THE DYNAMIC PORT RESERVATION PROTOCOL

BY

ANDREW JOSEPH REITZ

B.S., Case Western Reserve University, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

Abstract

In the current Internet, Network Address Translation (NAT) gateways that provide port address translation are quite popular. These gateways allow many hosts to be multiplexed on one single IP address and still maintain full *outbound* connectivity. However, the ability to share a single IP address with many hosts doesn't come for free - the NAT denies these hosts the ability to receive unsolicited *inbound* connections.

The lack of inbound connectivity is fine for a user base that wishes to only surf the web and check e-mail. However, with the rise of peer-to-peer applications such as instant messaging, Napster and Internet-enabled games, users are demanding inbound connectivity at an ever-increasing rate.

Most NAT gateways already provide a method to restore limited inbound connectivity. It is possible to instruct the NAT gateway to forward unsolicited inbound packets on a specific port to a specific internal host. This is typically hard to configure and is only available to the administrators of the NAT gateway.

In order to make the port-forwarding solution generally usable, a new network protocol is needed. The Dynamic Port Reservation Protocol (DPRP) allows end-users to establish their own port-forwarding rules on the NAT gateway. These port forwarding rules are not static – rather, they are dynamic. End users are only able to get a *lease* on a port, which they may use for a limited period of time, before it is reclaimed by the NAT gateway. In this manner, the gateway administrator is able to keep some measure of control over the port-forwarding rules and users are able to run their peer-to-peer applications with full functionality.

Acknowledgments

I would like to thank a number of people, without whom, this project never would have reached completion. My thesis advisor, Dr. Robin Kravets, provided me with invaluable direction with every phase of this project. DPRP would be a hulking, unpublishable mass if it were not for her keen guidance. Additionally, I would like to thank Dr. Ralph Johnson, who's programming techniques helped me keep the DPRP reference implementation on track.

Several of the friends that I made at the university also played key roles in making this project happen. Keith Wessel bootstraped me on a lot of the ins-and-outs of thesis writing at UIUC. His demonstration of the LaTeX style files was invaluable in producing this document in such a timely manner. Kris Wehner also supplied lots of Java and LaTeX help, as well as general encouragement. Fredrik Vraalsen came in with many clutch pieces of advice, and saved me from tearing out all of my hair when I couldn't get an underscore to show up the references.

Finally, I would like to thank my parents, Pat and Randy, for all of their continued support, including feeding and housing me during the last several months of this project. I would also like to thank my sisters, Beth and Sara, for their support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Network Address Translation (NAT) [1], was developed as a stop-gap measure in order to extend the life of IPv4 [2] networks in the face of address exhaustion. As the Internet (the world's largest IPv4 network) grows, it faces an exhaustion of unique addresses. Part of this exhaustion is due to inefficiencies in the allocation of addresses (class-based, broadcast addresses, unusable "zero network", multicast). For example, a Class A network provides approximately 16 million addresses to only *one* entity. Furthermore, if the assigned entity doesn't use all of these addresses, there is no provision to allocate them to other entities. Consequently, these unused addresses are considered to be wasted. Some improvements to IPv4 have been made, such as allowing the "zero network" (a zero in one of the octets) to be used. Furthermore, Classless Inter-Domain Routing (CIDR) [3] has been developed in order to ease the restrictions of the class system, thus increasing address efficiency. However, the rate of growth of the Internet has surpassed all of these enhancements to IPv4.

In light of these problems, the Internet Engineering Task Force (IETF) has engineerd a successor to IPv4, which they have creatively entitled "IPv6" [4]. This new protocol provides 128-bit address, which should eliminate IP address exhaustion concerns for several decades. Unfortunately, the rate of adoption for IPv6 has proven to be quite slow [5]. Furthermore, since IPv6 is not currently available to the general

public, facilities that extend the usefulenesss of IPv4 are still of great interest.

Consequently, a methodology for multiplexing many host computers onto one (or more) IP addresses was developed. This technique, called NAT, has allowed the useful life of IPv4 to be greatly extended, by easing the requirement that each host on the Internet requires a unique IP address. Essentially, this technique employs the use of a dual-homed device that modifies packets as they travel through it. This modification usually entails substituting the source IP address for that of a different source IP address. A mapping is kept, from <original source IP> to <new source IP>, so that when a return packet comes back with a destination of <new source IP>, the destination field can be changed to <original source IP>, and sent to the original source.

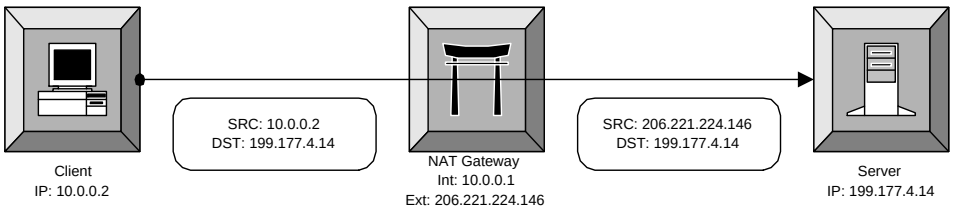

Figure 1.1: A simple NAT gateway. Notice that the gateway only modifies the *source* IP address of the packet.

Variations of this technology allow multiple hosts to share one IP address. This variation is typically called "Network Address Port Translation" (NAPT) [6], and takes advantage of the fact that the majority of Internet workstations only launch outgoing connections (to fetch a web page, for instance). As such, if you consider a host to be the <IP address, port> pair, then because there are approximately 65,000 possible ports, it is possible to multiplex many hosts on one IP address. This has become quite popular in the marketplace, from large organizations all the way down to small home installations. It is quite common for large organizations to use private, non-routable IP addresses on their intranet, and to provide a NAT gateway for Internet access. This increases address efficiency and flexibility, and also promotes

security. On the Small Home/Home Office (SOHO) end, the increase in computing power demanded by Internet applications has not kept up with the increases provided by Moore's law [7]. Thus, as people purchase newer computers, their old models are still viable. The drive to allow all of these computers to share one Internet connection has made NAT quite prevalent in the home.

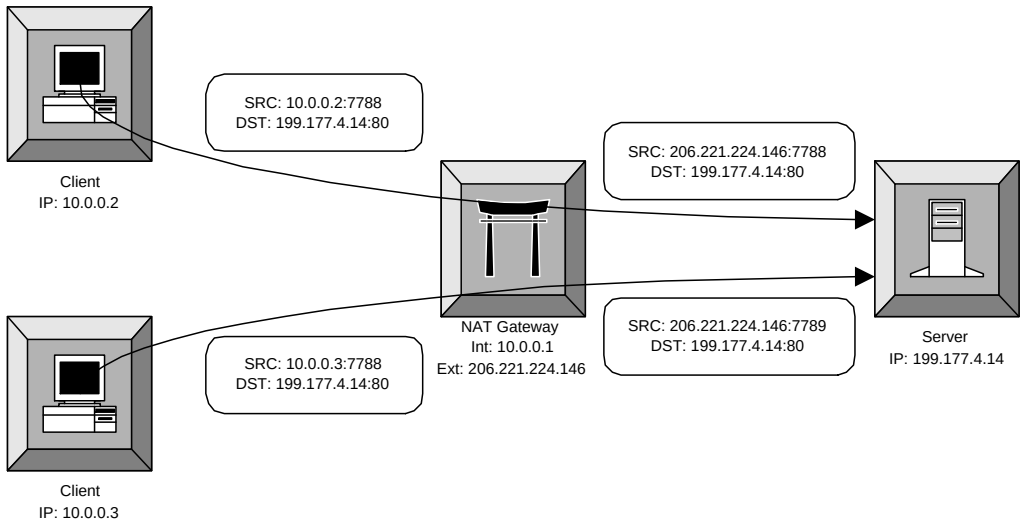

Figure 1.2: A complex NAT gateway, performing port-address translation. Notice that the gateway modifies the *source* IP address and port of each the packet.

As demonstrated in figure 1.2, NAPT gateways operate by maintaining a translations table, consisting of an conversion from <internal IP, port> to <external IP, port>. When the gateway receives a packet from an internal machine, it matches the IP address and port to all of the entries in the table. If a table entry is found, then the corresponding <external IP, port> will be substituted in place of the <internal IP, port>, and the packet will be passed to the external interface. If a translation entry isn't found, then a new one will be created by selecting a free <external IP, port> pair. When an incoming packet is received (on the external interface), the translation table is again searched, this time on the <external IP, port> side. If an entry is found, then the destination IP and port are replaced with the <internal IP,

port> as specified in the table. However, if no entry is found, then the incoming packet is discarded – because the gateway has no way to divine where this packet needs to go[1].

NAPT has been implemented everywhere, and it functions quite well for applications that only establish out-bound connections. However, for applications that require the use of in-bound connections (i.e. some host on the outside of the NAPT gateway attempts to establish a connection to an inside host), the NAPT scheme forms an insurmountable barrier. There can be good reasons for applications wishing to establish an in-bound connection. The most popular implementation of the FTP protocol [8], for instance, defines transfers such that the server opens a connection to the client machine in order to send it a file. However, the need for in-bound connectivity has really been highlighted by the upward trend of "peer to peer" (P2P) applications, such as Napster [9]. These applications typically work by providing a central index of shared resources. Clients connect to the central index, perform a search, and find links to data elements on other client machines. In order to fetch one of these data elements, the searching client attempts to establish a direct connection to the client that holds the data. Hence, in this architecture, there is no clear distinction between "client" and "server".

Because the NAPT gateway appears to the world as a single entity, there is no way for external hosts to address internal hosts directly. Furthermore, because the NAPT gateway is transparent, there isn't any way for the internal hosts, once listening to a port, to notify any outside parties about it's port. The NAPT gateway isn't required to guarantee that ports used by the internal hosts will be the same after translation. Thus, supporting peer-to-peer applications where both peers are behind NAPT gateways becomes impossible. It is this problem that this work will attempt to solve.

[1]Actually, in some implementations the "lost packet" is consumed by the gateway's IP stack.

Chapter 2

Motivation and Background

Although NAT has a problem of destroying inbound connectivity, we do not want to replace it as a solution. The addition of NAT[1] to IPv4 has given network administrators an amazing amount of flexibility in the way they plan, provision and implement their networks. Freed from the restrictions of requiring registered IP addresses for each of their hosts, network administrators can focus on hierarchical organization and pure network design. Furthermore, on the other side of the spectrum, user's are free to configure all of their computers so that they can share a single Internet connection. Not only is this incredibly convenient, but it also saves time and money.

Additionally, NAT carries with it several nice security properties. Because NAT obscures multiple physical computers behind one IP address, it is far more difficult for the outside world to determine the topology and contents of the internal network. Additionally, because NAT gateways don't allow unsolicited connections to ports, security is increased in the face of vulnerabilities of network daemons. Therefore, sites concerned with security can take a relaxed attitude on their internal network and only worry about those machines to which in-bound access through the NAT has been configured.

[1] In the context of this document, all references to NAT are really references to NAPT. The two terms are used interchangeably.

Because NAT has so many wonderful properties, nobody is seeking to eliminate it until IPv6 arrives. However, many trends in networked applications are colluding to make inbound connectivity a higher requirement than ever. For example, many peer-to-peer applications such as file sharing services, network games and even instant messaging are *insanely* popular, with hundreds of millions of users world-wide. It is important to address the lack of inbound connectivity that NAT affords. Summarized below are several other efforts made at resolving this problem.

2.1 Related Work

The problem of non-existent in-bound connectivity has existed since NAT was invented. However, with the rise in peer-to-peer applications, this problem has come under more scrutiny. As such, several other projects have been started in order to deal with NAT in-bound connectivity problems. What follows is an analysis of three of the most popular projects.

2.1.1 Hack for Peer-to-Peer/UDP Applications

Dan Kegel has described a potential solution, which he calls "NAT and Peer-to-peer networking" [10]. In this scenario, we assume that the hosts that wish to establish a connection already have open connections to a single server (just as a master P2P server, or a gaming server). Through this connection, each host can relay information to any other host. So, for two hosts that are both stuck behind NAT gateways, they can exchange connection-setup information with each other through the common server link. In this scheme, this information will consist of each host's NAT gateway IP address, and a UDP port. Once this information has been exchanged, the hosts will each attempt to build outgoing connections to each other, using the local port that they sent, and connecting to the port number that they received. This will cause

each NAT gateway to build a translation <external IP_1, $port_1$> to <external IP_2, $port_2$>. Hence, because this translation will be installed in each gateway, the two hosts will be able to communicate directly.

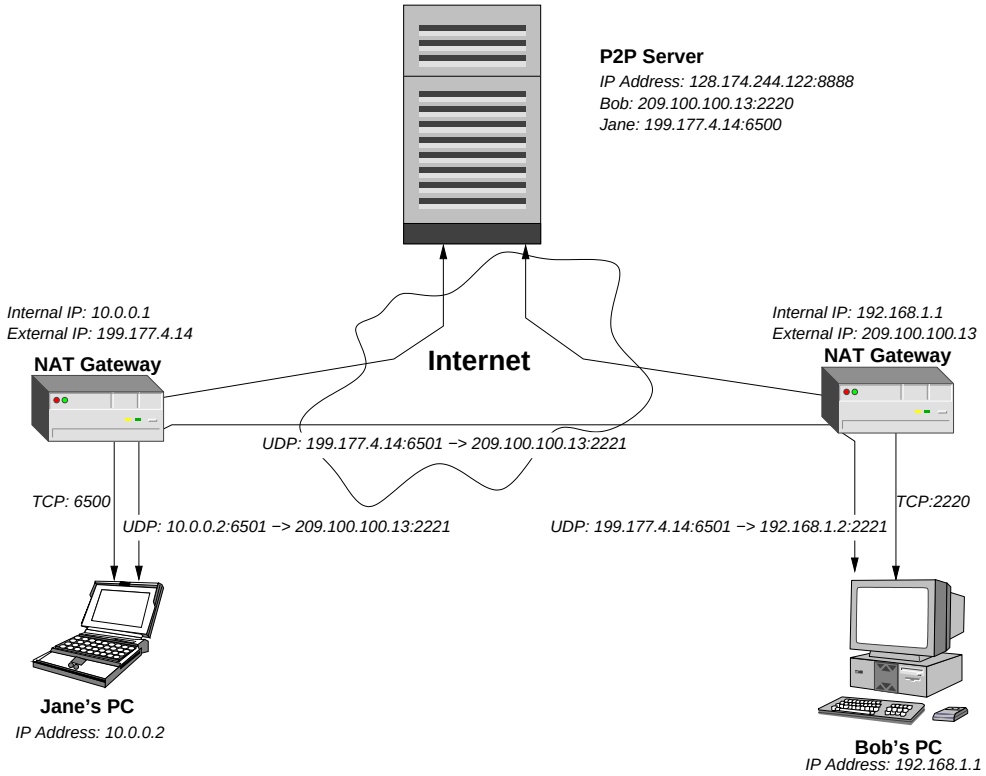

Figure 2.1: Example of the P2P/UDP hack.

Figure 2.1.1 depicts an example of Dan Kegel's methodology in action. In this scenario, Jane and Bob wish to play a network game. Both Jane and Bob start the game on their respective computers, and the game automatically logs them in to the common *gaming server*. Using this server, Jane and Bob are able to start a network game. In doing so, the gaming server gives what it thinks is Bob's IP address and port to Jane's PC, and vice-versa for Jane's IP information to Bob's PC. With this information established, both Jane's and Bob's PCs start attempting to connect to one another with UDP packets. These packets cause each person's respective NAT gateways to "open a hole", that permits the other's packets to come in through the network. In this manner, the UDP connection is established, and the gaming session

may commence.

The positive aspects of this idea are that it requires *a minimal* amount of changes to existing infrastructure. No changes need to be made to existing IP stacks, routers, or to *compatible* NAT gateways. Only software applications require modification, in order to attempt to build the UDP connection simultaneously from both peers. Furthermore, since the implementation rests solely in the application, it is easy to deploy. For example, two different games already implement this technique: "Civilization: Call To Power" and "Heavy Gear 2".

There are several aspects of this idea that render it insufficient. The first is that it only works for applications that use some sort of "common server", in order to exchange the remote IP and port information. Without this "meta channel", this idea simply won't work. Furthermore, this scheme requires a *compatible* NAT gateway. By compatible, all of the NAT gateways must ensure that they will only translate IP addresses, but not UDP ports. However, in the case of two internal hosts attempting to use the same UDP port (to the same remote host), one UDP port will have to be translated, and the system breaks down. The software will have to work around this, by choosing random ports. Furthermore, some NAT implementations simply don't choose the same port for the internal and external sides of the translation. These NATs are completely incompatible with this protocol, and must be modified in order to not arbitrarily exchange port numbers.

Finally, this idea only works with the UDP protocol [11]. Because UDP is a *connectionless* protocol, the NAT has to work optimistically, and assume that *any* UDP packet sent is the start of a connection. Thus, the NAT can be *spoofed* into thinking that a new connection is being created with the above method. However, TCP is connection-oriented [12], and as such, it has a definite connection setup procedure. This procedure consists of a three-way handshake between the two hosts. As a part of this handshake, each host encodes a random "sequence number" in each of it's

packets. When replying, the hosts send their partner's sequence number back, incremented. Thus, the NAT gateway can inspect these TCP connection setup packets, and only allow the packets in the proper sequence through the gateway. Since it is *highly* unlikely that the two hosts will randomly choose compatible sequence numbers, the only way to make this work would be to use a raw socket, and for the application to implement it's own TCP protocol. Obviously, this is highly undesirable, and for all intents and purposes, this method is relegated to use with UDP.

2.1.2 Realm-Specific IP

Realm-Specific IP (RSIP) [13] is a new IP technology that is currently being developed by the IETF. Essentially, RSIP allows hosts with private IP addresses s to get a "lease" on a valid public IP from the NAT gateway. The host then builds packets using the valid public address, and ships them off to the gateway via a tunnel. The gateway then de-encapsulates these packets, and sends them out its public interface. Return packets are intercepted by the NAT gateway, and are again tunneled back to the private host.

Figure 2.1.2 provides an example of one way that the RSIP protocol can be employed. In this example, Bob wants to connect to Jane's PC for some friendly video-conferencing. In order for this to happen, Jane must first acquire an external IP address from the RSIP-enabled NAT gateway. Next, Jane must establish an IP tunnel between her PC and the RSIP-enabled NAT gateway. At this point, Jane is ready to advertise her IP address to Bob. Eager to start his video-conference, Bob directs his PC to connect to the IP address that Jane gave him – 199.177.4.15. But the NAT gateway is actually listening on this address, so it intercepts all of Bob's packets. Determining that these packets belong to a valid RSIP lease, the NAT gateway will then place them into the tunnel, destined for Jane's PC. When Jane formulates her reply, her packets to Bob will be first encapsulated in the tunnel, and then forwarded

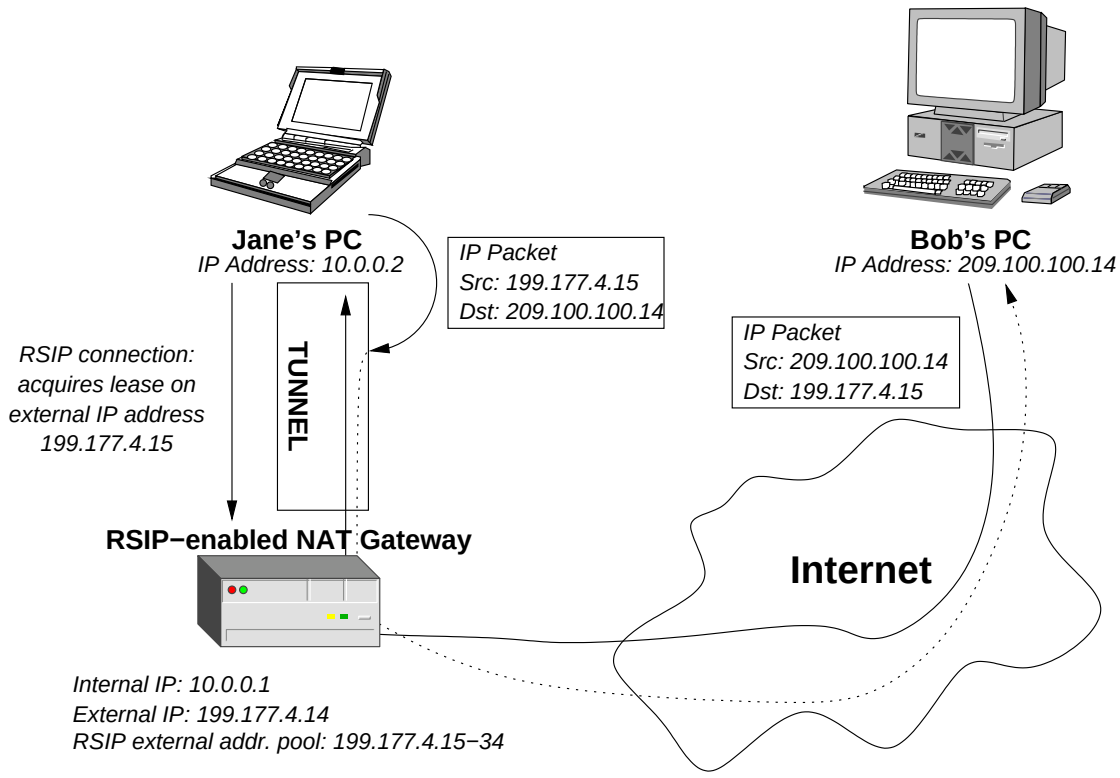

Figure 2.2: Example of Realm-Specific IP.

on to Bob by the NAT gateway.

All told, RSIP is a complete solution to the problem of establishing full bidirectional connectivity between differing IP domains. As such, RSIP is the only solution that allows things like IPsec [14] to work with full functionality. IPsec is difficult to support with NAT gateways because as a protocol, it is quite sensitive to modifications to packets once they have been created. NAT, by its very definition, modifies packets as they traverse through the gateway. The primary drawback of RSIP is that it requires extensive modifications to existing infrastructure. Not only must the NAT gateways be modified, but the IP stacks on all hosts that wish to obtain RSIP leases particular must be modified as well. For any scheme that requires modifications to the IP stack on host machines, a slow adoption rate must be expected. The majority of the Internet-connected hosts run proprietary operating systems; and vendors are notoriously slow at upgrading their IP stacks to support new technology. Thus, the

extent of these modifications means that the rate of adoption for RSIP will be quite slow.

2.1.3 AVES

The *Address Virtualization Enabling Service* (AVES) [15] is a scheme for providing "complete" in-bound connectivity to private hosts. The authors have striven to create a system that supports any arbitrary service, hence their usage of the word "complete". AVES has been shown to support applications such as FTP, telnet, SSH, and even the X Window system. The only applications that encounter difficulties with AVES are those that are not "NAT-friendly" [16], such as IPsec. Essentially, this system works by assigning a DNS [17] name in a special domain to each private host that wants in-bound connectivity. When the DNS server for the special domain receives a lookup request for an AVES name, the DNS server contacts the *waypoint* (a sort of proxy). This initial contact marks the beginning of an AVES *session*. The waypoint chooses a valid public IP address (from a pool), and returns this address to the DNS server. The DNS server then returns this IP address to the requester. So, now a dynamic binding, or session, has been established for the private host – it has a valid public IP that it can use to service incoming requests. This session will remain valid until the waypoint times it out.

The functionality provided by the waypoint is more than just the binding of DNS names to public IP addresses. Because AVES is transparent to existing IP hosts, the waypoint must be used in order to maintain this illusion. The initiator of the session is the host that performed the initial DNS lookup. Since it was given a public IP address that the waypoint controls, it will send all of it's packets to the waypoint directly. Thus, the waypoint must encapsulate these packets, and route them to the NAT gateway of the corresponding private host. This gateway will de-encapsulate the packet from the NAT gateway, and forward the initiator's packet to the private

host. The private host can then respond to the initiator directly.

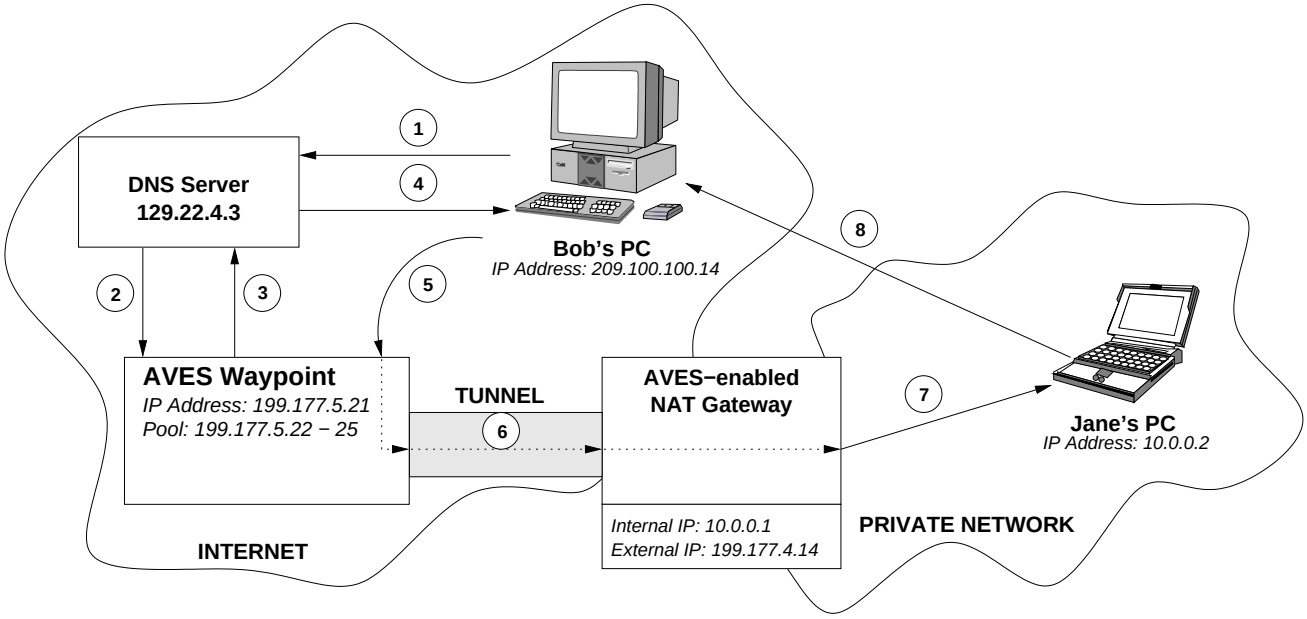

Figure 2.3: Example of AVES.

Figure 2.1.3 and table 2.1 conspire to provide an example of AVES in action. In this example, Bob wishes to establish a TCP connection with Jane. To do this, Jane tells Bob her computer's AVES-based address "jane.aves.org". Subsequently, Bob directs his PC to this address, in order to establish the connection. Figure 2.1.3 gives an overview of the messages that are either sent or mangled in order to establish Bob's TCP connection. Table 2.1 provides the details concerning the contents of each message.

The advantages of this scheme are that it supports a wide range of applications, not just peer-to-peer. Furthermore, it can even be used to setup static servers on private hosts (such as SSH or WWW). Also, this idea doesn't require any modifications to Internet hosts, which eases the deployment burden. In fact, AVES only mandates changes to DNS servers and NAT gateways, for domains that wish to provide public services from behind the NAT. So, those changes, coupled with the deployment of

Message:	Description:
1	Bob's PC makes a DNS query: "Who is jane.aves.org?"
2	The DNS server starts session with Waypoint, asking it to assign an IP address to 'jane.aves.org'.
3	The assignment is successful, and the Waypoint sends the reserved IP address, '199.177.5.23', to the DNS server.
4	The DNS server replies authoritatively that 'jane.aves.org' resolves to '199.177.5.23', but that this information should not be cached.
5	Bob's PC starts its TCP session with Jane, by sending a packet to '199.177.5.23', which is controlled by the Waypoint server.
6	The Waypoint changes the destination address of Bob's TCP packet to '10.0.0.2', the private IP address of Jane's PC. It then encapsulates this packet into a new IP packet, with a source address of '199.177.5.21' (the Waypoint) and a destination address of '199.177.4.14' (the NAT gateway).
1	The AVES-enabled NAT gateway de-encapsulates the Waypoint's packet, and places Bob's TCP packet on the wire, bound for Jane's PC.
8	Jane's PC formulates a response, which it sends directly to Bob's PC (via the NAT gateway).

Table 2.1: Description of messages in Figure 2.1.3

waypoint servers, means that AVES can be rolled out into the Internet at large in a progressive, as-need fashion.

There are many disadvantages to the AVES system, however. The primary disadvantage is that AVES requires public IP addresses that are separate from those assigned to the NAT gateway. In the worst case, AVES can require a one-to-one mapping between private hosts and public IP addresses. This can occur when every host behind the NAT gateway has an entry in the AVES-aware DNS, and is handing an in-bound connection. At this point, every purpose of using a NAT gateway has been utterly defeated. The NAT gateway is no longer concealing any hosts (they all have DNS entries in the AVES-aware DNS server), nor is it conserving any IP addresses. Conceding that this "worst-case" scenario may be rare, AVES does move against the grain of NAT. The entire reason for developing NAT was use IP addresses more efficiently. AVES weakens the efficiency gained by NAT, and thus it is somewhat opposed to the main driving forces of NAT.

The fact that AVES requires modified DNS servers poses several problems. The first of which is that the DNS requirement implies a high administrative overhead. Instead of just registering public hosts, now the DNS administrators will also have to administer private hosts (which were once immune from consideration in the DNS tables). Furthermore, the DNS administrators must be aware of all of the NAT gateways in an organization, because these gateways must be added to the DNS configuration. Furthermore, the AVES-specific modifications currently aren't a part of the main *Bind* [18] distribution. As such, whenever a new release of bind is made (in order to fix a security hole or add a new feature), the site administrator will have a more complex time upgrading, because the AVES code must be rolled into the new release. Finally, even though AVES-specific DNS names are set to be "un-cacheable", non-compliant applications (such as web browsers) will cache these names. Thus, even though AVES purports to be transparent, there could be problems with some applications, due to their non-compliance with standard DNS protocols. Finally, tying DNS records to private hosts and NAT gateways is problematic when dynamic addressing is used. Dynamic DNS extensions can be used to ease the pain for private hosts that are assigned dynamic addresses, but this increases complexity, and also has security implications. And AVES contains extensions for NAT gateways that receive dynamic addresses (such as a SOHO router for DSL or Cable Modem). However, this all implies added complexity that is required in order to make the system work.

The last major disadvantage of AVES occurs when the private AVES hosts are sitting behind a router that performs ingress filtering [19]. Ingress filtering is a technique where packets that have a *forged* source address are dropped by appropriately-configured routers. This sort of filtering has been on the rise, because it is effective at quelling most *Denial-of-Service* [19] attacks at the source. As defined, AVES will send return packets directly from the NAT gateway. However, if this were a real IP connection, the packets should appear to be coming from the waypoint. Thus, ingress

filtering will knock these forged packets out of the network, and the reverse path from the private host to the initiator will be quelled. In order to work around this, the authors (correctly) suggest that the return traffic will have to be tunneled through the waypoint server. This is a costly scenario, in terms of performance and waypoint scalability. Furthermore, this scenario has disastrous implications in the peer-to-peer space, because not only does it destroy the scalability of the peer-to-peer environment, but it also raises legal concerns because the peer's traffic is passing through a third party's waypoint servers. Thus, this is really a show-stopper for AVES. It all depends on how much ingress filtering is growing within the Internet as a whole.

2.2 A New Solution Is Needed

In this existing body of work, no one method achieves the ideal balance between connectivity and deployability. There is still room for improvement. In this proposal, a new method for addressing the inbound connectivity problems of NAT will be presented. Currently, most NAT gateways provide a facility where an administrator can allocate specific ports, and make those ports "passthru" to specific private hosts. For example, the administrator could say that all traffic bound for TCP port 80 on the gateway should be translated and forwarded to a specific internal host. That host will service any requests and send the resultant data back through the NAT, for the usual packet mangling. This technique works well, in that it allows NAT-unfriendly applications to work through a NAT. The only downside is that it requires explicit administrator intervention and management for each host in the private network.

The approach described herein applies DHCP-like [20] leasing techniques to this scenario, via the Dynamic Port Reservation Protocol (DPRP). When an application decides that it needs in-bound connectivity from the outside world, it can send a "protocol port request" message to the NAT gateway. The gateway will examine

it's internal translation tables and satisfy the request with an appropriate port. For example, if a private host wanted to run a web server, it would contact the NAT gateway and request TCP port 80. If unallocated, the NAT gateway would set up a translation for TCP port 80 and pass all traffic through to the private host that requested it. However, this translation won't be static – it will be leased. So, the requestor will have this translation for some period of time (say 24 hours), at which point he/she will have to renew the lease. This is the "DHCP leasing scheme". DPRP carries other DHCP-like features with it, such as reliability in the face of NAT gateway and client failures (on the NAT gateway, the leases will be logged to some stable storage).

There are two different methods of utilizing this protocol in order to obtain an outside port. The first method entails creating a simple client application that an end-user can use in order to request a port. Since a vanilla web server doesn't understand anything about DPRP, the user could run the client, get a port and then configure that port into the web server. The application would continue to run, updating the lease as necessary. This solution would provide support for *legacy* applications.

Additionally, because there is an actual protocol in place, "smart" applications could be developed. These applications would handle outside port allocation without explicit action on the part of the end-user. The vast majority of NAT users (Cable Modem/DSL users sharing a connection, etc.) are only going to run certain types of applications that require in-bound access (primarily peer-to-peer applications like Napster and games). These niche applications could write code that detects if they are running behind a NAT (this infrastructure is already in place for most applications), and if so, attempt to register a port with the NAT gateway.

It's important to note that the *functionality* of the NAT gateway is not changing at all. Instead, another feature is being added – that of "dynamic/static port assignment" for private hosts. Everything else about the NAT (the way it mangles packets, manages its translation table, etc) will remain the same.

Chapter 3

DPRP Design and Implementation

3.1 The DPRP Protocol

The design of the Dynamic Port Reservation Protocol (DPRP) was driven by the twin factors of simplicity and robustness. The protocol needed to be simple (so that the corresponding implementations of it would be easy to create) and have a low operating impact. By it's very nature, DPRP is a protocol that is going to have to be added "after the fact" to a variety of devices. On the client side, DPRP is positioned to end up in a range of applications from running on fully-fledged PCs to instant messaging applications running on cellular phones. On the server side, DPRP will see implementations in everything from classic UNIX-based routers, to proprietary hardware routers, to small SOHO routers. Therefore, DPRP as a protocol, must be structured so that it is easily implemented on a wide range of networked computing devices.

In terms of robustness, DPRP must be able to operate properly in network environments that experience packet loss and latency, and also in the face of client and server failures. Further, it must be robust enough to handle resource exhaustion, non-conformant client and server implementations and malicious users. It should even be feasible to separate the DPRP server and NAT gateway functionality be-

tween two separate hosts, and still maintain the necessary robustness properties. In short, DPRP should be capable of running in a variety of environments and under a variety of conditions. DPRP should run wherever and whenever it is needed to solve NAT connectivity problems.

Basing DPRP on the UDP/IP protocol is the correct choice in order to keep the protocol simple. The UDP protocol is the easiest of the TCP/IP suite to implement, and as such will be available on the maximum number of devices. All of the packet exchanges in DPRP will be small in number – the longest conversation between a client and server is 4 packets. Consequently, it isn't complex to make these short exchanges work in the face of UDP's unreliability The other alternative, TCP, imposes too high an overhead for such short packet exchanges.

In order to ensure that DPRP carries a low code overhead, much attention was paid to how the data fields were placed into the DPRP packet. The two design choices were: to use a compact, packed byte representation or to use an ASCII text-based format, similar to the format employed by the HTTP protocol [21]. Text-based protocols, however, require a text parsing engine, which carries significant code space and complexity costs with it. Also, text-based protocols provide a degree of flexibility that would have gone to waste in the DPRP space. So, a packed format of fixed-length was crafted to serve as the cornerstone of the DPRP protocol. Fixed-length, byte packed messages are simple to decode because they do not require dynamically allocated memory or complex parsers. Furthermore, the elements are all packed in network byte order, and are thus immune to endianness translations. That makes this design as compatible as possible with the widest range of network-capable devices.

With the above factors in mind, the DPRP client was largely designed through a state diagram. Figure 3.1 shows the state diagram that was used to guide the reference implementation. Since a DHCP client has a similar form of functionality to the DPRP client, the same structure was adopted. The REQUEST, OFFER,

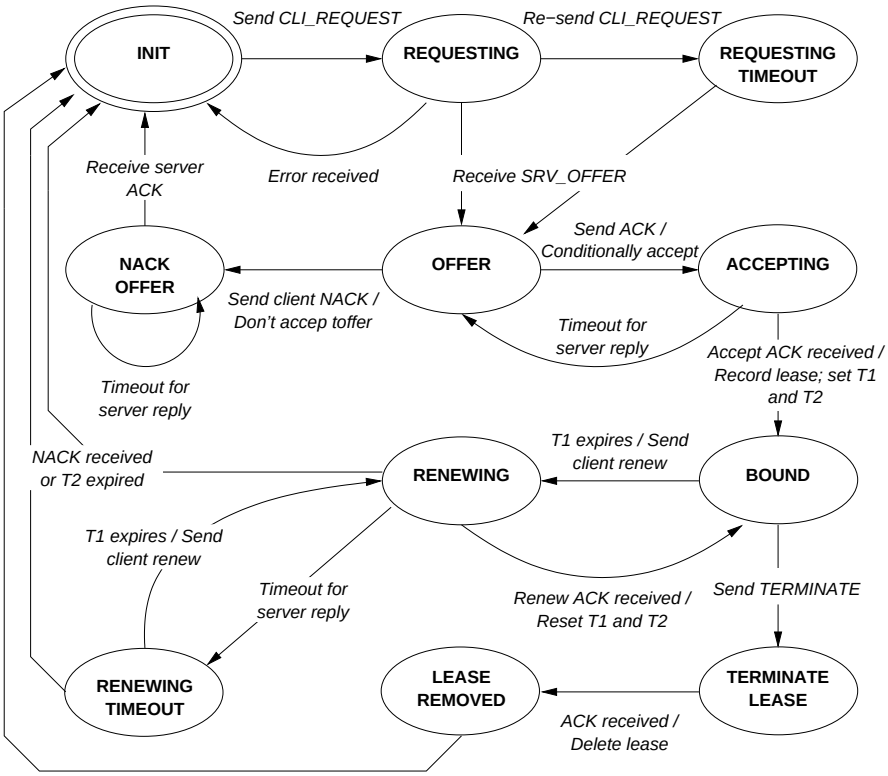

Figure 3.1: DPRP client state diagram.

ACCEPT, RENEW, and BOUND states were all inspired by DHCP. The nature of the DPRP lease request phase permits the client a measure of flexibility, while still guaranteeing that the process will complete in the face of network errors. With the ability to NACK an offer, clients can negotiate with servers in order to determine the set of parameters that appeases both sides. Also, response timeouts and packet retransmissions are built into every stage of the protocol, for correctness in the face of network errors.

The most important element of the DPRP server's design is the guarantees it makes before a lease is committed. It is imperative that the server only commits a lease when it is guaranteed that the client is in the correct state. As figure 3.2 demonstrates, the server will only commit a lease when it has received a proper "OFFER ACK" message from the client. This commit operation isn't really permanent, however, until the server is assured that the client has received its "COMMIT ACK".

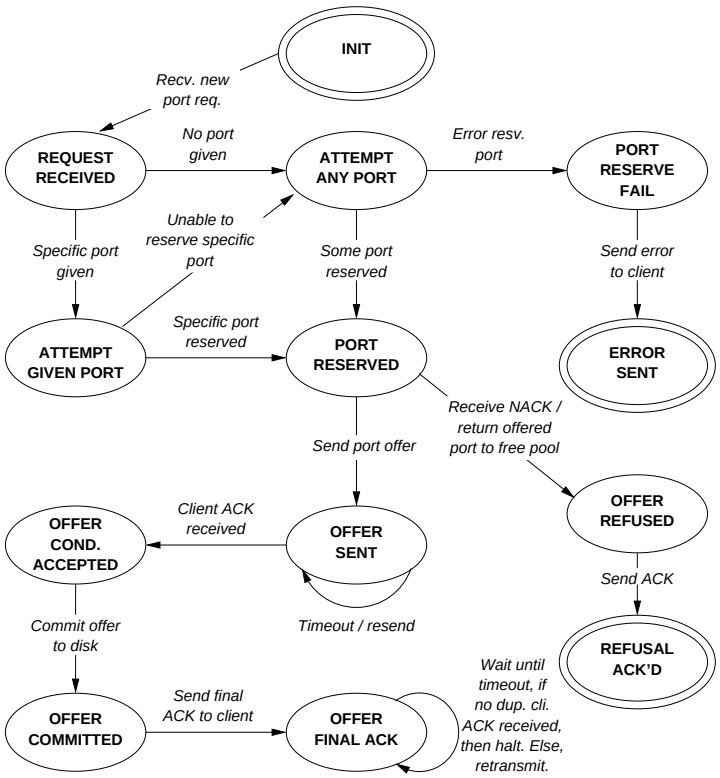

Figure 3.2: DPRP main server state diagram.

If this final ACK cannot be processed, then the lease is immediately destroyed and the port returned to the gateway. Additionally, the server's ACK handling is explicitly designed to function correctly in the face of lost packets. Whenever the server sends a final ACK packet, it waits for a specific timeout period. If no further packets are received from the client, then the server assumes that the transaction completed successfully. If, however, the server receives a duplicate of the last packet in the session, it assumes that the ACK packet was lost, resends it, and resets it's timeout counter. In this way, the DPRP server has a guarantee that it can correctly identify and handle network problems.

The renew transaction is guided by the T1 and T2 timers. DHCP was the inspiration for this process. The T1 timer is initially set to be half of the lease duration. The T2 timer is initially set to be $\frac{7}{8}$ of the lease duration. When T1 expires, the DPRP client attempts to renew the lease. As figure 3.3 shows, it is possible that the

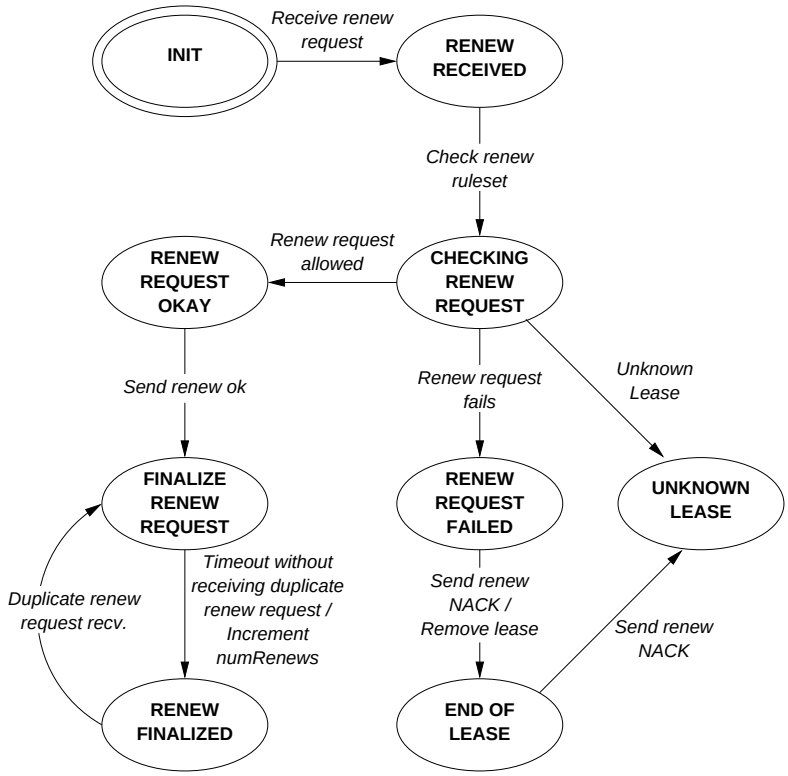

Figure 3.3: DPRP server renew state diagram.

server will explicitly reject the client's lease renew request. If this happens, the client is mandated to immediately terminate the lease and inform the user. However, if the renewal attempt fails without getting an explicit "lease revoked" message from the server, then the client will wait for $renewTimeout = \frac{1}{2} * (T2 - T1)$ seconds. Once the $renewTimeout$ timer expires, the client will attempt to renew the lease again. If the renew attempt is still unsuccessful, $renewTimeout$ will be divided by half, and the client will wait again. The client will continue this process until either T2 expires, the lease duration is exceeded, or the lease is successfully renewed. If the client is unable to renew the lease, it will be expunged, and the user will be notified.

Finally, the implicit design of both the client and server is that unreliable error messages should be sent whenever possible. For example, if the client receives the wrong message at the wrong time, an error should be generated. Or, if a server is unable to decode a received message properly, an error should be sent to the source of

the message. These error messages not only aid the client and server in recovery from failed transactions, but also provide useful feedback to both users and administrators.

3.2 Sample DPRP Client/Server Implementation

Although a design for DPRP was in place, no protocol truly exists until it has been implemented in code. Code is the most effective means for determining if a protocol specification is complete and correct. Further, in order to determine the ultimate viability of DPRP, it needed to be enacted on a live NAT gateway. For reference implementation purposes, it is important to get something working rapidly and portably. Since this is only a reference implementation, sacrifices of speed and compilability are forgivable. To that end, Sun's Java [22] language was chosen for the implementation of both the DPRP client and server. Further aid was afforded by Jason Goldschmidt and Nick Stone's JDHCP [23], a sample DHCP implementation also written in Java. This project inspired several key pieces of the DPRP implementation, including how to create UDP network packets from Java objects, a "smart socket" that understood DPRP message objects.

The sample DPRP implementation started simply, and grew with every feature added, into several fully-fledged applications. The code was refactored and reorganized several times, as new functionality was added. Furthermore, after each change, the code was extensively tested. A feature wasn't decreed to be "complete" until it was tested and found to be working properly. This iterative process kept the project manageable at all times. Additionally, extensive use was made of Sun's `javadoc` facilities, to document all of the DPRP classes and methods. Not only did explaining the code help to find problems, but it also made the code more accessible and understandable to others.

DPRP Data Field	Java Datatype
messageType	byte
protocol	byte
flags	short
xid	int
duration	int
srcAddr	InetAddress
srcPort	int
dstAddr	InetAddress
dstPort	int
extAddr	InetAddress
extPort	int

Table 3.1: `DPRPMessage` fields and data types

3.2.1 The `DPRPMessage` class

The first major class to be written was the `DPRPMessage` class, which represents an individual DPRP datagram. JDHCP demonstrated that the packets that constituted DPRP are ultimately the most essential class in any implementation. The major components of this class are the instance variables, which detail each field in the protocol. Table 3.1 highlights these fields (as named in the `DPRPMessage` class) and specifies their corresponding Java datatypes.

The second major component of `DPRPMessage` is that it contains all of the code that serializes and deserializes a `DPRPMessage` object instance. Serialization is the process of converting all of the object's data elements into a single byte stream, suitable for sending on the network. Deserialization is the accompanying process of converting a byte stream back into an object instance. These methods became part of the basic `DPRPMessage` based upon influence from JDHCP. This location suited the DPRP code base well, although the choice of datatype for IP addresses posed several unforeseen programming complexities.

Java's `java.net.InetAddress` class is the language's way of representing an Internet address. `InetAddress` is a powerful IP address object that supports name resolution, object equality and is compatible with Java's socket API. Unfortunately,

`InetAddress` also has several important limitations that unnecessarily increased the complexity of the `DPRPMessage` serialization code. The primary limitation of the `InetAddress` class is that, although it is possible to convert the address *into* a signed byte array (suitable for serialization), there is no way to instantiate an `InetAddress` *from* a signed byte array (as needed for deserialization). The four unsigned bytes are first converted into a dotted-quad string representation of the IP address, at which point the proper `InetAddress` object could be instantiated. This presented another problem: Java has only *signed* datatypes. Thus, the individual bytes could not be converted into strings directly – they each had to be converted to the unsigned representations, and then stored into an integer. From this integer, each value could then be converted to the proper string representation.

The only other major problem with the `DPRPMessage` class came with the attempt to overload the `equals()` method from class `Object`. This method must be overloaded so that `DPRPMessage` instances can be stored properly in a variety of hashing data structures. In the first attempt, a method with the following prototype was created:

```
public boolean equals (DPRPMessage cmp)
```

But even with this method, the hashing data structures were still functioning improperly. After debugging, it was determined that the `DPRPMessage.equals()` method was never being called. Further exploration discovered that hashing data structures were calling a method of the following prototype:

```
public boolean equals (Object o)
```

This is because the hashing data structures can accept any object, and then have to cast-down to the most basic object type. The above method was added to `DPRPMessage`, so that upon invocation, it checks to make sure that the parameter is really of type `DPRPMessage`, in which case, it simply calls the first `equals()` method that was written. With this bug out of the way, `DPRPMessage` was largely complete, and it was time to start putting `DPRPMessage` objects on the network.

3.2.2 The `DPRPSocket` class

Some sort of structured facility based upon UDP sockets was needed so that the DPRP client and server could exchange messages. In JDHCP, a layer of code was placed above the raw datagram sockets so that `DHCPMessage` objects could easily be sent and received. This layer of code consisted of a class named `DHCPSocket`, and contained `send()` and `receive()` methods. The `send()` method serializes the given `DHCPMessage` object and places the resulting byte stream into the underlying datagram socket. Similarly, the `receive()` method extracts a byte stream from the datagram socket, which is de-serialized into a `DHCPMessage` object instance, and then returned. This architecture was a natural fit for DPRP, so the `DHCPSocket` class was adapted to become the `DPRPSocket` class.

This conversion was fairly straight-forward, with the exception of some problems encountered in the `receive()` method. The DPRP code mandated changes to the timeout structure, so that `receive()` would block both on the underlying socket and also in the face of an error. This gave an added calling consistency (since `receive()` naturally consumed socket errors) to the overlying code. A `sendError()` method, was also added so that senders could be notified of error conditions.

With a functioning `DPRPSocket` in place, the path was cleared to create the first `DPRPClient` and `SimpleDPRPServer` classes. In order to start passing real `DPRPMessages` on the network, the initial DPRP client simply sent a DPRP `CLI_REQUEST` request message, and the corresponding server simply sent the same packet back. Refactoring further, the client grew until it implemented the entire "client request" portion of the client state diagram, as depicted in figure 3.1. But as this implementation progressed, it become evident that some sort of object was going to be necessary in order to track all of the elements of a DPRP lease. Thus, the `DPRPLease` class was born.

3.2.3 The `DPRPLease` class

The `DPRPLease` class is where both the DPRP client and server store all of the variables that make up a "lease". In the DPRP sense, a lease object needs to consist of several private variables comprising the external IP address and port on the gateway, as well as duration, protocol, flags and the starting time for this lease. Furthermore, from analysis of DHCP, it became apparent that these lease objects needed some sort of unique identifier. Drawing upon the field of database systems, the "unique identifier" called for is what is known as a *primary key* [24]. By giving each lease a primary key, certain properties would become easy to enforce system-wide. For example, the DPRP server would be able to verify that it didn't accidentally issue two leases for the same resource. The correct primary key for `DPRPLease` objects is the conjunction of the external IP address, the external port, and the protocol, separated by colons. This text string is guaranteed to be unique in the space of valid DPRP port reservations.

The final dimension of the `DPRPLease` class is the ability to serialize each object into a text string. Part of the DHCP server specification is a mandate that the server store its offered leases on some sort of stable storage. Then, in the event of server failure, a new server instance will be able to track the same state that the previous instance tracked. Since a DPRP server provides functionality that is similar in spirit to that of a DHCP server, it should also store its leases on stable storage. Some facility was needed for the server to write all of its leases out to disk. To this end, a method named `toString()` was created. This method prints out a lease in a computer-parsable format, suitable for an on-disk database of leases. The format is as follows:

```
extIP:extPort:protocol|cliIP:cliPort|srvIP:srvPort|startTime| \
        duration|flags|numRenews
```

In order to restore `DPRPLease` objects from disk, an analog for the `toString()` method

is needed. To that end, the `fromString()` method was created. This method instantiates a new `DPRPLease` object, based upon the data values that are found in the lease string. Additional routines sat above these methods, writing all of the leases out to disk, and also reading them all back in. With all of this infrastructure in place, it was time decide with how the DPRP server would actually *manage* these lease objects.

3.2.4 The `ServerPortManager` class

As a service, a DPRP server must be designed so that it can handle many concurrent client requests for external ports. Each port request must be able to operate without infringing upon the activities of any other request. This requirement brings a strong influence upon the design of the server's internal data structures. In particular, special care must be paid to the server component that tracks outstanding leases. A centralized component is needed to serve as an arbiter between these requests. Further, because each request must ask a series of questions, it is natural to think of this arbiter as an object that can be queried. Thus, the `ServerPortManager` class was created. In particular, this class was designed to be a singleton that runs in the server's address space, and is able to answer questions of the following form:

1. Is this client allowed to request a lease?
2. Is the client requesting a specific port?
3. If so, is that port free to offer?
4. Is it within the client's allowed port range?
5. If not, what is the next free port to offer to the client?
6. Is the client's requesting duration valid?

The questions in this list can be broken up into two groups: lease management and client restrictions. The lease management portion is responsible for tracking existing leases, maintaining a list of free ports and deciding if client requests for specific ports should be allowed. The client restrictions portion is responsible for

enforcing the administrator's limits on client access, lease duration, port range and renew capability. Since the lease management functionality was the most critical to proper server functioning, it was written first. Client restriction capabilities will be described in section 3.2.5.

The lease management component of the `ServerPortManager` uses a standard Java `HashMap` in order to store the `DPRPLease` objects. This table maps from the primary key string to the actual lease object instance. Furthermore, when a new request comes in, a temporary lease is created, and used to query the `ServerPortManager` instance. In this manner, a series of routines were written that took a lease reference as a parameter, and then made checks against the `ServerPortManager`'s internal state. Thus, a lease could be checked to see if it contained a valid port. Furthermore, the `ServerPortManager` instance tracked all of the free ports, and could be queried to take a free port off of the list, for use in a server offer.

In order to store leases in the `ServerPortManager` instance, the `addReservation()` method was written. This method adds a reservation (lease) to the `ServerPortManager`'s lease map. Generally, the a server session will only invoke this method after it has determined that the lease in question has reached the "commit" stage. Since `addReservation()` is called at "commit" time, it is also where the lease commit functionality resides. Thus, the `addReservation()` method must not only add the lease to its in-memory map, but it must also write the lease to disk, using the output facilities of `DPRPLease`. In order to keep the in-memory map and the on-disk lease database in sync, the `addReservation()` method actually over-writes the entire on-disk database for every lease added. This is done by iterating over every key in the map, and outputting the associated value to the file.

But even with all of these routines in place, the lease management task was still not complete. Some facility was needed that could purge the server of expired leases. Ultimately, the "lazy approach" won out. The server wouldn't do anything to remove

reservations that had expired. Instead, the `addReservation()` method was modified so that additional processing would be performed as the leases were being written to the on-disk database. Instead of simply writing each lease out, `addReservation()` now checks the lease to see if it has expired. If an expired lease is found, then it is removed from the in-memory map, and it is *not* written to disk. Since this processing was done every time a lease was added, the server was assured that it wouldn't get into a position where it thought that it's port resources were exhausted, even though in reality, they were not.

3.2.5 The `DPRPRestriction` class

In the process of implementing the `ServerPortManager` class, the set of restrictions that the DPRP administrator should be able to place on client connections was determined. However, in order to test the client restrictions facility, some facility was needed in order to configure each restriction into the server. Essentially, not only was an object that bundled all of the restrictions for each client into one place needed, but so was a means for creating these objects and populating them with the proper data. The `DPRPRestriction` class was a natural way to bundle all of the client restrictions. The exact method for creating instances of this class, however, was less clear. Because adding more command-line parameters would have been unwieldly, a different strategy was needed.

The real solution was to create an ASCII text-based configuration file, that contained all of the client restrictions. This would make it easy for many sets of restriction objects to be instantiated in the server all at one time. Not only would this make testing easier, but it would also make things easier for DPRP server administrators. The first step in realizing this goal was to decided upon a file format:

```
host
     [renewAllowed = true | false]    default = true
     [numRetries = <#>]               default = 0 (unlimited)
```

```
[portRange = <#> - <#>]          default = 0 - 65536
[minDuration = <#>]              default = 500
[maxDuration = <#>]              default = 2147483647
[deny = true | false]            default = false
```

With this format in place, a parser could be written to load each host block out of the file, parse out any options present, and return a series of `DPRPRestriction` objects. On the Java side, the text-parsing efforts were greatly aided by the `java.util.StringTokenizer` class. Given a string, this class will parse it into tokens, based upon some notion of a separator character (the default, a space, worked great for this file format). This class greatly simplified the parser and allowed it to be developed quickly. With this functionality in place, it was easy to load many host-based restrictions into the DPRP server, which fulfilled the intended purpose.

3.2.6 The GUI client

A DPRP client employing a graphical user interface (GUI) allows end-users to easily reserve DPRP ports for "legacy" applications that are not DPRP-enabled. During the initial DPRP implementation, a command-line interface (CLI) client was created. As the implementation progressed, the command-line architecture became too difficult to extend, to the point that all of DPRP's features could not be stressed. The GUI DPRP client eased these limitations. Sun's Swing [25] toolkit was used to create the Java implementation of the GUI client. As figure 3.2.6 shows, the DPRP GUI client consists of a single window that is divided into two panes. The upper pane is used for acquiring a new DPRP lease, and the lower pane is used for managing outstanding DPRP leases.

The upper window provides the interface for allowing the client to acquire new leases. The transition from the CLI DPRP client to the GUI client was simply a matter of integrating the original client functionality into the graphical interface. This functionality is a simple message exchange between the client and the DPRP

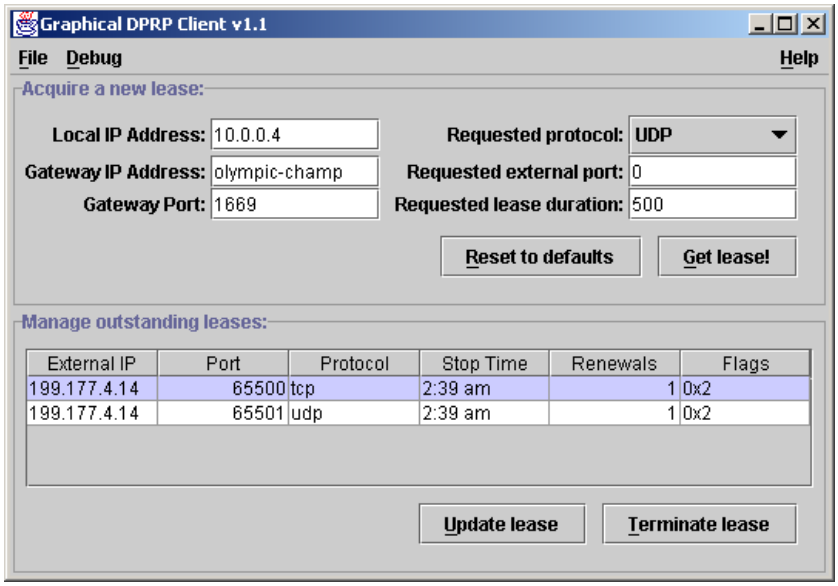

Figure 3.4: Screen capture of the DPRP GUI client.

server. All possible configuration elements are gathered from text-input boxes, and a new `DPRPClient` object is instantiated from those parameters.

Creating the lease management window, however, was much more complex. The existing DPRP client code was not suited to the task of managing leases. Although the proper thread structure was in place to renew a lease as needed, there was virtually no infrastructure to *notify* the user of the lease status[1]. As a result, some extensive modifications were needed to the existing DPRP client code in order to create a usable GUI.

Like most other graphical environments, Swing is event-driven. As such, all actions are represented in the form of events. An event is generated whenever a user clicks the mouse, when an error occurs or when a graphical widget updates its contents. In order to make a *natural* fit into the Swing framework, the core DPRP client code needs to emit Swing events. Specifically, the DPRP lease maintenance thread needs to emit Swing events whenever a lease is created, updated or destroyed. Unfortunately,

[1]The only infrastructure in place was the debugging output, which was really only suitable for viewing by the developer.

creating objects that actually *source* swing events is not a well documented aspect of Java. With great difficulty, the methodology for emitting a Swing event was extracted by example from the JDK sources. Once in place, DPRP client code that could emit Swing events proved to be quite useful , not only for setting up a dialog box informing the user of a status change, but also for implementing the requisite lease management functionality.

The next challenge was to decide upon a GUI widget that was capable of displaying all of the lease-specific information for several leases. Since each lease consists of several key data elements, and many leases needed to be displayed, a table structure was used. Swing provides the `JTable` component, which is exactly the component that was needed. In Swing, `JTable` is based upon a modified Model View Controller (MVC) [26] framework. The table itself is simply a view on the underlying data, which is encapsulated into a model. Because the DPRP GUI needed to display active leases, a custom model needed to be created. The custom model, `DPRPTableModel`, contains a list of `DPRPLease` objects that can act as a handler of DPRP lease events. Whenever a lease is updated, the model receives a lease event and is, therefore, able to update the tables view. With this final widget in place, the GUI DPRP client was completed. With this client, it was possible to arbitrarily update and terminate leases, thus testing those facets of the server. With the client code finally finished, the sample implementation was ready to perform port reservation on a live NAT gateway.

3.3 Building a DPRP Gateway

In order to determine if DPRP had any chance of achieving its goals, a NAT gateway that contained a DPRP implementation was needed. Obviously, it is beyond the scope of this project to develop a NAT router from scratch. Thus, an existing NAT solution was modified to recognize the DPRP protocol. The easiest way to do this

was to locate an open-source NAT implementation, so that the full source code was available. Since Linux is popular and fairly robust, it was targeted to be the DPRP NAT gateway reference platform. Specifically, since the packet mangling framework on Linux changes with every major kernel revision, the most recent one was targeted, which is "Netfilter".

3.3.1 About Linux/Netfilter

Netfilter [27] is the latest-and-greatest packet mangling framework for Linux. Essentially, the aim of this package is to be an end-all for every type of packet mangling, routing, and firewalling. Thus, NAT functionality is simply one *component* of the Netfilter suite. The Netfilter system is based upon a series of *tables*, each one representing a major component. For the purposes of DPRP integration, the NAT table was the most important. For each table, there are a series of *chains*, which are lists of packet-matching rules. Of the default chains that come with the NAT table, the DPRP integration needed to leverage two: the `PREROUTING` and the `POSTROUTING` chains. These two chains deal with packets that are mangled by NAT before or after the kernel routing code sees them. The `POSTROUTING` chain is good for doing source NAT, because the exit interface must be determined *before* the source address can be replaced. Similarly, the `PREROUTING` chain is a good place for destination NAT rules, because the destination address must be replaced with the proper internal address (according to the internal tables) *before* it can be routed. When a packet enters the gateway, it is thrown to the appropriate chain and tested against each rule in succession, until a "match" is found. Thus, in Netfilter, there can be multiple rules on a chain that vie for the same resource. The only rule that will ever be matched, however, is the one that appears first in the chain.

Parameter:	**Description:**
`-v`	Enables verbosity, which is only useful for debugging.
`-t nat`	Selects the "nat table.
`-A PREROUTING`	Selects the "PREROUTING" chain of the nat table. This means that the packets will be mangled *before* being passed to the IP routing code.
`-i eth0`	Specifies that the inbound interface for packets matching this rule should be `eth0`. *NOTE: This should not be hard-coded, and will be parameterized in a future release.*
`-j DNAT`	Causes the packet match to jump to the "destination nat" builtin target.
`-p <protocol>`	Sets the protocol (tcp or udp) type that packets must be in order to match.
`--destination <external IP>`	Ensures that the packets are bound for the external address of the gateway before they are mangled.
`--dport <allocPort>`	This is the actual port that we have reserved – if the packet is bound for the external IP address, and matches this port, then it will be NAT'd and routed into our client.
`--to <clientIP>:<allocPort>`	The ultimate destination for these packets – an IP address and port on the client that made this reservation.

Table 3.2: `iptables` command-line arguments

3.3.2 Integrating the Java DPRP Server with Netfilter

In order for the DPRP server to be able to setup live translations on a Netfilter gateway, it must install the appropriate rules. In Netfilter, rules are installed via the `iptables` command. Various parameters to this command allow the proper table, chain and rule properties to be selected. For the purposes of the DPRP server, all `iptables` commands executed will conform to the following template:

```
/usr/sbin/iptables -v -t nat -A PREROUTING -i eth0 -j DNAT \
   -p <protocol> --destination <external IP> \
   --dport <allocPort> --to <clientIP>:<allocPort>
```

Table 3.2 provides the details for each parameter and it's intended function.

The final caveat to mention about this port allocation code is that the DPRP server needed some way to avoid conflicts between ports reserved by DPRP clients and ports reserved in the gateway during the normal course of source NAT routing. Since the `iptables` command doesn't emit an error in the case of such a conflict, this became an administrative issue. The range of ports allowable for use with SNAT needs to be restricted and the left-overs given to the DPRP code.

Detecting a proper Linux host from Java

It is desirable for the server code to detect if it is running in a proper Linux/Netfilter environment. The benefits of this detection are two-fold: server administrators will receive immediate notification if the DPRP server likes their Netfilter environment, and the server isn't necessarily restricted to running on Netfilter-based hosts. If a proper Netfilter environment is detected, then the DPRP server will go ahead and actually allocate Netfilter ports for each client request. However, if the proper environment is not detected, then the server will run in a test mode, handing out "dummy" ports just as before. This gives the server an additional level flexibility. In Java, the OS detection was simply a matter of querying the system properties to

see if the OS name was "Linux", if the user was "root" and if the `/sbin/iptables` command exists. If all three of these conditions were met, then the server assumes that it should be possible to `exec()` Netfilter commands.

Using `exec()` in Java

Although it is not widely known, a Java program *can* execute a system binary. This feat is accomplished via `exec()` method, which Sun buries in the `Runtime` object. Although the use of `exec()` is not recommended by Sun, it is well suited to the DPRP server's Netfilter integration. In order to use `exec()`, an array of text strings containing the binary to be executed and all of the command-line parameters must first be constructed. Because `exec()` doesn't have the luxury of a shell, each parameter that would normally be separated by a space had to be placed into its own location in the array. This array is then handed off to `exec()`, which starts a new process to run the binary, returning a `Process` instance to the caller. With the `Process` instance, the server is able to wait for the command to finish executing, at which point, the exit status is checked. If the exit status is non-zero, then an error occurred and the server would not complete the request. Otherwise, the lack of error is taken to mean that the port had been successfully reserved.

Modifying the `ServerPortManager`

If the OS detection code finds an appropriate Linux-based machine with Netfilter support, it sets a flag in the Server Port Manager instance. This effectively enables "linux mode" in the Server Port Manager. In this mode, the `allocFreeExternalPort()` method will utilize the `exec()` facility in order to place a reservation rule for the client in the "destination NAT" Netfilter chain. The `delReservation()` method will utilize the `exec()` facility in order to delete a rule from the "destination NAT" chain.

3.3.3 The "Netfilter Problem"

Unfortunately, during testing, a problem was uncovered with Netfilter, that merited a great deal of investigation. From time to time, a Netfilter machine that was configured with destination NAT port forwarding, would simply stop forwarding certain ports. Since the entire aim of DPRP is to *enable* port forwarding, it was imperative to resolve this issue. In order to get a handle on this problem, a static destination NAT rule was added to permit X connections from the outside in to the development workstation. So that this rule didn't conflict with any X server running on the gateway, the rule redirected port 6001 on the gateway to port 6000 on the development workstation. Thus, if the "gateway:1" were set on remote machines, X programs would subsequently show up on the development workstation, at the "workstation:0" display. This all worked great, at first.

However, at seemingly random times, Netfilter would simply stop forwarding the port 6001 reservation. Rebooting the gateway would make the forwarding start working again, so that, combined with the fact that this was a static rule (and not a DPRP-installed rule), narrowed this down to a Netfilter problem. Further investigation showed that whenever the external interface re-negotiated its DHCP lease, the port forwarding would stop. This behavior was verified manually by causing the DHCP client to renew it's lease, and witnessing the immediate halt of port forwarding. So, the solution seemed obvious – the external Ethernet interface was set to be statically configured. Unfortunately, the static IP address did not resolve the port forwarding stoppages. This time, the stoppages could not be traced to any particular system-level event. The gateway was upgraded to a newer version of RedHat, along with which came kernel version 2.4.7.-2. This seemed to improve port forwarding stability, but the problems continued to persist.

The final realization on this problem occurred when, during an outage, Netfilter was instructed to delete the 6001 port forwarding rule. Surprisingly, Netfilter refused

to delete the rule. Further, the error message seemed to indicate that the rule didn't even exist – yet it showed up in the list of destination NAT rules. Thus, it seems like Netfilter just "forgets" about certain rules after a period of time. Testing further, Netfilter was directed to re-add the rule port 6001 rule on top of itself. A listing of the rules now showed two identical rules, both attempting to forward "gateway:6001" to "workstation:6000". However, adding this second rule fixed the X forwarding. This makes sense – Netfilter simply walks down the chain of rules, looking for a match. Since it was no longer-seeing the first forwarding rule, the X port forwarding was "broken". With the addition of the second rule, however, Netfilter was now able to "see" the port 6001 forwarding rule, and thus forwarding service was restored. So, it seems like there is some sort of bug in Netfilter, that could quite possibly be fixed in newer kernel versions. Fortunately, it was possible to work around this bug, and as such, it didn't hamper the development of a DPRP server with Netfilter support.

3.4 Embedding DPRP into Napster

The most important aspect of DPRP is that client functionality can be embedded into existing applications in order to make NAT gateways a "fully transparent" user experience. In order to demonstrate the promise that embedded DPRP clients hold, a peer-to-peer file sharing network was chosen to be modified as a proof-of-concept. Back in its prime, Napster [9] was the largest peer-to-peer file sharing network in existence. Although Napster's dominance has waned for political reasons, it is still interesting from the DPRP perspective. First, Napster is server-based which means that clients connect to a central server that acts as a central repository for file sharing information. Searches are performed on the server, the results of which can be used to download files from the appropriate peer. Second, the limitations of Napster and NAT gateways are well documented and understood. Additionally, the Napster protocol

[28] has been reverse-engineered and documented by the open-source community. With this protocol, the open-source community was able to produce a free Napster server, entitled OpenNap [29]. An open Napster server was important because it could be modified to support DPRP if needed. Also, it would be much easier to test the DPRP Napster implementation with a private Napster server. Finally, because Napster has been around for several years, there are a wide variety of clients available, many of which were suitable for modification with DPRP.

As proof-of-concept, a pre-existing Napster client was selected to take on an embedded DPRP client. The simplest way to embed DPRP into a Napster client was simply to have the embedded DPRP client negotiate a TCP port at startup, and maintain that port for the entire lifespan of the Napster client. While this might not be the most efficient approach, it mitigated all of the issues with negotiating a specific port for specific transfers. Furthermore, using this approach, the OpenNap server did not need to be modified. It would simply advertise the IP address and port number that the client assigned to it, which in the DPRP case, would be the external IP address of the gateway and the leased external port.

Of the many open-source Napster clients available, XNap [30] was selected as the basis for the DPRP implementation. XNap is a Napster-based file sharing client, written in Java, with the code freely available under the GPL [31]. This is the only such Napster client that is still under active development. With effort, it was determined that the existing Java DPRP client could be added to XNap. This client would function without a controller, so it would not be able to notify the user if a lease was acquired or of any changes in lease status. If the embedded DPRP client could not establish a lease, then it would shut down the entire XNap client. Furthermore, the XNap preferences were modified so that end users could not change the XNap listening port, since that was now in control of DPRP.

With the embedded DPRP client in place, the XNap client was able to function

on the Napster network as if it was not behind a NAT gateway at all. External clients were able to download songs off of the DPRP-enabled XNap client with ease. In the situation where the DPRP-enabled XNap user wished to obtain a song from a legacy Napster client stuck behind a NAT gateway, the remote Napster client was able to "push" the song onto the XNap client with the open port. Thus, this demonstration proves that embedded DPRP functionality can truly and effectively remove the inbound connection limitations of NAT gateways.

Chapter 4

Evaluation

4.1 Security Implications

Aside from facilitating greater efficiencies in the use of IP addresses, NAT gateways also provide additional security benefits hosts in the private address space. Since the NAT gateway doesn't allow incoming packets to internal hosts for which a translation isn't already established, this prevents malicious packets from making contact with internal hosts. This means that things, such as port scans, are easily defeated. Furthermore, flood attacks such as the "SYN flood" are also stopped at the gateway. As a result, many installations of NAT gateways are done with these security properties in mind. Consequently, any modifications to the functionality of a NAT gateway must keep security concerns in mind.

This protocol doesn't inherently weaken the security properties of the NAT gateway. It does make it easier for *users* to weaken these properties, and this is the concern that administrators must keep in mind. The ability for NAT gateways to forward "unsolicited" traffic from certain external ports to specific hosts has been available for a long time. This behavior has always been controlled by the gateway administrator. This protocol puts the power of controlling this functionality in the hands of the user. Thus, users will be able to receive unsolicited packets directly on

their machines. This means that administrators won't be able to rely on the NAT gateway to provide total security against incoming connections.

A NAT gateway that implements this protocol will still give the administrator some control over it's use. Certain hosts could be denied access to the protocol. Other hosts could be restricted to a certain number of valid leases. Furthermore, restrictions could be placed on lease length and number of renewals allowed (to prevent the running of web servers, for example). All of these options should provide enough "granularity", so that the balance between functionality and security can be set to match the administrator's desires.

4.1.1 The "worm" implication

In considering the latest round of IIS and e-mail worms, it's possible that this protocol could slightly aggravate the security problem that these worms represent. Consider a public webserver, that is operating via port forwarding from behind a NAT gateway. If a worm is able to infect this public webserver, then it would then be free to infect many other machines that are on the private network. Under normal circumstances, the worm wouldn't be able to penetrate these machines directly, because they would be protected by the NAT. If the worm were to contain an embedded DPRP client, however, it could negotiate external ports with the NAT gateway. These ports could then be advertised to the worm author, who could then penetrate these previously protected machines.

4.2 Complete end-user transparency is hard

One of the main goals for DPRP was for it to be a protocol that lives behind the scenes; something that end users' benefit from on a daily basis, but have no idea even exists. DPRP is the last piece of the NAT puzzle – something that makes NAT

truly transparent to the user[1]. During the course of creating the DPRP reference implementation, however, it became evident that this goal of complete transparency to the end-user is still quite difficult to achieve. The initial assertion was that embedding DPRP into mainstream applications would provide the desired degree of transparency. Unfortunately, there is more to transparency than the location of the DPRP client.

Essentially, what I failed to realize is, that for proper operation, any DPRP client needs access to the following pieces of information:

1. That it is behind a DPRP-enabled NAT gateway. While it is possible for a client to determine that it is behind a NAT gateway [10; 32], it really isn't feasible to detect if said gateway is DPRP-compliant, because the DPRP server may not be running on the standard UDP port.
2. The IP address of the NAT gateway.
3. The UDP port number of the DPRP server on the NAT gateway.
4. The local IP address to use if multiple interfaces are present in the system.

In current incarnations, all of these parameters must either be guessed (which is usually crude), or be user-supplied. If a user has to supply DPRP-specific values, then DPRP is not transparent. Ideally, some additional infrastructure is needed so that the DPRP client can query the network for all of these parameters. The IETF is developing a protocol entitled "Service Location Protocol" (SLP) [33–35] that may fit the bill. Essentially, SLP allows *services* to be located in the network transparently, without user intervention. The DPRP client could request a handle to the DPRP service, and it would then gain all of the information that it needed in order to function properly without additional user intervention. At this point, the integration of DPRP with SLP is a topic for future investigation.

[1]Of course, NAT gateways will never be truly transparent to the "power users" who are always trying to do something outside of the mainstream

Chapter 5

Conclusion

As realized, the DPRP solution has met the design goals that were set for it, and is poised to solve the NAT in-bound connectivity problem in an effective manner. The DPRP protocol itself is simple and lightweight, and should be portable to a wide range of networked computing devices. The DPRP reference implementation has properly validated the protocol. The reference implementation's ability to reserve ports on an actual Linux/Netfilter NAT gateway validates the concept of dynamic in-bound port reservation on a NAT gateway. Finally, the modifications to the XNap Napster client show that DPRP can solve real NAT-related connectivity problems for real applications.

DPRP stands up well to the pre-existing solutions that were reviewed in chapter 2. When compared to Dan Kegel's UDP hack, DPRP excels because it supports TCP and can function with NAT gateways that are forced to change port numbers during the process of translation. Although Dan Kegel's solution requires less implementation overhead, the overhead to implement DPRP is not much more than for Dan's minimal solution. When compared to RSIP, DPRP is drastically simpler to implement, requiring only that additional functionality be added to the NAT gateway and the addition of DPRP clients. However, DPRP does make some sacrifices in terms of the amount of connectivity offered. RSIP is still a more complete solution.

Finally, DPRP requires less infrastructure changes than AVES, while offering nearly the same level of connectivity. The only advantage of AVES over DPRP, is that the AVES clients have greater flexibility in the ports that they may use, while DPRP users must share one port space.

The next step for DPRP is to be recognized as a standard by the Internet community. Because standardization guarantees inter-operability and protocol stability, it is necessary to encourage developers to include it in their products. More work is needed on the reference implementation in order to make it easier for developers to add DPRP functionality to their products. The feature-set of the current reference implementation should be expanded to make it more easily reusable. Other implementations should be written in popular languages such as C. With these changes in place, the barrier of entry for DPRP developers should be low enough to enable widespread adoption. Ironically, once widespread adoption of DPRP is achieved, the next step for the protocol will be to die. It is predicted that IPv6 will eventually take over the Internet. When it does, NAT gateways and DPRP along with them will largely be rendered irrelevant.

References

[1] K. Egevang and P. Francis. "The ip network addres translator (nat)." RFC-1631 (May 1994).

[2] J. Postel. "Internet protocol." RFC-791 (September 1981).

[3] V. Fuller, T. Li, J. Yu, and K. Varadhan. "Classless inter-domain routing (cidr): an address assignment and aggregation strategy." RFC-1519 (September 1993).

[4] S. Deering and R. Hinden. "Internet protocol." RFC-2460 (December 1998).

[5] C. Shinn. "Ipv6 today and tomorrow." http://www.hill.com/news/archives/mtsarticle_cshinn.shtml (January 2000).

[6] G. Tsirtsis and P. Srisuresh. "Network adress translation - protocol translation (nat-pt)." RFC-2766 (February 2000).

[7] G. E. Moore. "Cramming more components onto integrated circuits." *Electronics* **38** 4 (April 1965). Http://www.intel.com/research/silicon/moorespaper.pdf.

[8] J. Postel and J. Reynolds. "File transfer protocol (ftp)." (1985).

[9] N. Incorporated. "The napster music sharing system." http://www.napster.com (1999).

[10] D. Kegel. "Nat and peer-to-peer networking." http://www.alumni.caltech.edu/~dank/peer-nat.html (1999).

[11] J. Postel. "User datagram protocol (udp)." RFC-678 (August 1980).

[12] J. B. Postel. "Transmission control protocol (tcp)." Technical Report 793, SRI International (1981).

[13] L. Phifer. "Realm-specific ip for vpns and beyond." http://www.isp-planet.com/technology/rsip.html (2001).

[14] S. Kent and R. Atkinson. "Security architecture for the internet protocol." RFC-2401 (November 1998).

[15] USENIX Annual Technical Conference 2001. *A Waypoint Service Approach to Connect Heterogeneous Internet Address Spaces*, Boston, MA (June 2001).

[16] D. Senie. “Nat friendly application design guidelines.” (2000).

[17] P. Mockapetris. “Domain names - implementation and specification.” RFC-1035 (November 1987).

[18] I. Consortium. “Berkeley internet name domain (bind).” http://www.isc.org/products/BIND (June 2000).

[19] P. Ferguson and D. Senie. “Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing.” (1998).

[20] R. Droms. “Dynamic host configuration protocol.” RFC-1531 (1993).

[21] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee. “Hypertext transfer protocol – http/1.1.” RFC-2616 (June 1999).

[22] S. Microsystems. “The java language: A white paper.” http://java.sun.com/doc/overview/index.html.

[23] J. Goldschmidt and N. Stone. “Jdhcp - dynamic host configuration protocol for java.” http://www.dhcp.org/javadhcp/main.html (1999).

[24] M. A. Poolet. “Sql by design: Good database design starts with the right primary key.” http://www.sqlmag.com/Articles/Index.cfm?ArticleID=5113 (1999).

[25] K. Walrath and M. Campione. “Creating a gui with jfc/swing.” http://java.sun.com/docs/books/tutorial/uiswing/index.html (2001).

[26] S. Burbeck. “Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc).” http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html (1992).

[27] R. Russell, H. Welte, J. Morris, and M. B. et. al. “The netfilter project: Packet mangling for linux 2.4.” http://netfilter.samba.org/ (2000).

[28] drscholl@users.sourceforge.net. “Napster protocol specification.” http://opennap.sourceforge.net/napster.txt (march 2001).

[29] drscholl@users.sourceforge.net. “Opennap: Open source napster server.” http://opennap.sourceforge.net (September 2001).

[30] F. Berger, Y. Leist, F. Student, M. Ransburg, and S. Pingel. “Xnap - a pure java filesharing client.” http://xnap.sourceforge.net (September 2001).

[31] F. S. Foundation. “Gnu general public license.” http://www.gnu.org/copyleft/gpl.html (June 1991).

[32] K. Lin. “ipcheck.py - a python script for use with dyndns.org.” http://ipcheck.sourceforge.net (December 2001).

[33] C. Systems. "An introduction to slp." http://www.openslp.org/doc/html/IntroductionToSLP (June 2001).

[34] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. "Service location protocol." http://www.openslp.org/doc/rfc/rfc2165.txt (June 1997).

[35] E. Guttman, C. Perkins, J. Veizades, and M. Day. "Service location protocol, version 2." http://www.openslp.org/doc/rfc/rfc2608.txt (June 1999).