 2008.

[52] Bo Wang, Xiangmin Wen, Sun Yong, and Zheng Wei. A novel NAT traversal mechanism in the heterogeneous environment. In *Eighth IEEE/ACIS International Conference on Computer and Information Science, 2009. ICIS 2009*, pages 161–165, June 2009.

[53] Ulhas Warrier, Prakesh Iyer, Frederic Pennerath, and Gert Marynissen. WANIPConnection:1 Service - Standardized DCP. Technical report, UPnP Forum, November 2001.

[54] Dan Wing, Philip Matthews, Jonathan Rosenberg, and Rohan Mahy. Session traversal utilities for (NAT) (STUN). Request for Comments 5389, RFC Editor, October 2008.

[55] Pinggai Yang, Jun Li, Jun Zhang, Hai Jiang, Yi Sun, and E. Dutkiewicz. SMBR: A novel NAT traversal mechanism for structured peer-to-peer communications. In *2010 IEEE Symposium on Computers and Communications (ISCC)*, pages 535–539, June 2010.

[56] Jianwei Zhang, Renjie Pi, Fuzhou Yao, Jicheng Quan, and Yunfei Guo. Router UDP switch support NAT traversal. In *2nd International Conference on Pervasive Computing and Applications, 2007. ICPCA 2007*, pages 568–570, July 2007.

[57] Zepeng Zhang, Xiangming Wen, and Wei Zheng. A NAT traversal mechanism for peer-to-peer networks. In *2009 International Symposium on Intelligent Ubiquitous Computing and Education*, pages 129–132, May 2009.