IPv4 to IPv6 Bump In Stack

Tabitha Tipper
t.tipper@lancs.ac.uk
BSc (Hons) Computer Science
Library Card Number: 0103292

Tuesday 16th March, 2004

I certify that the material contained in this dissertation is my own work and does not contain significant portions of unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and associated documentation.

Abstract

This project is based on integrating two different versions of the IP (Internet Protocol). It enables an apparently IPv4 host to exist on an IPv6 only network, while appearing to be IPv4 to the applications and users of it. This project has been implemented in Linux (2.4.x) as a pair of kernel modules to filter network packets by means of a transparent Bump In Stack (BIS) and a kernel module and userland process that together handle the IP address inconsistencies which will occur in this host.

Contents

Working Documents may be found at: http://www.lancs.ac.uk/~tipper/fyp

1 Introduction

A "bump in the stack"[56] at its highest level overview is a totally transparent transitioning method between two protocols, in this case IPv4[36] to IPv6[8] (henceforth referred to as v4 and v6).

It can also be seen as a method of allowing v4 applications to be used on a v6 network. Whereby all their network traffic would be seamlessly modified so that they think they are dealing with an v4 network when in fact they're existing on an v6 one, while the network and all servers on it think they're just dealing with another v6 host.

So a BIS (Bump In the Stack) is useful for the point of transitioning whereby the network is running v6 but all the users applications only work with v4, and newer versions supporting v6 haven't been released yet.

1.1 A basic overview of the architecture.

This project uses a fairly modular architecture to perform its operations. It can be broken down into two main sections, the DNS[35, 48] (Domain Name System) section and the packet manglers.

The DNS section features two programs, a transparent DNS proxy and a DNS results hash. The DNS proxy is designed to filter DNS traffic, catching incoming DNS replies. The idea being that it will catch incoming replies about v6 hosts, store a mapping between that v6 address and a v4 address in the results hash and rewrite the DNS response to include that v4 address before returning it. This way v4 programs may consistently address v6 machines using a v4 address which will be mapped over.

The packet manglers change all incoming v6 packets to v4 and all outgoing v4 packets to v6. This means that to the users of the machine it appears to be an old v4 machine, but it exists to every other machine on the network as an v6 host. These filters use the data stored in the results hash to translate the addresses over so that to the inside the rest of the network appears to be v4.

1.2 This Report

This report beings by covering the background to the whole project: The characteristics of v4 and v6, transitioning methods between them and the a chapter of technical background on the Unix OS. These chapters are largely not specific to this project but cover areas that support it. Then the report

moves on to the design of the system, its individual components and the implementation of those components. These chapters are an in depth coverage specific to this project. Finally there are two concluding chapters, one on the evaluation of the system and one as a general conclusion to discuss the project overall.

2 Background

This chapter of the report details the background of the various technologies involved. It will start by covering v4 and its limitations, followed by covering v6 and v6 transition methods. This will be followed by a section on GNU/Linux[29, 15].

2.1 IP version 4 and its limitations

IP is the Internet Protocol[36, 4], it is the protocol that underlies most communication on networks today, such as TCP/IP and UDP/IP. IP was designed to create a packet switched network i.e. one where discrete blocks of data are moved between devices. It does not dictate end-to-end reliability, flow control, sequencing or timing. It simply makes a best effort to deliver each packet.

IP version 4 was originally created for the ARPANET[17] before there was even an Internet. It predates DNS and most modern networking technologies.

2.1.1 Address space

The largest weakness of v4 is its address space[53]. Each v4 address is only 32 bits (4 bytes) long, arranged into so called "Dotted Quads", four integers between 0 and 255 usually written separated by dots (e.g. 212.159.102.2). This means that there is a maximum of 4,294,967,296 uniquely identifiable and fully routable hosts on the Internet at once.

The allocation of this space is controlled by an organisation called IANA, the Internet Assigned Numbers Authority. They coordinate the distribution of blocks of v4 addresses to the Regional Authorities (RIPE, ARIN, APNIC, JPNIC, LACNIC etc), who in turn distribute the address space to providers who actually use them. It is however up to IANA to try to ensure that the space is allocated sensibly and fairly (as their tag line states: "Dedicated to preserving the central coordinating functions of the global Internet for the public good.") and they have total control over it.

While it may sound like IANA have a lot of addresses to allocate most of it is currently allocated[22]. At a recent count (17/02/04) IANA had 89 Class A address blocks free, of the 221 that they own total[10]. This gives them 16,777,216x^{89} addresses left to distribute, which is less than a billion. The current world population stands at 6,353,213,900[60] on 09/03/04. Currently most of the world's population is without computers, but since half the worlds population is lacking adequate access to such basic amenities as

clean water[14], and thus cannot utilise IP addresses. But looking to the future we can foresee a time when each person on the Earth has several always connected devices, such as computers, mobile phones and other pieces of technology, many of which are mobile and moving between networks. On top of this there will be substantial infrastructure needed to support a truly global communications network. Since IANA cannot provision such an effort at the minute it seems that v4 as it stands must be considered a passing phase of technology that is unsustainable into the future, as to use it for such an endeavour would be ill advised, therefore we must move to v6 at some point.

2.1.2 Classes

When v4 was created a decision was made to break the available address space up into classes of addresses[42] to create easily manageable chunks of address space. However while this action simplified the management of the IP address space it also introduced an inefficiency into the implementation.

5 classes of address were created, named A to E. A is a block of 16,777,216 hosts, B is a block of 65,536 hosts and C is a block of 256 hosts. Class D is reserved for use with multicasting and class E is a block of IP addresses reserved for future use. Also several other blocks were reserved for various uses[20] common instances of these include: 192.168.x.x, 10.x.x.x and 172.16.0.0-172.31.255.255 which are all reserved for internal network use[41]. Whereas 127.x.x.x is reserved for loop back traffic (i.e. any traffic addressed here should never leave the host but loop back to its network input instead).

The class D and E addresses are not used to address public machines, so this leaves the rest of the entire range of IP addresses carved up into classes A - C. As soon as a site is connected to the Internet (under this system) it needs to be given an entire class C. Assuming that many sites only need one or two addresses then this wastes over 200 addresses. Once a site reaches over 256 full addressable machines it would need an entire class B, which would waste over 65,000 addresses and so on. This allocation system is obviously inefficient and wastes much of a limited resource.

This class based distribution was stopped in the early 90's when it was realised to be a terribly inefficient way of allocating addresses. Not only did it cause waste but because the address space was chiefly carved up chronologically with no thought to network topology it caused what became known as "The Routing Table Explosion". A temporary fix involving issuing multiples of a single class and not just stepping up from one to another was used during the 90's but was surpassed by the current work around.

In order to fix these problems a new method of breaking up the address space was created. CIDR (Classless Inter Domain Routing)[13] which allowed an unprecedented granularity, splitting blocks between different AS (Autonomous Systems, i.e. different organisations or groups on the Internet) at the bit level. Generally this was used to break up individual class C blocks. CIDR reduces the waste of addresses as only the amount of IPs that are needed at each site are given to it. More importantly it also reduces the routing table effects as the Internet can route to a single router at the AS that is responsible for the class C which then takes care of distributing the packets to each owner of an address.

2.1.3 NAT

NAT (Network Address Translation)[11] is one of the more common methods of building networks. It dramatically reduces the amount of v4 addresses that an organisation needs to request for its network, and as a side effect helps secure the hosts behind it. The idea behind NAT is that you have a large group of machines on reserved internal-use only IP addresses (like 10.x.x.x) and a machine that is set to their default gateway, this gateway machine performs NAT for them.

NAT works by rewriting the headers of network packets, chiefly those dealing with the addresses. The idea being that the NAT machine appears to be all of the external Internet to the hosts inside, and appears to be all the internal machines to the rest of the Internet. This means that you only need one fully routable IP address to have a large number of machines accessing the Internet, currently most companies and many universities make widespread use of NAT to reduce their use of the IP address space.

The chief problem with NAT is that while a large number of machines without real IPs can access resources on the Internet they are not full Internet hosts. The Internet was designed with full end-to-end connectivity in mind. With NAT internal machines may initiate connections to the Internet but hosts on the Internet may not initiate connections to internal machines, so they effectively become half-hosts, able to passively receive information the the rest of the network but unable to serve any themselves.

NAT machines also present a single point of failure for network outages or security risks. So if the NAT machine crashes or comes under a [D]DoS ([Distributed] Denial of Service) attack then all the internal machines will be unable to access the Internet. Similarly if the NAT machine is compromised then not only can all network traffic going into and out of the network be

sniffed but also the NAT machine may be used as a platform for further attacks on the internal machines.

2.2 IP version 6 and its advantages

v6 is the new version of IP and has been written with many of the lessons learnt from v4 in mind. It uses a 128 bit (16 byte) long address field which allows for $3x^{1038}$ addresses. This size of address space removes one of the worst issues with v4. Also v6 doesn't have classes of addresses, and unlike v4 it has been broken up around the idea of geographical splits.

An v6 address is 128 bits long, this makes it 4 times longer than an v4 address, and as demonstrated above the amount of address space is exponentially larger than v4. However this space is not unstructured and a v6 address can be broken down into sections[18] to define the hierarchy that look after the leasing of the addresses[2]. Primarily v6 addresses are broken into two main chunks, that defining the location of the v6 router for the host, and that defining the hosts unique interface ID. Both of these are 64 bits. The Interface ID is generally taken from the network card of the machine. When a v6 host boots for the first time if its lacking a statically allocated v6 address it broadcasts (or more technically multicasts as those have replaced broadcast in v6[19]) to the network, performing router solicitation, that is looking for a router that will manage its traffic. The router that accepts this request will take the 64 bits of identifier the host provides and combine them with its own address to return a fully routable 128 bit long address.

The first 64 bits of the address is broken down into multiple aggregation sections to enable easy routing, management and location of responsibility for each address. These include a TLA (Top Level Aggregation Identifier), NLA (Next Level Aggregation Identifier) and SLA (Site Level Aggregation Identifier), as well as 8 reserved bits between the TLA and the NLA for future use.

The full implications of these boundaries are not fully examined yet but one idea of a structure is that IANA sits at the root of the hierarchy. They hand address blocks to the Regional Internet Registries (e.g. RIPE, ARIN, APNIC), who in turn hand address blocks out to National and Local Internet Registries, for example ISPs and Telcos. These people finally assign them to the end users of the addresses.

v6 also includes increased support for multicast networking[24], the idea of an efficient one-to-many communication method using IP. v6 also creates anycast networking[25], which is similar to multicasting, in that its designed

for a central server to send to many clients, but only sends the packets to the nearest host. That host then being responsible to forward it on.

v6 has been designed to be faster for routers and hosts to process. Its header fields have been simplified and reduced in number from those of v4 (from 13 down to 8) this should result in an increase in the amount of packets a machine can process in a set amount of time.

To ease the change over from v4 to v6 there is a reserved block of addresses as large as the current block of v4 addresses. These map onto each other one for one[19], these addresses are known as v4-mapped v6 addresses. This enables current routing and address allocation to continue working during and after the change over to v6. This is the block of address space where the first 80 bits are entirely 0s, followed by 16 bits set to 1, finally followed by the v4 address as the last 32 bits. There is also a similar system (known as v4 compatible v6 addresses) which is 80 0s followed by 16 bits of 0s then the 32 bit v4 address. This format is used for routers which carry v6 traffic using transitioning methods over v4 networks.

QOS (Quality Of Service) is properly supported under v6. The headers support both IntServ[5, 6] and Diffserv[3] types of QOS. IntServ uses the "flow label" field, and DiffServ uses the "traffic class" field. While there is still no implemented standard for v6 flow label fields there is an agreed standard for the traffic class field, since with v4 neither of these methods of QOS is standardised at all this this represents a major step forward. Since there is a single standard for v6 networks to use for marking at least some of their QOS packets this will help domains maintain their QOS SLAs (Service Level Agreement) with each other.

v6 also includes optional sets of headers that can be used to signal that the contents are encrypted and include an MD5[43, 1] checksum against which the encrypted contents can be checked to verify they have not been tampered with. This adds an additional built in layer of security to the whole protocol, whereas for v4 each application would have to do its own encryption at the application layer.

Mobile devices are now seen as an important part of networking, and as such v6 supports them to a greater degree, the router solicitation method of gaining an address means mobile devices can move between networks gaining and losing addresses. To keep them from being lost by the hosts they communicate with, mobile devices normally have a "home address" to which packets should be sent in the event a client losing contact with them. A device at this home address will forward any packets sent to it to the current known working

address of the mobile device. The final step is for the mobile device to send updates to those communicating with it to inform them of its new address. So while moving between networks will create some delay in communications the flow between hosts will not be broken. The increased address space available to v6 means that this use of several addresses for one device is considered far less wasteful.

2.3 Transitioning to V6

It seems at this point inevitable that the world of networking will one day move from v4 to v6. This shift may be caused by political pressure from those interested, or from those with a vested interest in the technology. Alternatively it may be that the limitations of v4 just become too large to deal with comfortably. In any event the transition is likely to be a slow and piecemeal affair.

During the process there are likely to be so called islands formed. These will be separated networks that are largely cut off from the rest of the world, running the non-dominant version of IP. During the initial phases of the transition v6 islands will exist, eventually there will exist a plateau where the v6 islands and now v4 islands exist in balance, the network carved up between them and many transitioning methods linking them. Finally the balance will turn against v4 and the older protocol will then exist in separate islands, communicating with a larger, newer, v6 speaking Internet.

2.3.1 Tunnelling

Tunnelling is a method not of connecting islands of one protocol to a network of another protocol (for example a v6 island to a v4 Internet), but instead of just connecting islands together to form their own larger network. This is a commonly used technique to link research networks running v6 together. The methods described here are both encapsulation methods i.e. they wrap one protocol in another to hide it. For example v6 may be encapsulated in v4 simply by placing the entire v6 packet (including its headers) in the data section of one of more v4 packets.

6to4 The 6to4 tunnelling method relies on having two routers with v4 addresses. These routers have their external v4 address with the prefix of 2002 for their v6 address and appear to act as normal v6 routers however they encapsulate the v6 packets inside v4 before sending them over the Internet as normal packets. The router at the other end (which also runs 6to4) reassembles them, unwrapping them from their v4 encapsulation. After this they continue on their way. The advantage of this is that it allows you to link various separated hosts via a cheap connection to the

Internet and it requires little configuration outside the routers. However it cannot provide access to the wider Internet for your v6 hosts.

6over4 6over4 acts in a similar way to 6to4 but is designed more for use on a single site where you have scattered v6 hosts that need to find a v6 router for their traffic. It uses v4 multicast traffic in order to locate the v6 router on the network. Once found it encapsulates the v4 traffic inside the data segment of a v4 packet and sends it out over multicast. This method doesn't help the v6 hosts communicate with the v4 hosts, thus limiting its usefulness as a means to transition to an v6 Internet. 6over4 scales quite badly when a lot of hosts on a network use it, creating more traffic as multicast is largely a broadcast based system.

The tunnels themselves need ways of being setup. The methods described above are mostly hand configured, in that static tunnels are created and all the machines involved are carefully configured to work well together. There are a number of ways to automatically setup and tear down tunnels as they are needed, which reduces the time required to maintain an island, and allowing traffic to be encapsulated and sent down them to other islands as its needed.

Tunnel Brokers Tunnel Brokers are automated systems that can be contacted by the user to arrange for a tunnel to be set up for their traffic. These systems may be contacted via the web or presumably any other interactive system. They put the state for the tunnel into place on the network (whatever kind of tunnel it is) and then returns a script to the user which will configure their host to use the tunnel. Tunnel brokers are mostly a good method of reducing administration time, since they automatically set the state in the network and at the hosts machine. Theoretically they will set both correctly (if setup right) which will reduce the risk of user error.

Terado Is an automated method to allow hosts behind a v4 NAT to communicate with v6 hosts via the Internet at large. It works by encapsulating the v6 traffic inside single UDP packets (which carries some risk as this doesn't guarantee delivery but the TCP stacks at each end should resend the encapsulated packet enough to ensure delivery). These UDP packets should be allowed through the NAT relatively unharmed, as UDP is less often regulated than TCP traffic. The host directs these to a Terado server, which unpacks the v6 packet and essentially proxies it onto the v6 network.

ISATAP Intra-site Automatic Tunnelling Protocol (ISATAP) is a method for a lone v6 host with only a v4 network between it and its v6 router to work as normal. The v6 host sends a DNS query to its name servers asking for a well known ISATAP named host. It gets the v4 address of that host back. It then encapsulates v6 inside v4 to communicate with it over the v4 network, soliciting it for an IP address. Once it has that it tunnels v6 packets inside v4 to the router, which then deals with sending the traffic down a tunnel to the rest of the v6 network.

2.3.2 Translation

A direct way of integrating islands of one protocol to islands of another protocol are the translation methods. These are techniques whereby v6 hosts in an island may communicate with the v4 Internet at large as if they were a part of it, and in the future such methods will allow v4 islands to communicate with v6 islands during the transition to a v6 Internet.

SIIT Stateless IP/ICMP Translation[27, 59] is a method that is usually implemented on routers. This method translates the headers of the packets passing between it to ensure that the network on the other side sees v4 packets and the island inside only sees v6. This means that as far as each is concerned all networks are like theirs. The only problem with this technique is that it doesn't support multicast, nor does it support optional header sections (such as the v6 options). Also like more traditional NATs this translation method is stateless so the SIIT machine appears to be all hosts on the internal network.

NAT-PT NAT/Protocol Translation[55, 59] is simply put a router that translates between two protocols. This means that on one side of the network the traffic is v4 for example, and on the other side is v6. This is one of the more useful techniques in order to transition between the two systems as it means you can slowly move over in islands and each island will think the entire Internet speaks the same IP version. The NAT-PT router also has a pool of v4 addresses so that it appears that full end to end connectivity exists and unlike SIIT a session can exist between a host on the inside and a host on the outside. However since this method requires several v4 addresses per NAT-PT router space.

SOCKS SOCKS servers are essentially application layer gateways. Applications use the SOCKS libraries, which wrap around standards sockets and communication with the server. The server manages connections to the outside world, since it normally performs NAT (SOCKS is used for regular v4 networks which don't want full NAT) then it can translate the protocol on the fly as well.

BIS Bump In the Stack is similar to SIIT, and can simply be thought of as a translation method for a single machine. In this case the internal view of the network is of one protocol (e.g. v4) and the rest of the network sees traffic emerge from the machine fitting another protocol only (e.g. v6), neither being aware of the other. BIS is described further in many other parts of this document.

2.3.3 Dual Stack

One final transitioning technique is simply to ensure that all hosts have both a v4 stack and a v6 stack, and can talk both protocols. However this solution has problems in that it does nothing for many of the v4 issues, it just enables us to live with them on more congested networks.

DSTM (Dual Stack Transitioning Method) is a modified version of this, in that you have all machines with two stacks (v4 and v6) but only the v6 one by default has an address. If a host on the network requires a v4 address to communicate with another host (or a request comes in for that FQDN (Fully Qualified Domain Name) in v4 protocol) the host communicates with a DHCPv6 server (Dynamic Host Configuration Protocol) and requests a temporary assignment of a v4 address. Once its finished the communication it releases the address. This way a relatively small number of v4 addresses can be used by a number of v6 hosts, assuming that not too many are requested for at once and that the DHCP server doesn't get flooded with requests.

2.4 DNS

Understanding a lot of this project requires understanding parts of how DNS operates. DNS is mainly used to control the mappings of an FQDN to an IP address (or addresses). This means that the user and their applications can refer to a resource by its name (for example cent1.lancs.ac.uk) while the network itself can deal with numbers, this helps both parties as people find it easier to deal with symbolic names and the network routing relies on being able to divide the world into numerically named hosts, some of whom route traffic to other hosts. Traditionally nameservers are run on port 53 and accept UDP packets containing a query. They search their own information, or if they don't contain the information required check other name servers further up the hierarchy.

DNS traffic is composed of a header section, which includes a flag for if this packet is a query or a response, as well as the number of questions and "resource records" attached. Generally queries are sent to the server with a number of questions attached to them. The server then returns the same header section, with the flag set as a response, it has the questions attached in the same order and all the responses are attached in formatted blocks of data known as "resource records". These contain the information asked for (e.g. the IP address that is assigned to that name) or other pertinent information. If DNS cannot answer a question, even trying from the top of the hierarchy, then it will just return the headers and questions with no resource records attached.

For example www.lancs.ac.uk is an FQDN. If we perform a type A query on it we will be requesting the v4 address associated with that FQDN. Currently i is mapped to 148.88.1.1 (an v4 address) so the name server will return a response with a resource record attached of type A with that address in.

However DNS can also be used for a variety of other looks up on FQDNs. It can look up v6 addresses (type AAAA), the Mail exchanger (MX, the machine that accepts mail for that domain), the name server responsible for that domain (NS), the canonical name of a host (CNAME), to look up FQDNs from IP addresses (PTR, also known as reverse query) and even to get miscellaneous text information about the host (TXT). This project is mainly concerned with the type A query (v4 lookup) and type AAAA query (v6 address lookup), although it also deals with PTR queries (reverse mappings).

2.5 The GNU/Linux Operating System

2.5.1 Brief description and history of the OS

The Linux operating system is a reimplementation of the Unix operating system. The kernel (which is the only part truly called Linux) was designed initially for for the x86 architecture CPUs, as produced by Intel, AMD and originally IBM. It was created by Linus Torvalds[28, 37] while he was a student at Helsinki university and released under the GNU GPL license[16]. This has enabled it to be distributed, modified and examined by thousands of programmers around the globe. The official version is still maintained by Linus Torvalds and a core group of kernel hackers[34]. Official releases are being distributed from www.kernel.org.

Traditionally the bulk of the userland programs to compliment the Linux kernel are from the GNU project[15] (as well as many individual ones). These form the bulk of the system that users interact with (the C library, compiler, editors, init, the servers etc.) and so many people insist that the Operating System should be referred to as GNU/Linux to properly reflect the input of the GNU project[51].

2.5.2 Impact on this project

This project is based around modifying the Linux kernel by writing several modules for it, one to store data and a pair of modules to modify the operations of the the netfilter[32] packet-filter (which is part of the Linux kernel). This task has been made easier because the source for both these projects are open, and numerous websites, mailing lists and usenet groups cover the attempts of people to modify them.

3 Implementation Background (Technical Unix Information)

This chapter deals with technical background information that is more specific to the environment that this project will be implemented on than the previous background chapter. Chapter 3 deals entirely with the parts of the Unix operating system that are pertinent to the system, covering the root user, the differences between the kernel and the rest of the system, device files, networking with Unix systems and finally Unix daemons.

3.1 Root and other users

All processes which run on a Unix host[34], including interactive shells which users interact with, the GUI and network servers, have several properties. Chief among these is which user account they run under, known as their UID (User IDentification) which identifies which account the process is running under. This determines what they can do in terms of connections to the outside world (as Netfilter can filter based on users), what files they can access, how quotas affect them and who can send them control signals (to kill them, restart them, pause them etc).

The Unix operating system is designed around several simple core ideas, one of which is the division of the user accounts on the system into two categories, root and all other users. The root user is defined as any account with a UID of 0, it is the account used for system administration and the running of certain key processes and its powers are unrestricted on a normal Unix machine (variants do exist in which root's power is limited for security[31]).

The root account can access any file, doing anything to it including deleting it. Processes running as root can bind to the restricted ports (all networking ports below 1024 are considered to be of use only for servers, and thus normal users cannot bind to them[21]), change the CPU priority and quota (both disk and memory) for other processes or even modifying the kernel of the operating system itself. The root account on a Unix system is powerful, therefore any mistakes made by a process running as root (or a user logged into an interactive shell which is running with root permissions) can be very bad for the system, including destroying it to the point where it needs to be reinstalled. Needless to say that the security of the root account is paramount on a Unix machine.

This is important to note for this project because it features one program that runs as root and also three kernel modules which must be loaded into

the kernel of the operating system by the root account. The implications of this are that no normal user can install this system on their computer, they require access to the administrator account. It also means that the implementation of this project will be in a potential position to seriously damage the machines it is installed on.

3.2 Userland, Kernel Space and the design of the Linux kernel

Linux is a classic monolithic kernel[52] and forms the centre of the entire operating system. From the point of view of traditional operating system design all code execution falls into two broad categories. user land and kernel space. Kernel space is those instructions that are executed inside the kernel itself, for example receiving packets from the network or committing data to the disks. All other instructions are part of (what is known in traditional Unix parlance) as userland[40].

The Linux kernel, while still following the traditional monolithic model in most places, is more modular[49] than older implementations of the Unix kernel. It has been designed so that modules of code that perform certain functions (device drivers[26], filesystem drivers, networking filters, etc) may be loaded and unloaded to a running kernel, modifying what it can do without needing to take down the entire system and reboot it. The root account can insmod or rmmod (INSert, ReMove MODule) modules by hand, or they may be loaded as they're needed (which is common for file system modules, to only stay in memory while filesystems of that type are being used). This allows the kernel to stay more organised and compact in size as a lot of its code is modular. However since these modules are loaded completely into the kernel if they have errors in them they can crash the kernel which is known as causing a kernel panic. This generally causes the machine to need to be hard rebooted, although many times systems may be able to reboot themselves.

If things are not part of kernel space then they exist as part of userland[9]. Processes here essentially run inside sandboxes. They are subject to restrictions on their CPU priority, memory usage (they can only access what is allocated to them, the kernel may access all memory) and access to hardware and the filesystem. Userland processes form themselves into a tree-like structure, the root of which (the first process) is known as "init". init is the first process spawned on a system, and is created directly by the kernel, it forks to create all other processes and takes care of orphaned processes (child processes whose parents die before they finish).

It is important to understand this distinction to understand the implementation of this project, as it features a split between components that exist in

kernel space and those that exist in user land, errors in the former being far more urgent than those in the latter.

3.3 Devices and the File System

Unix systems traditionally have only a few simple rules backing the way that programs and the system interacts. Chief among these is that everything is a file[61]. On Unix systems access to the hardware devices is represent by access to files. These files generally live in a directory in the root of the system called '/dev'. These files are linked via deep magic[39] to the device drivers in the kernel of the OS. Doing things to these files results in the kernel doing things to the device itself.

A classic example of this behaviour on a Linux system is "cat foo.au > /dev/audio". This example works in a fairly simple way, cat opens the file foo.au, which is a file containing encoded audio data, and pipes its contents into the device /dev/audio (which on many systems is a symbolic link to audio0, allowing the sysadmin to easily change what the standard audio device is by linking it to another file in /dev), the kernel opens the module that controls that audio device and runs a standard function inserting the data that is offered by the shell. The end result is that the sound file is played by the kernel module that controls the soundcard, and sound being outputted to the speakers.

The implementation for this project features a kernel module (the DNSRH) which is linked to a device file in this manner and so knowledge of how this is done will be helpful to fully understanding how this module works.

3.3.1 Communication between Kernel Space and Userland

There are two major ways for there to be communication between Userland and Kernel space. The first is system calls[58]. System calls are function calls that trigger a function to be run inside the kernel and the response to be returned, appearing to be a standard function in a library. The other method is for a module to be represented as a device in /dev and for it to have a set of functions that are associated with IOCTL numbers[44] (Input Output ConTroL).

IOCTL is the name of the syscall that is used to perform special operations on these device files, things outside of the standard opening/reading/writing etc. Traditionally this has included such things as low level hardware diagnostics and tests. IOCTL calls require only 2 things, a device file to operate on and the number of the function to request from that kernel module, optionally

they can take a pointer to some data in memory as well, which can be a struct of multiple values if needed, which allows you to pass information to the device to control how it performs the operation, or gives it space to fill in the structure with information about the success or failure of its attempted operation.

The most interesting advantage of this model of access is that you can use the filesystems normal permissions model to limit access to the kernel modules to certain users or processes. For example /dev/dnsrh (which is part of the implementation of this project) on my system has the permission bits "crw———-" and is owned by the root user, and is in the root group. The standard permissions model for Unix systems is to break them into three groups of three permissions. These are permission for the owner of that file, the group of that file and all other groups, and they're always rwx (Read, Write, eXecute). Device files also have a bit at the start to declare if they are for block devices (which deal only in chunks of data of a certain size, for example hard disks) or character devices (which deal with streams of bytes for data, such as parallel ports). These permissions mean that only processes running as root may access this character device. And those processes may only read data from it or write data to it. Other processes, even those in the root group, cannot even read data from it.

This security model means that checking for certain permissions, user IDs or secrets/passwords is not done by the library or the module itself, all the complicated authorisation is done by the filesystem giving the sysadmin greater control over the module, and means the module author has less complicated code to write which should make their module more reliable and secure. For example the sysadmin may change the ownership of the device to a less privileged user and quota the amount of CPU time that user may use, or the priority with which that users processes run. Then by running the programs that use that device as that user all processes that use that module can be restricted in terms of CPU, disk and network usage and don't need to be run as the potentially dangerous root user.

Knowledge of the inner workings of the Unix device file system of accessing hardware and kernel modules is not needed to understand the implementation of this project. However understanding the overall concept behind the linking of IOCTL calls and the /dev file system will help.

3.4 Networking and Unix

3.4.1 Sockets

The Linux operating systems uses the BSD sockets style of networking[12, 46], in which an abstraction of a tunnel of traffic with a socket at each end, where data is piped into one socket will appear at the other socket. This has been the standard model of networking programming for many years and is used in many operating systems including the Unixes, Microsoft's Windows and Apple's MacOS.

3.4.2 DNS and Unix

DNS on Unix systems is traditionally controlled through the use of three configuration files (I will be ignoring more modern distributed methods such as NIS for the purposes of this report), all of which are stored in /etc, which is the main place for config files for the whole system.

host.conf This stores the overall config and decides in which order the next two files are consulted. Its gives an order for the use of two possible variables, "hosts" or "bind". "hosts" refers to /etc/hosts and "bind" refers to the use of /etc/resolv.conf

hosts Hosts is simply a flat text file that lists a number of IP addresses and the FQDNs that these are linked to. It can be used to list hosts in a small network without DNS, as a backup in case DNS fails (although for modern networks this is not advised) or to store local aliases (such as linking 148.88.8.1 to "cent1") to reduce lookup times. The use of hosts predates the existence of DNS and was originally the only way to match a symbolic name to an IP address.

resolv.conf This file lists the nameservers that this machine should query, starting with its primary NS and moving onto any backup name servers that exist. It also has an option called "search" in it, this is used to append a domain to hosts without one. For example if it was set to "localnet" and a user attempted to look up the hostname "otto" it would search for "otto.localnet".

Of the several name servers that are available for Unix systems this project will be making use of BIND[23] (Berkeley Internet Name Daemon) for its testing. This is one of the original name servers that were created and has versions for all Unixes (including MacOSX) as well as Microsoft Windows systems. BIND is a large and complex program that runs as "named" on Unix machines (the Name Daemon).

3.4.3 Netfilter overview

Linux has a stack that is built into the kernel. It supports v4, v6, Decnet and a host of other protocols. Packets themselves are represented as a large and

complex struct called an sk_buff[63] (see /usr/src/linux/include/linux/skbuff.h). The packets themselves are filtered, routed and modified by a section of the kernel known as Netfilter[32]. Netfilter centres around the concept of hooks, which are critical points in the path of a packet (such as its entry to the system, or just before its passed to a user process that is waiting for it), and as such are the best places to place filters to modify traffic on the machine. Understanding the basic concepts of Netfilter is crucial to understanding the implementation of two of the three modules of code developed for this project.

3.4.4 Netfilter modules and nf_register_hook()

The Netfilter architecture[64] allows a simple way for modules to affect the flow of packets. The module constructs a static struct of type nf_hook_ops, this lists the function that the packets will pass through, the family of packets and the point of the path (known as the hook) you're module is interested in. When a module is loaded it can call nf_register_hook[47] and pass that function its nf_hook_ops struct, from then on its function is inserted into the stack at that hook point and has a chance to modify all packets which pass through that point of the stack. Essentially nf_hook_ops details what kind of traffic and at what point in its path you want to see, whereas nf_register_hook "switches on" the filter by registering it with the stack.

3.4.5 Netfilter module return codes

The function that modifies the packet must return one of 5 returns which are stored in enumerated types. NF_ACCEPT, NF_DROP, NF_STOLEN, NF_QUEUE and NF_REPEAT[62].

NF_ACCEPT Allows the packet to continue on its path through the stack with whatever modifications the function did to it.

NF_DROP Means the stack will carefully prune the packet from the lists its currently on and free its memory.

NF_STOLEN Returning NF_STOLEN means that the packet is removed from its lists and totally forgotten about by the packet filter. Netfilter will not clear up behind the function, nor will it do any checking, it will simply carry on to the next packet. From here you can do anything with the packet.

NF_QUEUE Sends the packet out to a userspace process that can then modify the packet and return it back into the stack or drop it. This allows for more functionality to be implemented by userland processes, which in event of a failure are less dangerous as they cannot crash the entire system, just lose packets.

NF_REPEAT Sends the packet around through the function again.

3.4.6 Netfilter hooks

There are 5 main hooks in Netfilter NF_IP_PRE_ROUTING, NF_IP_LOCAL_IN, NF_IP_FORWARD, NF_IP_POST_ROUTING and NF_IP_LOCAL_OUT[45]. The first three affect incoming packets, the last two affect outgoing packets. How these feed into each other is displayed in figure 3.1

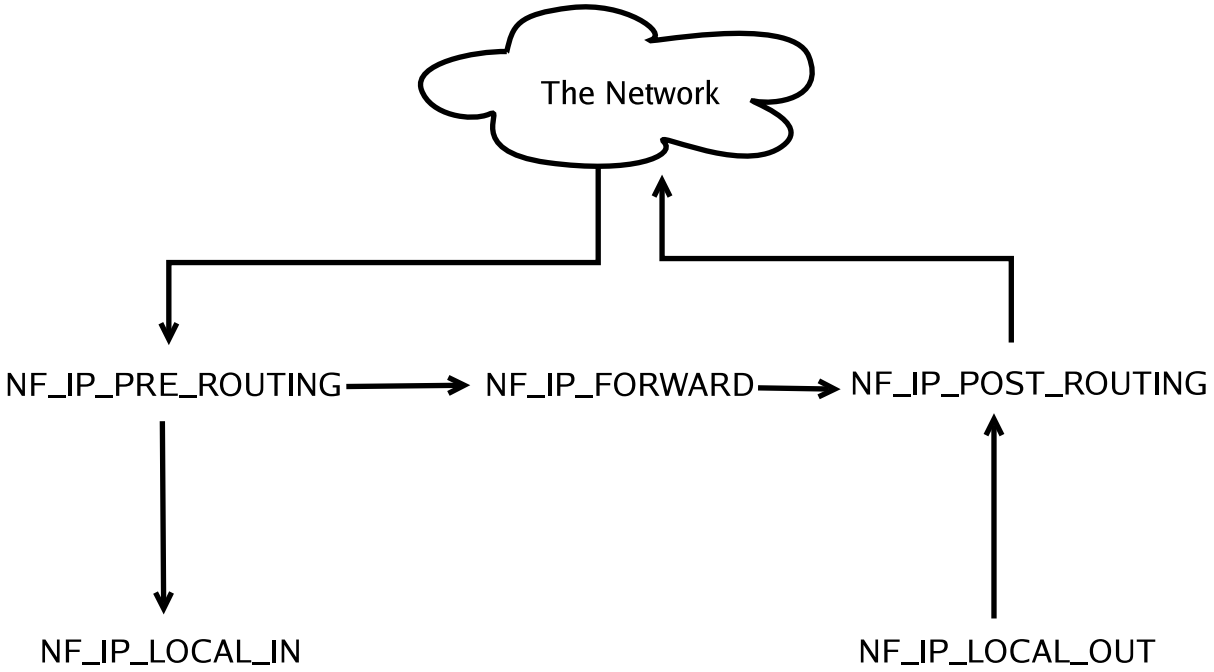

Figure 3.1: Structure of netfilter hooks[47].

Packets arrive on NF_IP_PRE_ROUTING and from there are sent to either NF_IP_LOCAL_IN or NF_IP_FORWARD. Packets on NF_IP_LOCAL_IN progress through all the rules on that table and once they reach the end they are delivered to the user process they should be sent to. Packets on NF_IP_FORWARD are mangled and forwarded off other network interfaces by being sent to NF_IP_POST_ROUTING, this hook is used to create NAT routers for example.

NF_IP_LOCAL_OUT is where locally created packets are first placed. At this point any filters that affect their overall makeup should occur. After they've passed through this hook they're passed onto NF_IP_POST_ROUTING, which does any last minute changes not based on routing them and finally places them on the wire.

3.4.7 Netfilter Protocol Families

Netfilter also knows about several families of protocols, these include PF_INET and PF_INET6 for the v4 stack and v6 stack respectively. For the v6 stack

the hooks discussed above are named in a similar way with '6' inserted after the IP. So there is NF_IP6_POST_ROUTING and NF_IP6_PRE_ROUTING for example.

3.5 Daemons and Unix

Traditionally under Unix systems there is a class of persistently running processes, the most common of which seem to be networking servers, that are referred to as daemons[38]. These processes wait in the background for a specific event (normally a connection attempt from a client) and follow the traditional client-server model for communicating. Generally they provide some service or abstraction for the user.

There is a fairly standard model for programming network server daemons. First the program binds to its port and listens for input. Then every time it gets a request on that port it forks a child process. The child process does all the data processing and even returns the response, meanwhile the parent server returns to listening for another request.

Fork used to be the best method for doing this, however whenever a process forks (using the fork() system call) it creates two nearly identical copies of the process at the same execution point. The only difference between them is the return value of the fork() call. The idea being that the program examines this return value at this point and the parent and child choose different execution paths.

In modern Unixes there is a library for threading called the pthread library (for POSIX Threads)[50] which enables you to create lightweight threads, not quite a processes in their own right, more a function that is run in the same memory area as the main processes but with its own flow of control. These reduce the weight of having to spawn an entirely new process each time a new connection comes in, as they are generally implemented as true kernel level threads instead of user level threads. This means the kernel doesn't just spawn another processes to represent the thread (which would still have expensive context switching) but instead has knowledge of creating truly light threads.

4 Design

The design chapter contains details of an non-implementation specific architecture for a BIS. This means that the design that is discussed in this chapter could be implemented pretty much for any operating system using several methods for each section. The next chapter (Implementation) covers the system that was actually built as part of this project to the design laid out in this chapter.

4.1 Overview

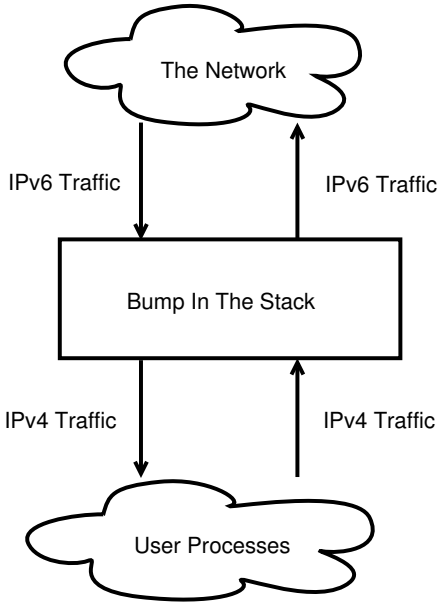

Figure 4.1: A simple overview of the BIS architecture

The above figure (4.1) gives a very simple overview of the idea behind a bump in the stack. That there is a transparent filter between all user processes and the network at large. This outgoing filter (bump in the stack) translates all their outgoing packets from protocol A to protocol B, and incoming traffic is then translated from B to A.

This project is designed around the normal design of a BIS but has several constraints over it. These have been decided based on the environment that its being designed for which is a Linux specific environment. These constraints will be reflected by the design choices made later. Its also being written as a method of helping the transition towards v6, and for this reason translates from v4 to v6, with the inside of the machine appearing to be v4 and the outside v6. A more specific, and less black-boxed, design for this implementation of BIS is shown in figure 4.2.

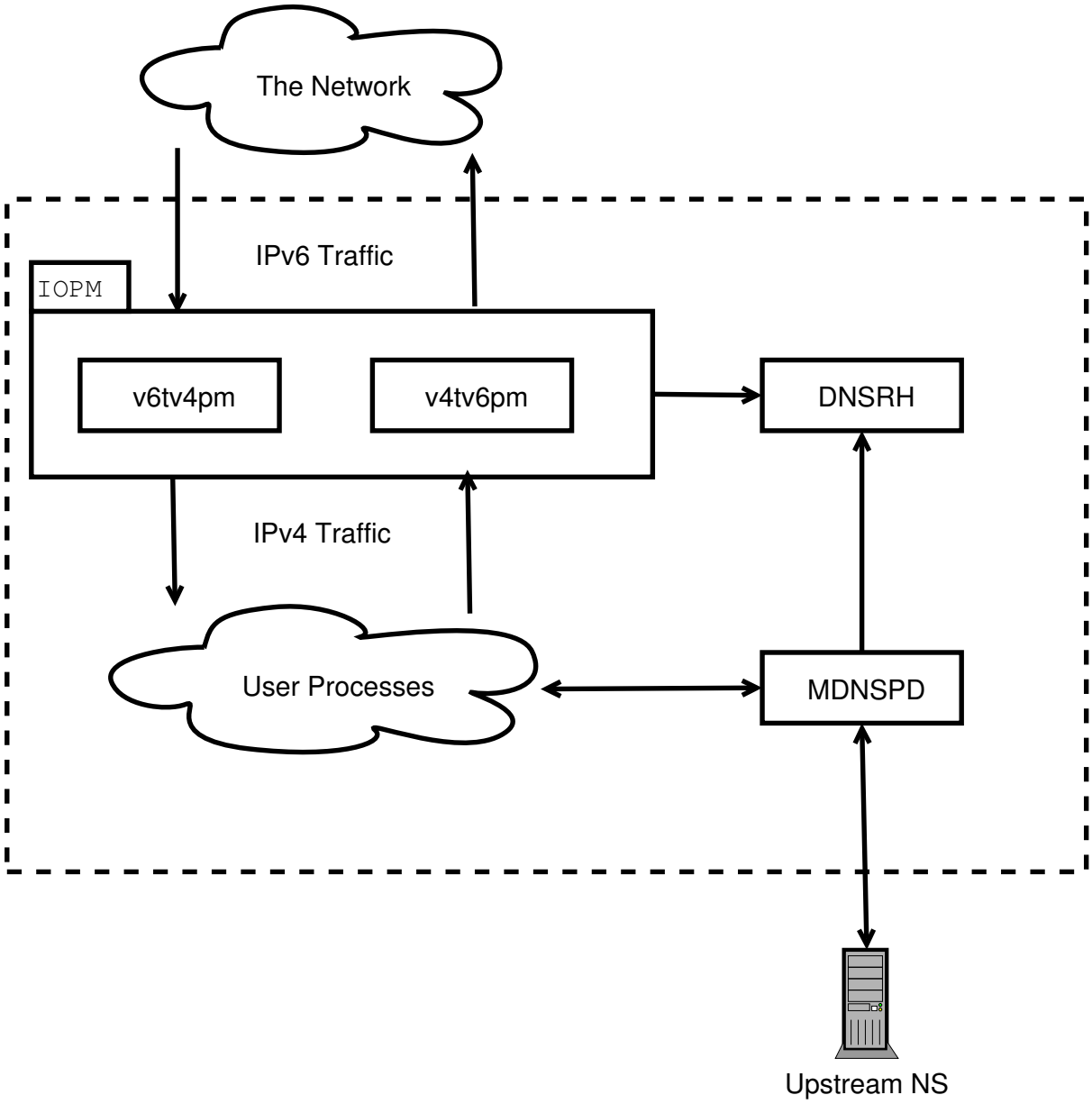

Figure 4.2: Non-Implementation specific detailed architecture diagram.

Thus this project can be loosely broken into three large chunks. Firstly the DNSRH (DNS Results Hash) which is responsible for storing the mappings between v4 and v6, secondly the MDNSPD (Mangling DNS proxy daemon) which deals with resolving DNS requests and returning fake addresses for v6 hosts. Finally there is the IOPM (Input and Output Packet Manglers) which actually translate the incoming packets from v6 to v4, and the outgoing packets from v4 to v6, translating the addresses based on the information in the DNSRH.

The diagram above (figure 4.2) shows the interactions between the elements, and between other parts of the system. As you can see the MDNSPD is the most visible element of the system to the outside world. It forms the interface by which all user processes communicate with the DNSRH and is the only way new mappings can be added. It looks up DNS queries via an off-machine name server, and stores its results in the DNSRH for future use, this enables v6 machines to be consistently addressed by a version 4 address.

The DNSRH never initiates communication with the other elements, instead it simply sits in the kernel waiting to be queried by the MDNSPD and the IOPMs. The IOPMs intercept all incoming packets just as they arrive and all outgoing packets just before they are sent, query the DNSRH for the mappings for those addresses and rewrite the packets appropriately. This means that the translation will seem to not occur as its the first and last steps for all packets.

A full example of how the different parts of his system fit together is in section 4.5.

4.2 The DNS Results Hash

The DNSRH is the largest module in terms of system resources consumed. The job of the DNSRH is to store the mappings between the external v6 addresses of real machines and the fake internal v4 addresses which applications on the machine will address packets to. This means that for every v6 host that the BIS machine has queried there will exist a mapping between its v6 address and a v4 address (and one back the other way) in the DNSRH. Its tasks can be broken down into two sections, storing the data and communicating with userland.

4.2.1 Storing the data

The most important factor in the management of the data for the DNSRH is speed. Retrieval speed of the data, and the speed with which a v4 address can be mapped to a v6 (or vis-versa) is the most important part. The speed of storing a v6 to v4 mapping is less important.

The reason that mapping speed is the most important factor of this module is that poor implementation at this stage will result in a loss of performance. Every single packet that enters or leaves the machine needs to have a lookup performed in the DNSRH and then needs to be rewritten. Obviously if the lookups are slow then network access will be slow, the faster the lookups the less impact a lookup in the hash will have and the less impact on the machine's

performance will be noticed. Since a major goal of this transitioning method is transparency then it must be as fast as possible to avoid breaking this by slowing down communication with the network.

In contrast the adding of an element to the Hash is done only when DNS lookups are required, which is a relatively rare process compared with packets needing filtering. Therefore this function can be relatively slower than the mapping speeds, taking time to create an efficient hashing method to speed up the mappings.

For reference a design decision about communication with userland was made in that a standard struct will be used which contains both a v4 and a v6 address worth of data. This means that all functions calls (adding a v6, looking up a v4 or a v6) require the use of only one type of struct as the answer will be written over the appropriate space in it for the functions that look up data.

4.2.2 Communicating with other modules

The DNSRH is a section of code that exists only to store data. The way that networking works insists that client programs (such as the IOPM or the MDNSPD) query it for data and it wait to service clients between queries.

The API to access the DNSRH has been designed to be fairly simple and only three calls will be implemented and externally accessible to the other parts of this project.

addV6Addr() accepts an v6 address and assuming one doesn't already exist generates an v4 address to map to it. It then stores this mapping and returns the v4 address.

checkForV4() accepts an v6 address and returns the v4 address that it maps to, or returns all 0s to say there wasn't one.

checkForV6() accepts an v4 address and returns the v6 address that maps to it.

The DNSRH will be implemented such that these three functions are available to the MDNSPD, and that at least the checkForV4() and checkForV6() functions are available for use by the IOPMs.

4.3 The mangling DNS proxy daemon

The MDNSPD is designed to sit between the user processes and a normal name server (known as the Upstream Name Server). The program is designed to be running on the local machine, and configured to be the machines only

name server. Therefore all DNS queries will be sent to it, and answers will appear to originate from it, as opposed to the upstream name server. This proxy daemon must be as transparent as possible to the end user, to the point of view of programs it is the upstream NS.

Since it is transparently intercepting all DNS responses it will check to see if any of them include v6 addresses as one of their resource records. If they do the MDNSPD will use the DNSRH to generate a new v4 addresses and rewrite the DNS response correctly before returning it.

The MDNSPD is also designed to deal with the issue that all internal processes think the machine is a v4 host, whereas the network its residing on will be a v6 network, and thus largely populated by v6 hosts. This means that the processes running on localhost will largely be sending out type A queries (which are queries for the v4 IP address of a host).

The MDNSPD must therefore test to see if these queries return responses and if they don't try the same questions with an AAAA (v6) query, translating any response records received from this from v6 addresses to v4 addresses with the aid of the DNSRH. This way the querying processes will only have to deal with type A queries and type A resource records, yet the nameserver which only has mappings of type AAAA for those hosts will return their addresses.

4.4 The Input/Output Packet Manglers

This project revolves around a pair of IOPMs (Input/Output Packet Manglers). One that mangles all outgoing packets after all other filters, and one that mangles all incoming packets before all other hooks. The outgoing filter is the v4tv6pm (Version 4 To Version 6 Packet Mangler) and the incoming filter is the v6tv4pm (Version 6 to Version 4 Packet Mangler). Both of these modules rely on the existence of the DNSRH (which in turn relies on the existence of the MDNSPD to populate it). Without the DNSRH being available they won't function correctly and may cause interesting networking issues depending on their implementation.

Once the IOPM is loaded the stack will resemble the block diagram shown in figure 4.3.

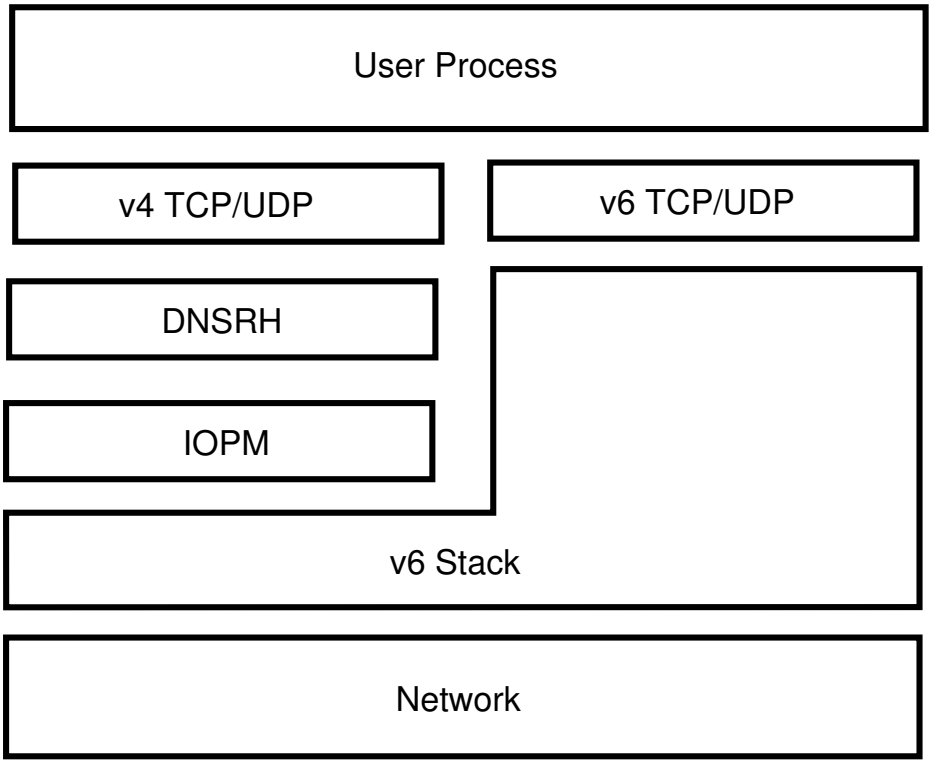

Figure 4.3: IP Stack diagram.

4.4.1 Identifying packets of the correct protocol

The first task of each filter is to correctly identify the packets its interested in. The v4tv6pm should catch all outgoing packets, looking for v4 packets and only v4 packets. The v6tv4pm should catch all incoming packets before other filters and look only for incoming v6 packets.

4.4.2 Checking for mappings

The packet manglers at this point need to check for mappings. The v4tv6pm uses the checkForV6() function of DNSRH, passing it the v4 address of the outgoing packet. The v6tv4pm module uses checkForV4(), looking for a corresponding mapping to its v6 address in the hash.

4.4.3 Rewriting

The IOPMs at this point must rewrite the packets. They must change all the protocol information so that the packet changes protocol. This includes changing all elements of the structures that relate to their protocol and also rewriting their headers. Both packet manglers must create a new header struct of the correct type (either v4 or v6), translate the old header into it, remove the older header and replace it with the new one. They also have to

ensure that the packets checksums are correct and all internal lengths relating to the headers are correctly adjusted. The addresses must be written in based on the lookup from the hash earlier.

4.4.4 Re-injecting

Now the packets are fully changed over to their new protocols, and the addresses are rewritten they must be replaced on the stack to proceed onto the wire as normal. The original packets must be removed to prevent double the transmissions occurring.

4.5 BIS Example

An example of how all these sections fit together can be shown by demonstrating how a user using a web browser would be able to transparently communicate with a v6 only web page to view a page of content.

1. A web browser looks up the FQDN www6.v6.localnet Because the web-browser is a v4 application it sends a type A query (v4 lookup).

2. The MDNSPD is the only nameserver for the machine, it sends the type A query to its upstream nameserver.

3. The upstream server returns a response. Unfortunately www6.v6.localnet doesn't have a v4 address so the upstream server returns a DNS query with no resource records attached.

4. The MDNSPD rewrites the DNS query from a type A (v4 address) to a type AAAA (v6 address) and resends it to the upstream server.

5. The upstream server sends back a response with a type AAAA resource record attached with the v6 address of www6.v6.localnet which is fec0:1:2:3::f:5

6. The MDNSPD sends the v6 address to the DNSRH.

7. Since the DNSRH hasn't seen the address fec0:1:2:3::f:5 before it creates a new v4 address to map it to (240.0.0.6). The DNSRH then adds this pair to the hash tables and returns 240.0.0.6 to the MDNSPD.

8. The MDNSPD rewrites the response from the DNS server so that its of type A, and now includes the address 240.0.0.6

9. The users web browser gets the answer 240.0.0.6 to its query of the IP address. It now starts a TCP connection to that address.

10. The outgoing packets are identified as v4 (this in fact happened for the DNS traffic earlier but it wasn't mentioned for clarity). The outgoing filter checks the DNSRH for 240.0.0.6 and finds fec0:1:2:3::f:5 which it then writes into the packet, changing the packet to make it a v6 packet.

11. The packet is transmitted, and has the v6 address of the machine running the BIS as its source address. It arrives at the machine www6.v6.localnet

12. The incoming reply from www6.v6.localnet arrives at the machine. The IOPM sees that it has a source address of fec0:1:2:3::f:5 and looks this address up in the DNSRH. The filter rewrites the packet to be v4 and changes the source address to 240.0.0.6 based on the response from the hash.

13. The web browser receives a response and a TCP connection is established.

14. All further packets are translated in this way, enabling a v4 only web browser to view web pages from the server at www6.v6.localnet even though that machine only communicates via v6.

The above example fits the non-implementation specific design that has been discussed in this chapter. The next chapter deals with the specifics of the implementation that was created for this project.

5 Implementation

This chapter details the details of the implementation of this project, as such it contains many technical details which are very specific to the C language, the Unix environment and IP networking. All of the technical information needed to understand this should have been contained in all previous chapters.

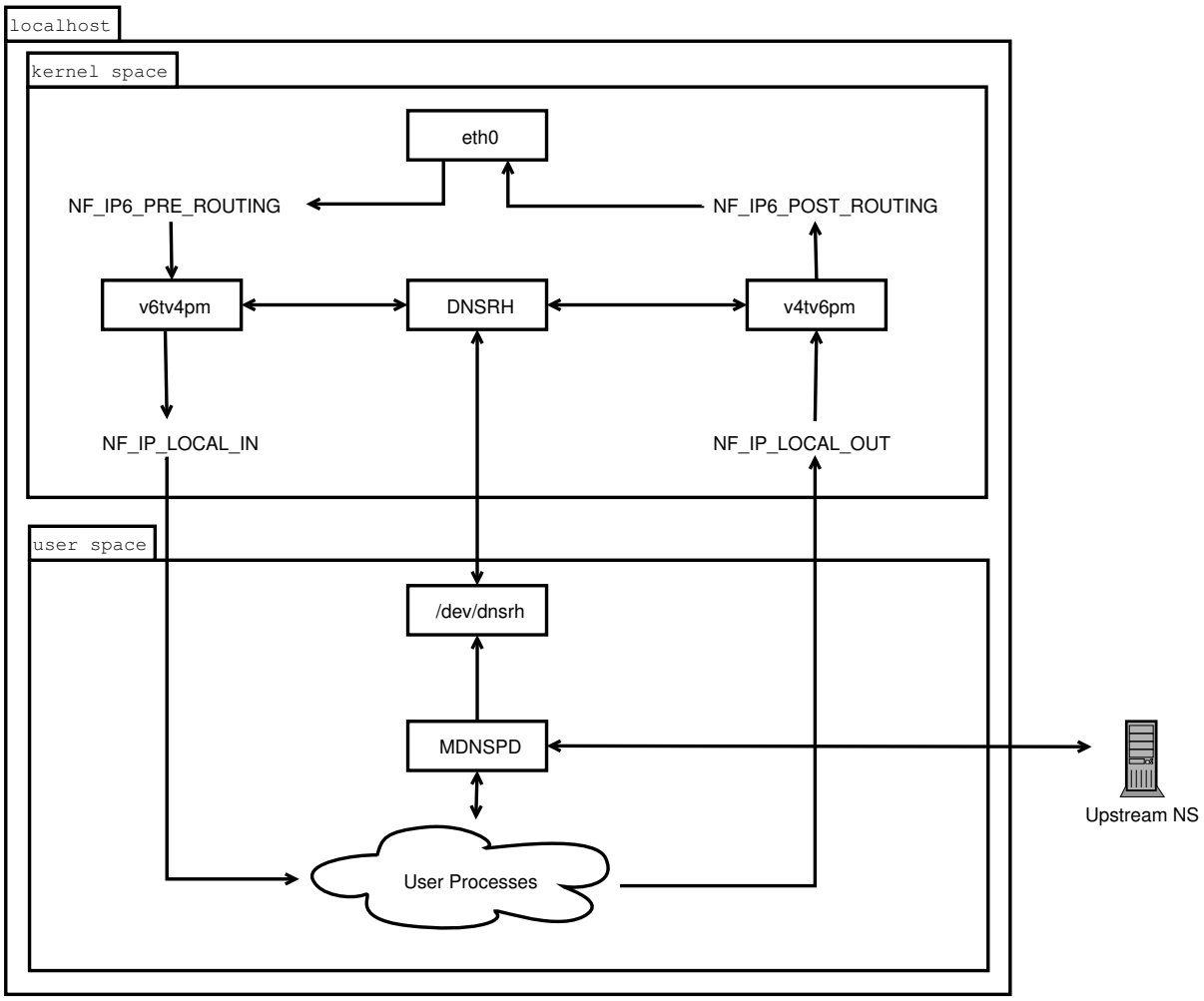

Figure 5.1: Implementation specific architecture overview.

The diagrams used in the design chapter were implementation agnostic, in that they were created to show how the system was designed and no feature any part that is only about how this version was implemented. This diagram (figure 5.1) shows how this implementation of the design was actually created. It shows the division between the three kernel modules (v4tv6pm, v6tv4pm, which together form the IOPM and the DNSRH) as well as the flow of packets through the Netfilter hooks. It also shows how the MDNSPD communicates with the DNSRH (via a IOCTL calls to a character device) and the upstream

name server. Figure 5.1 serves as the blueprint that will be discussed in the rest of this chapter.

5.1 The DNS Results Hash

The DNSRH has been implemented as it was designed in section 4.2. However the design above doesn't include the actual technical details of the final version of the project.

Because the DNSRH is being implemented as module for the Linux kernel then the language that must be used is straight ANSI C without any standard libraries, as the kernel is not linked against any library at all (in order to reduce dependencies and size). All the functions that were stated as forming the API in section 4.2.2 will be prefixed with dnsrh_ so that any symbols exported by the modules will not pollute the Linux kernel's name space with ambiguously named functions.

5.1.1 Storing the data

In the implementation of the DNSRH the mapping information is in fact stored in a pair of Hashes. This model was chosen because the actual Hashes themselves are simply a pair of arrays of pointers and thus are cheap in terms of resources (the exact figure is compiler dependent by assuming a pair of 500 element arrays with 4 byte pointers means 4008 bytes total). The part that is expensive in terms of computational resources is the storage of the data itself. Each mapping is represented as a an ip_pair struct (each of which is 20 bytes and a pointer in length, this enables it to store a single v4 and a single v6 address). Since the hashes are limited at 16,000,000 elements (the entire 240.0.0.0/4) then this means the kernel hash can theoretically grow to be 312,500 bytes (305Mb) at its maximum size. Assuming that the hash is ever filled (all v4 addresses are used) then it will start to overwrite entries on an oldest first basis on the theory that the older the mapping the less likely it is to be currently in use.

The functions that generate hash keys (hashV4Addr and hashV6Addr) themselves are designed around the principle of converting the addresses given to arbitrarily large numbers and using the modulus operator on them to ensure they are within the boundaries of the hash. Creating an algorithm which could turn a v6 or a v4 into the same key would render the use of two hashes pointless, however such an algorithm would be a large and complex thing to create. Because of this both functions work simply by multiplying all the sections of the addresses together, which is quite likely to wrap around the machines representation of an integer number, going from its maximum to minimum value.

The current method of hashing is a rather primitive method created for this project. It is however both fast and robust. Fast because although multiplication is slower in computational terms than addition it should produce a large enough spread of data to help ensure the hash is more efficient and that data is spread correctly through it. Its robust in that its easy to understand, thus you could work out by hand what each hash could would be for an address to debug the module if you so needed.

Because the algorithms used cannot guarantee that the hash keys generated will be unique to the data stored in the hash (and the hash itself certainly doesn't have enough elements to ensure this will happen) the data structures used (ip_pair structs) are designed with a pointer in them. This enables "chains" of data to be constructed after each hash key. While this reduces the efficiency of the structure (as the algorithm may have to look up the key, then follow to the end of a chain with no length limit), this again has been written in order to have a faster and more robust set of algorithms for dealing with the hash rather than a more elegant system.

The DNSRH exists as a kernel module, this means that the data stored in the hash can never be moved out of memory and into swap. If the system running the DNSRH is under heavy load, and has had a lot of DNS queries performed then it has less memory to deal with other processes. However this means that the system running the DNSRH and MDNSPD as a combination facing the Internet have a new and very dangerous DoS (Denial Of Service) attack open to them.

5.1.2 Communicating with userland

The DNSRH uses three IOCTL calls for its communication with userland processes. These are bound to the following functions: dnsrh_addV6Addr(), dnsrh_checkForV4() and dnsrh_checkForV6(). All three IOCTL calls expect pointers to ip_pair structs which contain the data needed for them to operate.

addV6Addr takes an ip_pair struct and uses the v6 address in that struct, checking for its presence in the v6 hash. If it finds it it writes the mapped v4 address into the users struct and returns from the function. If it doesn't it generates an v4 address, inserts this new pair in the v6 and v4 hashes, fills in the users struct and returns to indicate it added fresh data to the hash. If it finds a matching v4 address it overwrites that entry in the hash for spacing saving reasons.

checkForV4 takes an ip_pair struct and extracts the v6 address from it. It checks the v6 hash for a matching v4 address and writes it (if found) into the users struct. It returns an integer based on if a match was found, assuming it wasn't it will overwrite 0s into the v4 section of the struct it was given.

checkForV6 essentially performs the opposite of the checkForV4() function, except it extracts the v4 address and searches for a v6 match. Both these functions operate by linear scans of the hash tables and thus the amount of time they take to run depends upon if

All the functions that deal with the data structures (addV6Addr, checkForV4, checkForV6) have their performance directly affected by how full the hash is. The data in the hash is indexed by key, but the key generation is designed to be random yet consistent. This should lead to a good distribution of data across the hash for the most efficient performance. However the algorithm used (section 5.1.1) is probably not that good at its job, which will lead to bunching of data in clumps. To get around this problem the data is designed to be chained so that in event of more than one address generating the same key they do not overwrite each other.

5.1.3 Support Functions and Internals

The internals of the DNSRH are implemented in the simplest fashion possible on the basis that simple code is correct code. There are four support functions (compareV4(), compareV6, hashV4Addr(), hashV6Addr()) all of which are used to deal with the pair of hashs. There is also one function for for dealing with the creation of new v4 addresses (generateV4()).

compareV4() and compareV6() both take two ip_pair structs. They do simple linear comparisons of the relevant address type in that struct returning as soon as either a match or discrepancy is found.

hashV4Addr() and hashV6Addr() take an ip_pair struct each. They multiply each byte of the addresses together and return a modulus of the result as the hash key. This is not the most efficient algorithm that could have been used, see above for discussion on this.

generateV4() returns a struct of a single v4 address, it generates new v4 addresses to map to V6 addresses in a linear fashion. Each time the module is loaded it starts at the start of the Class E block (240.0.0.0) and counts from there incrementing one each time until the address space is exhausted (240.255.255.254) at which point it resets to 240.0.0.0 and restarts. The addV6Addr() function has been built to deal with this behaviour. It was written in to ensure that the hash can grow to large, but not extreme, levels (maximum 16 million elements) which will hopefully help to protect the running system from external malicious agents, software bugs and network faults.

The DNSRH is implemented to link to the device file /dev/dnsrh. Any processes with the correct permissions (read and write) against this file can use the IOCTL calls to communicate with the DNSRH, passing it ip_pair structs in the calls to collect the data they want. Its three IOCTL calls are listed as symbolic constants for easy changing in dnsrh.h (which all files wanting to use this device should include). These are IOCTL_ADD_V6 IOCTL_CHECK_FOR_V4 IOCTL_CHECK_FOR_V6 which correspond to the functions they are named after.

These constants are #define'd as macros from ioctl.h (/usr/include/linux/ioctl.h). IOCTL_ADD_V6 is defined using _IOR and the other two are defined using _IORW. These macros change how data can flow, in an IOCTL call using _IOR data cannot be written to the device, only read from it, where as in the _IORW calls data may be read from the device and written to the device.

5.1.4 An example of how the DNSRH internals work

The following example will deal with both hash tables and illustrate how the hashing functions work. It will deal with hashing the v6 address fec0:1:2:3::f:5. The v6 hash is essentially just a 500 element long array of pointers. As such the element number will be shown by a number in square brackets. After this the two character arrow "->" will be used to show the contents of that pointer. Empty elements, or the end of chains will be indicated by the value NULL. Any element marked "ip_pair" can be assumed to be a v4 and v6 address pair that is unimportant to the example.

The v6 hashs elements we're interested in look like this:
[199] -> NULL
[200] -> ip_pair ->ip_pair -> NULL
[201] -> NULL
[202] -> ip_pair -> NULL
[203] -> ip_pair -> NULL
[204] -> NULL

The v4 hashs elements we're interested in look like this:
[87] -> ip_pair -> NULL
[88] -> NULL
[89] -> NULL
[90] -> ip_pair -> ip_pair -> NULL

1. The MDNSPD calls addV6Addr() and gives the memory address of an ip_pair struct as an argument. Its .v6 element contains the address fec0:1:2:3::f:5 (in decimal network byte order form). The .v4 element is set entirely to 0.

2. addV6Addr() takes the v6 address and passes the ip_pair to the function hashV6Addr(). This returns the int 202.

3. The address 202 is looked up in the v6 hash and the chain of pointers is followed until NULL is found.

4. A v4 address is created by generateV4(). In this case 240.0.0.6

5. The new ip_pair struct (of fec0:1:2:3::f:5 and 240.0.06) is inserted at the end of the 202 chain.

6. The v4 address is then hashed by the function hashV4Addr() this function returns 88.

7. Since this cell in the v4 hash is empty the ip_pair struct is inserted as the first element here.

8. The v4 address is written into the ip_pair that the MDNSPD passed to the function and the function returns 0 (which indicates a new element was added to the hash).

After this example the v6 hash table would look like this:
[199] -> NULL
[200] -> ip_pair ->ip_pair -> NULL
[201] -> NULL
[202] -> ip_pair -> ip_pair (fec0:1:2:3::f:5/240.0.0.6) -> NULL
[203] -> ip_pair -> NULL
[204] -> NULL

And the v4 hash elements we were watching look like this:
[87] -> ip_pair -> NULL
[88] -> ip_pair (fec0:1:2:3::f:5/240.0.0.6) -> NULL
[89] -> NULL
[90] -> ip_pair -> ip_pair -> NULL

5.2 The mangling DNS proxy daemon

The MDNSPD has been implemented in C, using the standard GNU C libraries. It is designed to be run as root because it binds to a reserved port (TCP and UDP port 53 is the standard port for DNS). This combination of privileged user and use of the C language means that care must be taken when dealing with any data which comes in from the outside of the program in order to help eliminate the possibility of common buffer overflow vulnerabilities, which since the program is running as root could allow a so-called "remote root" exploit, whereby commands could be executed on the machine running this BIS as the root user.

The MDNSPD is implemented in a similar way to the traditional daemon model. It binds to its port (53 by default) and continuously listens for incoming requests, in the form of UDP packets. Once it receives one it creates a connectData struct, which contains the DNS request, its length, and details of the client. Then it spawns a thread (but does not fork) using its connection() function. The thread deals with all the data processes and is even given a copy of the server socket to reply with.

The connection function forwards the request (which is normally a standard v4 request, a type A) to the upstream name server, if it gets a response which includes no resource records (answers), and is a type A query, it upgrades the request to a type AAAA (an v6 query) and forwards it again.

If this AAAA query is answered it converts the query type back to a standard type A, and all the AAAA resource records into type A resource records. This way the process that requested the type A query gets the correct response.

During the conversation it needs to deal with any v6 addresses it finds. These are mapped to consistent v4 addresses by passing the v6 address into the addV6Addr() function of the DNSRH, using the IOCTL call number stored in the symbolic constant IOCTL_ADD_V6. Once this is done the response is returned and the thread terminates.

In this implementation of the MDNSPD it is highly dependent on the DNSRH. It requires several functions from it to operate correctly (which are outlined above), it also requires the header file (dnsrh.h) for the enumerated types of all the IOCTL calls, as well as the name of the device to be used. This should be taken into considering when using the system, the MDNSPD will not function correctly or even compile without the DNSRH being available and running.

5.3 The Input/Output Packet Manglers

With reference to the earlier Netfilter section (3.4.3), figure 3.1 and figure 5.1 I will now explain where in the stack the IOPM modules are inserted. The v4tv6pm attaches itself to the PF_INET family of filters and the NF_IP_LOCAL_OUT hook chain. This is where packets join when they are created, from there they move to NF_IP6_POST_ROUTING. This is the ideal place for the filter to pluck the packets off the wire as they won't have been processed by other filters, they can be replaced on NF_IP6_POST_ROUTING and from there hit the wire.

The v6tv4pm attaches itself to the NF_IP6_PRE_ROUTING hook on the PF_INET6 chain. This should mean that its attached to the hooks of the v6 stack and get the chance of modifying all packets that have already been routed on NF_IP6_LOCAL_OUT.

5.3.1 Identifying packets of the correct protocol

Netfilter modules get the chance to run their function against any packets that pass their hook while they are loaded. The v4tv6pm and v6tv4pm identify the correct packets to examine based on the protocol field of each sk_buff struct. There are constants defined for v4 and v6 packets (ETH_P_IP and ETH_P_IPV6 respectively). These are passed through the __constant_htons() macro in order to ensure their byte order[57] is set correctly.

5.3.2 Checking for mappings

It is assumed that the DNSRH module is loaded before this one is, as the IOPM modules rely on functions exported by it at this point. Specifically v4tv6pm relies on dnsrh_checkForV4() and v6tv6pm relies on dnsrh_checkForV6(). Both of them need to look up the relevant address mappings in order to create the illusion of consistent addressing in the packets they mangle. v4tv6pm needs to look up the v6 addresses and v6tv4pm needs to look up the v4 addresses.

5.3.3 Rewriting

If a mapping is found the packet must be modified. Before this happens the kernel modules unshare the sk_buff in case it is being dealt with by multiple modules. It will then create a new header of the appropriate type (v4tv6pm will create an ipv6hdr struct, v6tv4pm will create an iphdr struct). Then the modules copy the old sk_buff to a new one.

At this point the two filters differ in how they mangle their packets. The v4tv6pm changes the skb_buff's protocol field to read ETH_P_IPV6, then proceeds to translate the v4 header into the new v6 header, mostly it fills in the header with 0s, to signify no options, for example the flow_lbl (Flow Label) field is always set to 0 as there is no standard for getting this information from a v4 packet, however converted data includes:

payload length the payload_len field can be calculated by taking the total length of the v4 packet and subtracting the header length (tot_len - ihl). This is because v6 headers list only the length of data (and optional headers) not the full length of the packet as v6 headers are a standard length and so don't need to be included.

hop limit the hop_limit field is copied straight from the ttl field of the v4 header. These two headers serve the same purpose, but the v6 protocol changes the name to represent its more common use.

After this the module runs v4tv6pm_fixaddr(), handing it the local source v6 address, the new destination address that was looked up from the DNSRH, and the v6 header. This function writes the two addresses into the source (saddr) and destination (daddr) structs that are in the header.

Finally this the new sk_buff has its nh union moved so its no longer pointing at the v4 header and is instead pointing at the new v6 header and the check sum (csum) field is recalculated.

The v6tv4pm similarly mostly creates a header without any of the complex options set.

total length It calculates the total length (tot_len) by adding the v6 payload length (payload_len) to the size of a v6 header, which is fixed at 40 bytes.

time to live The ttl is calculated by simply copying the value from the v6 hop limit (hop_limit) field.

id and fragment offset These are both set to 0 because of the situation with v6 fragments[54]. All hosts must support a minimum MTU (Maximum Transmission Unit) size, 576 bytes. If a host sends more than this and a router cannot deal with it then it will return an ICMP error message. So based on this a host should not receive fragmented packets.

After this it copies the new source address (obtained from DNSRH) into the v4 headers saddr field. For the destination it checks to see if the incoming v6 address matches the local machines v6 address. If it does it copies it in, otherwise it attempts to lookup the v6 address in the DNSRH, if it finds a match it writes that in, otherwise it assumes that the packet is the localhost and writes the local address into the daddr field.

5.3.4 Re-injecting

At this point the IOPMs should re-inject the packet and return NF_DROP. This return call will cause the stack to clean up after the packet and remove it from any lists for the hooks its on, thus making sure its never sent. At which point the re-injected packet will continue on its way and leave the stack, whereas the original will never be sent. This way the original packet will be gone, the other protocol will be sent and the reliable netfilter code will clean up the original packet from kernel memory.

However this implementation of the BIS is incomplete as neither of the mangling packet filters had their re-injection methods completed. Theoretically use of functions such as ip6_output or ip6_xmit (which are used to send v6 packets) or use of ip6_rcv (for receiving v6 packets) and ip_output, ip_xmit and ip_rcv on the v4 side should have enabled me to re-inject the packets but all attempts met with failure. I even attempted to use NF_HOOK which is a macro for raw placing of sk_buff structs onto a queue but failed to get that working either. However despite this as the Evaluation chapter shows the project is still testable. Section 7.5 also discusses the successes of this project and the uses for the code that was written.

6 Testing and Evaluation

This chapter will deal with evaluating the system as it stands. It will compare a normal Linux system with the test system in several tests, and also test the performance of the test system in a normal networking environment.

6.1 The testing network

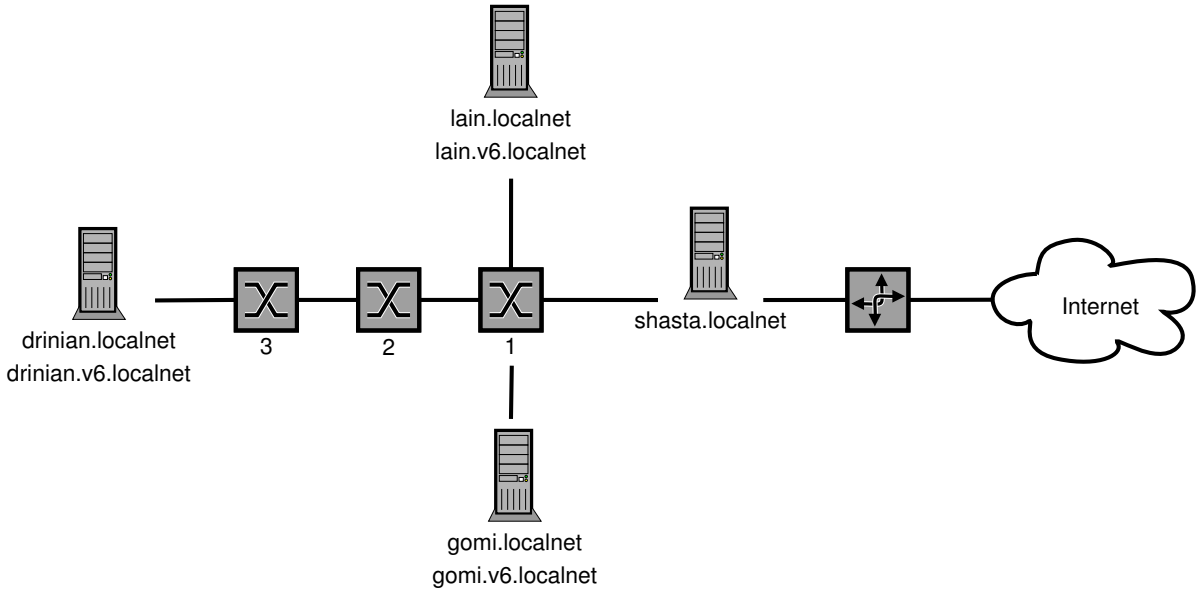

Figure 6.1: Test network layout.

Figure 6.1 shows the testing environment that was used for this project, including all the names of the hosts involved, the local domain is .localnet and all the host names on that domain resolve to v4 addresses. Some hosts also have .v6.localnet addresses. These machines are part of the local v6 network and have full v6 addresses. Due to the configuration of all the machines to use the LANs nameserver .localnet and .v6.localnet addresses can be considered to be FQDNs. The layout of this network may seem a little odd, being as the test hosts are nearer the router than the DNS but it is a representation of the homegrown testing environment that was actually used.

The network itself is composed of three 10/100 ethernet switches. These allow the two sets of machines we're concerned about to communicate, it can be assumed that the network has other machines on it which are using the network for their day-to-day business but we're unconcerned with those. On the first switch are the two test machines (lain and gomi) as well as shasta.localnet which is the gateway for the entire network and communicates via a hardware router to the Internet.

The second switch serves merely to connect the two we're interested in, both these switches have several other machines on it but most importantly the third has drinian.localnet. Drinian is the DNS server for the network, its a dual stacked Linux box that runs BIND v9 and knows about all the hosts on the network. It communicates via Shasta to speak to its upstream nameservers in the case of dealing with hosts that are outside the network. Drinian is also responsible for delegation of v6 addresses to hosts on the network.

The two main machines that were used for testing were Lain and Gomi. Lain is to be the control to demonstrate the normal system operations, Gomi is running the BIS to translate its data. Both machines are dual stack Linux machines with both v4 and v6 stacks. Lain is a Slackware Linux 9.1 box, Gomi is a Debian Linux 3.0 box. Both have hand created Kernels, Lain with 2.4.22 and Gomi with 2.4.24. Gomi is a 700Mhz Athlon machine with 384Mb of RAM, Lain is a 1.4Ghz Athlon with 256Mb of DDR RAM.

The two machines are similarly specified (i.e. neither is an SMP machine, and they are both fast machines of multiple hundred Mhz). Their connections to the network are equivalent and the network can be assumed to be lightly loaded with a verity of traffic at all points. This network layout can be seen as a reasonably common real world network, with several jumps between the machines and the outside world, and the machines and their internal servers.

6.2 The System in Operation

The goal of a BIS is to be as transparent to the user as possible, to the point where they might not even realise they're using it on their system. Therefore once the system has been put into place by the sysadmin, users must take no further action to even interact with it, it will all just work normally.

The system that is being tested however is not a full implementation of the system design shown in section 4. The DNSRH is the only component that was truly finished. The MDNSPD is a proxy for UDP only (although TCP DNS queries are rare they are possible), and thus larger DNS queries might become corrupted. Most importantly however the IOPM were not successfully implemented and are not in a testable state. The parts of the system that do exist (The DNSRH and mostly finished MDNSPD) do present a powerful option for proxying and rewriting DNS, if (in their current state) only in the highly specific manner described in section 5.

6.3 Design of the tests

The tests themselves will be run using both hosts in line with the standard scientific method of using controls and test subjects. Lain will be the control host, as its a standard Linux box. Gomi will be the test system and will throughout the tests be running the DNSRH and MDNSPD as its only method of resolving DNS. Since the goal the system is transparency then the tests will all be largely based around testing this quality. Thus they'll be designed to test the following qualities of the system:

1. Transparency
2. Speed
3. Reliability

Transparency will be a measure of differences detectable either by the user or by programs between the control system and the test system, be these in need for different options than normal of in badly formatted data being returned. Speed will be a measure of the difference in performance on the test system compared with the control, this being a part of transparency. Finally reliability will be the measure of normal system reliability compared with the test system, as malfunctions are not transparent.

6.4 System Performance

Being as the system in its current state is incomplete, that is the IOPMs cannot re-inject packets, testing will centre on examining the name resolving features of the system. Primarily transparency is the goal, and this includes the MDNSPD being a fully functional proxy whereby all normal features are available without any special modifications to data or programs by the user. However the system will also be examined under load to see how well it copes. Gomi will be the test system, and Lain the control system.

In order to test the performance of the system in an realistic as possible environment the nameserver has a number of entries in it relating to non-existent hosts. The v4 hosts are labelled "fakeN.localnet" where N is between 1 and 9999. The v6 hosts are labelled "fakeN.v6.localnet" again with a range between 1 and 9999. This should provide enough unique addresses to resolve to fully test the system. The v4 fake hosts were from 10.19.1.1 - 10.19.40.93 and the v6 fake hosts were from fec0:1:2:3::f:1 - fec0:1:2:3::f:9999. Not all these hosts will be resolved for each test but the amount of them resolved will be stated with each test.

Between each of the tests the test machine will be rebooted in order to ensure the DNSRH was clear and the system ready.

6.4.1 Standard Networking Tools

During the development and testing of the BIS much testing took place with an array of standard network testing tools. The implemented version of the project appears to work as standard with such tools as host, nslookup and most importantly dig. These tools represent the standard toolbox of sysadmins attempting to diagnose networking issues, and use the same standard as all other programs that use the DNS, meaning they are a good benchmark of transparency.

6.4.2 Resolving v4 hosts

The MDNSPD is primarily a proxy server, and as such it has to be a fast transparent service. Using a simple C testing program (found in testing/testv6 from the root of the BIS package) that tests the system time before and after performing a lookup the following data was collected.

System	Max	Min	Mean (seconds)
Test	0.4835	0.221	0.245965086
Control	0.3657	0.1424	0.174282947

Table 6.1: Comparing v4 resolving over 993 hosts

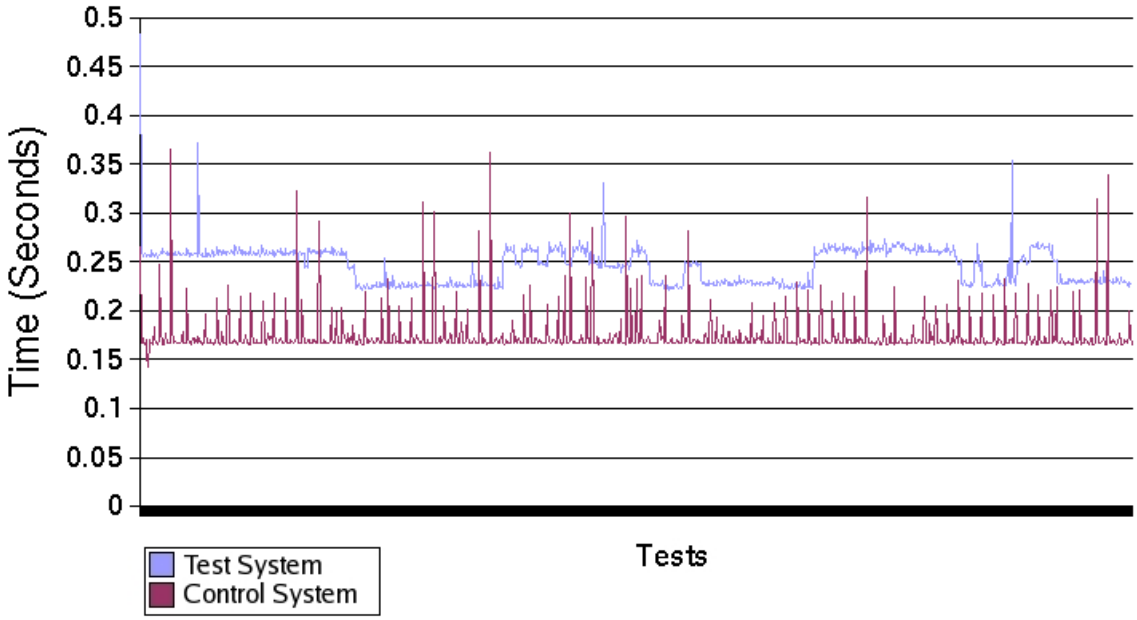

Figure 6.2: Comparing v4 resolving over 993 hosts.

The test shown in table 6.1 (figure 6.2) was conducted over 993 hosts because the test machine's MDNSPD locked up at that point. The control machine

was happy to resolve all 9999 hosts fine however (however only its data for the first 993 hosts was used in order for better comparison).

I will now examine the data produced against the three criteria stated above (section 6.3). This system can be held to be transparent for v4 address lookups, no strange behaviour was detected by the test program, and many tests appeared to work successfully. For the speed criteria we can see that on average the test system took an extra .071 seconds to complete a request. While this meant it took a little under 50% longer to perform its lookups on average being as they're still well within acceptable limits (i.e. they aren't causing the connections to time out and applications are perfectly happy with this kind of delay).

The fact that the MDNSPD locked up after 993 requests had been passed through it in quick succession means that it fails the reliability section of the evaluation. This behaviour exposes the user to the fact that they are running a BIS system and not a true v4 host. It also reveals a bug in the BIS implementation, that under heavy sequential access the MDNSPD will exhaust some form of system resource (which doesn't appear to be a memory leak, as the OS doesn't kill it), requiring a restart.

Inclusion of a sleep for 1 second between each call to the MDNSPD does not alleviate this problem, and its unclear exactly what causes this behaviour but it is a known bug in the system. Although the clients can still resolve their requests through the proxy at request number 1002 (consistently around this number if you start from 1) suddenly the time to complete each request jumps to 5.1x seconds on average and the MDNSPD stops emitting the correct debugging messages to its stdout. Clearly there is some form of resource exhaustion but what it is cannot be identified.

One possibility is that its something internal to the kernel that is causing this. The system tends to allocate roughly ascending ports for requests and it is about this point that the resolver tends to be given ports in the 62,000 range and above. The machine generally only has 65535 ports, and after slowly giving out the last of these (in my test runs the last one I saw given out was 65426) it then loops around and requests are seen coming in from port 147 (and climbing from there). Being as my name resolving is being run as a normal user process it should be unable to bind a port below 1024 and based off this I'm unsure what causes this behaviour.

6.4.3 Resolving v6 hosts

The resolving of v6 hosts tests the performance of the DNSRH element of the System, as this is where it will be called into play most, for adding, checking and controlling a massive influx of v4 to v6 mappings. While this suffers from the same bug as discussed at length the amount of data gained is still enough to give estimates of the performance impact from running this system.

System	Max	Min	Mean (seconds)
Test	0.7049	0.3745	0.423416303
Control	0.5658	0.1947	0.223601163

Table 6.2: Comparing v6 resolving over 963 hosts

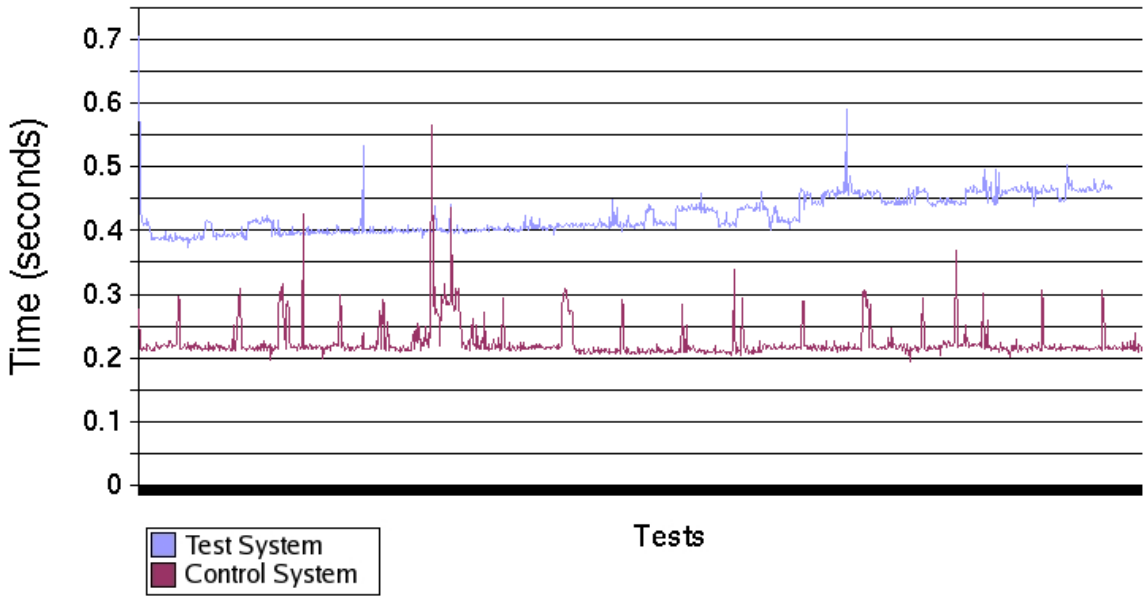

Figure 6.3: Comparing v6 resolving over 963 hosts.

The results for the tests shown in table 6.2 and figure 6.3 are very similar to the results of the previous test. The system appears to be totally transparent to user queries, the test program ran without a problem in its function calls. Also if we examine the speed of these tests we'll again see that the BIS was slower than the control system by 0.2 seconds. This was more significant than the previous test as the test system took on average almost double the time the regular system took to perform the lookup. This stems from the translation from A to AAAA and doing two individual lookups I would imagine from my knowledge of the system internals. However 0.423 of a second seems to be a reasonable amount of time to complete a DNS lookup in as again the test program completed fine, as did all the standard tools used in debugging.

However the MDNSPD locked up to the point of requiring it be restarted under this load, and again the control host didn't. This time it locked up at 963 hosts, whereas before it was 993, this could be linked to it being a v6 test (perhaps its a resource leak that leaks quicker with v6). This obviously points to a consistent bug in the design of the MDNSPD which cannot be identified except to say its consuming a system resource thats required.

6.4.4 Reverse resolving of v6 hosts

An important issue that the MDNSPD needs to address in terms of transparency is that it can cope with reverse v6 DNS lookups. Reverse lookups are used by a variety of system servers and users applications in order to help reduce the possibility of IP address spoofing and other problems. Its also useful for debugging networking issues for admins and users who are trying to configure programs. For these reasons it is essential that the MDNSPD can reverse its lookups from its fake v4 addresses to the FQDN of the real v6 associated with that address without any user intervention.

The MDNSPD partially implements this functionality. While it transparently translates PTR requests (those that map an IP address to a hostname) when they are outgoing it doesn't currently rewrite them correctly on the way back in. While the user will get their data (the answer will contain the canonical hostname associated with IP address) the response from DNS will include the so called v6 nibble address[7]. This might confuse a user or a program as instead of being the FQDN form of the v4 address (for example 240.0.0.1 is translated to the domain name 1.0.0.240.in-addr.arpa before being sent) its the form of the v6 address (for example 3.5.2.0.0.0.0.0.0.0.0.0.0.0.0.0.3.0.0.0.2.0.0.0.1.0.0.0.0.c.e.f.ip6.arpa). However accepting this slight flaw the system does return the data that is required and except in specific circumstances this will do.

6.4.5 Use of the MDNSPD as a DNS

Finally to be truly useful the MDNSPD must be able to translate requests for foreign hosts. If it couldn't do this no machine running it could also run a name server (without some complicated packet forwarding) which might limit its effectiveness on the network. With the MDNSPD a host can appear to be one of the local nameservers, for example for a number of regular v4 hosts surrounded by v6 hosts.

To test this the same v4 test as earlier was performed on Lain, except that Lain was using Gomi as for DNS, all its requests were really being relayed up to Drinian. 2000 requests were made, without pause (which required quickly restarting the MDNSPD on Gomi half way through, and once more before

the end). These were for "fake1.localnet" to "fake2000.localnet"

Max	Min	Mean (seconds)
0.47	0.2	0.22

Table 6.3: Proxying v4 requests via the test machine, over 2000 hosts.

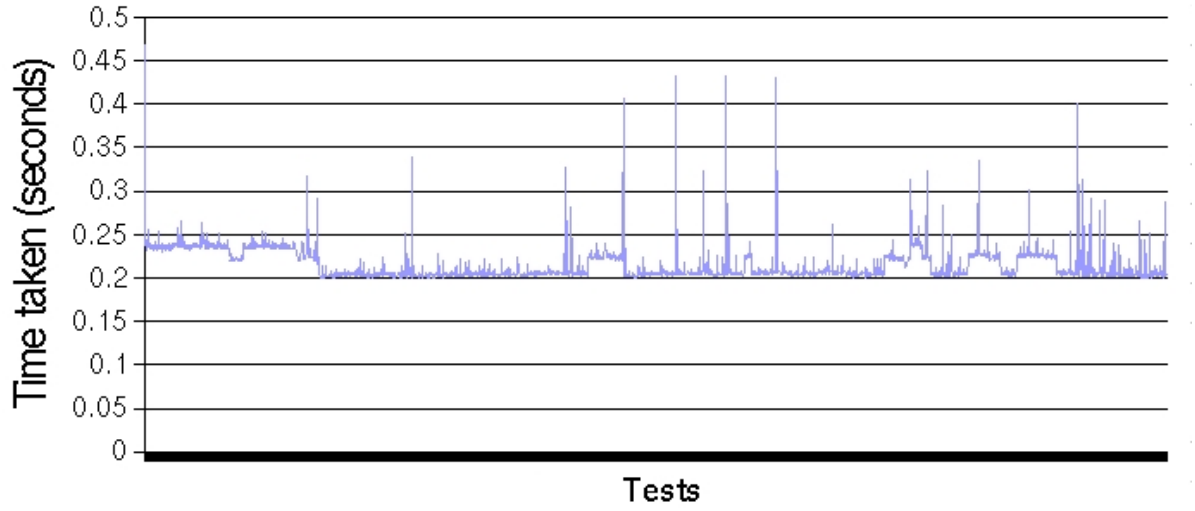

Figure 6.4: Proxying v4 requests via the test machine, over 2000 hosts.

This technique also works for resolving v6 hosts to the 240.x.x.x fake addresses. The use of this for hosts that aren't themselves running this particular BIS is limited, but assuming several hosts on a network were running this then it could help a systems administrator to debug routing issues. Failing that if there were a number of machines using this form of BIS they could share one of them as a singular nameserver. Doing this would enable them to refer to v6 machines by a common v4 address.

As can been seen from table 6.3 and figure 6.3 this test performed much as the previous tests. It appeared to be totally transparent, which indicates that the rewriting features of the MDNSPD are implemented correctly. Comparing these results to the results found in table 6.1 we can see that it takes on average 0.174 to resolve a host on the control system normally and 0.22 when going via the test system as a proxy. This indicates a difference of 0.046, which is on average an insignificant increase in the time taken for the operation. This means that this is a valid tactic if you wish the DNS to be synchronised on a network with multiple BIS hosts. However again the reliability let the system down, and means that this system is very unreliable under a heavy load, so having multiple BIS hosts sharing one of them as a consistent DNS could be a very dangerous idea.

6.5 Security

This project carries with it many security implications. It modifies an otherwise presumed secure system by inserting interdependent modules that do a lot of memory management into the kernel. These are allowed to communicate with any users via a proxy written in the C programming language, which binds by default to an externally visible low port running as the superuser.

The hash itself while first appearing secure is storing and modifying many data structures in kernel space at once and therefore carries a risk of being dangerous to the running of the system. It accepts data from the outside world, ultimately from users and their programs and not just from other services on the machine and therefore must be seen as a possibly vulnerable module. However in its favour the access to it is limited, the filesystem permission model is used to restrict access to the device file (which is required to input and output data to this module). Therefore if this module is correctly set up a program would need root privileges to communicate with the DNSRH, and if they have root privileges then they can modify the kernel anyway.

The greatest security risk for this project comes from its communication with the outside world. With reference to the earlier Implementation chapter I will now outline how machines running this system could be hit by a DOS attack.

Assuming the attacker knew that your machine was running this BIS then a malicious series of packets such as a forged source UDP port scan with a random source for each port scanned, or even a program designed simply to send a machine running the MDNSPD on an externally visible port a lot of fake DNS responses could cause a denial of the DNS service to your machine or even lock it up entirely. If the system encounters a large volume of lookups two main effects will occur. The first is that the kernel itself will bloat in memory, this will degrade overall system performance leaving less memory for all other processes that exist, also because of the design of the hash tables (chained pointers) then overall lookup speed could be degraded quite badly.

The second is that the data structures inside the DNSRH will become locked as the relatively long and expensive addV6Addr() function will be called. This function is estimated to be the slowest of the functions that access the data structures in the DNSRH, and since this module has been written with SMP (Symmetric Multi-Processing) machines in mind then all access to the hash tables must be controlled by Mutexes, meaning only one query can

access them at once. The repeated and long access and locking of these data structures will slow down the amount of packets that the machine can process going either in or out as all of these require a lookup.

All these points however are fairly rare, the system is constructed to be mindful of user input and deal with it very carefully, and the rest of the issues largely come down to if the sysadmin has vandelous users on the local system, as with correct network firewalling and permissions most of these problems can be prevented, but regardless of these points any sysadmin running this system must be careful and try to protect themselves from [D]DoS attacks.

Since the bulk of the project exists as modules inside the kernel these cannot be secured beyond the level they are now by the sysadmin. However they are generally not visible to the outside world. Several mechanisms exist for securing the MDNSPD, without changing this implementation of it.

Primarily it can be made to run inside a chroot (CHange ROOT) jail. The idea of this is that you build an entire file system with only what is needed inside it (for example in /opt/mdnspd). Then run the program inside the chroot wrapper, from the point of view of the program the root of the filesystem is now the top of the chroot jail. This will not stop the program from being exploited remotely to push too much data into the DNSRH but it will provide protection against the possibility of buffer overflow in the MDNSPD, as the only things that an attacker will be able to damage is the chroot jail, which contains nothing of value.

Apart from this using iptables[32] (the Linux firewall tool) in order to prevent access to port 53 from anywhere other than localhost will restrict the vectors of attack on the MDNSPD (and through it the entire machine) to those with local user accounts, which reduces the risk significantly.

6.6 Evaluation Summary

While the full project is incomplete I have established that there is enough of it finished to be worth testing. As can be seen from section 6.4.2 and section 6.4.3 the system is fully capable of resolving both v4 and v6 addresses, and consistently changing the v6 addresses to v4s. It also seems to work at a comparable speed to a normal host, with only a slight increase in the time taken to resolve FQDNs. Section 6.4.4 proves that the system is fairly transparent to the user, while they will see a little erroneous data being passed back the reverse lookups still work. Finally from section 6.4.5 we can see that a single BIS host could be able to resolve all v6 addresses for all the BIS hosts on a network, this would lead to a consistent v4 only network-view for all the

BIS hosts, leading to easier communication between applications on them (as they could all refer to the same 240.x.x.x addresses. Finally in section 6.5 we saw that while there are many security risks associated with this project, mainly [D]DoS issues, these can be largely avoided by placing the machine in a private network and performing some simple sysadmin tasks.

It is also worth considering how real applications would react to the presence of this BIS. Primarily v6 only applications would, in this current implementation, end up being badly broken by this BIS. All of their outgoing traffic would be sent fine, but they would only receive v4 packets in response which would all be marked (from their point of view) with an incorrect destination address. v4 applications on the other hand would largely work fine with this BIS in place. Applications which stream media, and thus rely on timeliness of data may be affected to some degree by the delay in their packets being mangled but I feel it will be quite insignificant compared to the time it takes to traverse the network. All other v4 applications that are in common use (Email, Web, Instant messengers, file transfer, ssh etc) should work properly with this BIS, being unable to see that they aren't truly on a v4 network. This obviously makes the BIS a useful transition method.

7 Conclusion

7.1 Fulfilment of aims

The main aim of this project was to enable a dual-stack machine to live in an v6 island while appearing to be an v4 machine to its users. Its goal was to create another means of helping the transition from v4 to v6, and to make that transition as easy as possible.

In these regards the project failed. It cannot be used to communicate with a v6 island, or to appear to be v4 while really transmitting v6 traffic. The IOPM was never completed to the point where it could re-inject packets, leaving only half the project fully completed.

However the MDNSPD and the DNSRH are good integrated modules, either of which could be rewritten as they communicate via IOCTL calls, which are available in many programming languages. These two components provide a good framework and partial reference implementation of a BIS.

7.2 Revisions to Design and Implementation

If this project was being reattempted it would stand a better chance of success. Several parts of the system needed to be redesigned during the implementation of the project in order to work around problems or incorrect assumptions that had been made during the design of the project.

The first example of this was the method of kernel communication. Traditionally there are system calls that enable a userspace application to communicate with kernel space. However under Linux these are written into the kernel headers and should only be changed when agreed upon by the official kernel maintainers. If module authors start to change them they may end up stepping on each others changes. For this reason the project is currently implemented using the unintuitive interface of masquerading as a hardware device.

During the original design and implementation of the MDNSPD I didn't anticipate needing to rewrite the outgoing requests from A to AAAA queries (v4 to v6) and this important stage thus took longer than expected.

During the attempted implementation of the IOPM, which was the final stage of the project a further complication was discovered. In stock Linux kernels an archaic header file (/usr/src/linux/include/linux/in.h:216,248) marks the entire v4 class E address space (which includes the 240.0.0.0/8 range that this

project uses for v6 mappings) as off limits. This problem took time to debug and reduced the portability of the code as it requires patching the kernel source and hand building it before this project could ever possibly work.

In addition to this many of the netfilter functions are not exported into the kernel at large, this made attempts at packet re-injection harder. Based on my reading of the netfilter mailing list[33] and associated documentation I would say that use of some of these functions will be critical in getting the IOPMs working, and so in order to do this the source (/usr/src/linux/net/ipv6/ipv6_ksyms.h) was changed to export the function symbols into the kernel namespace as a whole.

Apart from these small problems the design of this project is a good modular piece of work enabling chunks to be interchanged or written by third parties and hopefully the data will be separated by a strong enough set of interfaces (in the form of well defined functions) that they will be truly separate and not badly entangled. Such open design is a huge advantage for a project as it enables people to understand the design by reading the source and documentation for each small section, and good overviews such as found in sections 4 and 5.

7.3 Future Work

This projects unfinished status suggests good future work. Being as it actually provides a useful service for helping users migrate their hosts from v4 to v6 then it would be worth the effort of working on it further. Certainly even if the MDNSPD was just finalised to include TCP proxying and full reverse DNS translation it could be a useful tool for network admins to redefine the view their users see of the DNS world.

More importantly for the entire project the packet re-injection code for the IOPM needs to be finished. Theoretically this should be well inside the scope of the technology (netfilter) used to do it and thus should be possible with more time.

This project also needs the final touches to complete it. The implementation requires packaging, proper Makefiles written for it to enable it to be easily installed without doing everything by hand. It also needs the standard files distributed with it such as INSTALL, README, CHANGES etc. These will help any sysadmin who decides to make some use of the code, and any coder who decides to modify it. Finally the code could probably do with a review for clarity, if not security as it was coded at speed and is written in ANSI C, a language where its easy to write buggy code.

The MDNSPD could be improved by making it only keep the root permissions for as long as it takes to bind itself to port 53. After this it could be made to drop its permissions to a user with far less access ("nobody" is normally used for this, however a special "dnsrh" user could be created). However if this is done then access to the /dev/dnsrh would need to be changed to allow this new user to access it. Doing this would however reduce the risks associated with running network visible servers as root.

The performance of the DNSRH, possibly the most central section of this project, is I'm sure improvable. It is implemented in a simple and robust manner (see sections 5.1.1 and 5.1.4) but the algorithms and the design behind it, especially the design behind the functions that actually generate hash keys could be made more efficient, to give a greater spread, or possibly so that a v4 and v6 address could be passed through two functions to give one hash-key. The last point would be the most efficient way of adding elements to the hash as then only a single table of hash keys would be required to store all the lookups, instead of two arrays of pointers to the same data.

The IOPMs could also make use of the connection tracking features of Netfilter in order to track which streams of packets they are modifying. Doing this would enable v6 only applications to work on the BIS host, as their packets would be passed both in and out without being modified, whereas v4 applications would still have all their packet bumped in the stack.

7.4 Lessons Learnt

This project has been educational, informative and enjoyable if nothing else. In future projects I will learn to allocate more time for the low level tasks as I've learnt a lot about debugging these kinds of things during this project. I have also learnt that protocols with binary headers (like DNS, instead of for example HTTP) are harder to modify on the fly, even if they do save on bandwidth. I've learnt more about the Linux kernel internals than I ever thought I would as well as a lot about working around issues compiling it. This project has also taught me a lot about v6, routing, v4, the setup of BIND as a name server, the art of debugging protocols by sniffing the traffic and miscellaneous packet filtering tricks.

7.5 Final Overview

This project still has several issues with it, mostly relating to the uncompleted IOPMs. However I would consider this project to be useful overall, and not just for the amount of knowledge and experience I've gained from it. Chiefly since its licensed under the GPL it provides a good working framework for

anyone else attempting to build a BIS. Secondly it provides an almost complete DNS proxy for anyone who wishes to create a project based on this. Such a thing would enable a network administrator to change the way their users viewed the network by simply hiding somethings from DNS and making other things resolve to other places. While this breaks a number of Internet standards some net admins may appreciate this functionality. Lastly this project is close to being finished, so theoretically a fully working implementation of it is not that far off from being completed and distributed, which will hopefully be of help in the upcoming transition away from v4 to the new v6 Internet.

8 Acknowledgements

The following individuals and groups helped with the production of this project.

The Opensource and Free Software Communities For documentation and the open OS to write this project on.

The Netfilter Project For the documentation and mailing list about Netfilter, particularly Rusty Russell and Harald Welte.

The LuBBs Community Lancaster University Bulletin Board System is a community with a strong technical slant operating at Lancaster University, they have answered many programming questions.

Graham Clinch For answering many technical questions relating to Unix systems, network routing and the compilation of the Kernel.

References

[1] Mordechai T. Abzug. MD5 Homepage (unofficial). http://userpages.umbc.edu/ mabzug1/cs/md5/md5.html

[2] APNIC (2001). IPv6 Address Allocation and Assignment Global Policy. Draft of December, 22 2001. Version 2001-12-22. APNIC, ARIN, RIPE - NCC. APNIC. ftp://ftp.cs.duke.edu/pub/narten/global-ipv6-assign-2001-12-22.txt

[3] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss (1998). RFC 2475 - An Architecture for Differentiated Service. IETF Network Working Group. http://www.faqs.org/rfcs/rfc2475.html

[4] R. Braden et al (1989). RFC 1122 - Requirements for Internet Hosts - Communication Layers. IETF Network Working Group. http://www.faqs.org/rfcs/rfc1122.html

[5] R. Braden, D. Clark, S. Shenker (1994). RFC 1633 - Integrated Services in the Internet Architecture: an Overview. IETF Network Working Group. http://www.faqs.org/rfcs/rfc1633.html

[6] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, S. Jamin (1997). RFC 2205 - Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. IETF Network Working Group. http://www.faqs.org/rfcs/rfc2205.html

[7] M. Crawford, C. Huitema (2000). RFC 2874 - DNS Extensions to Support IPv6 Address Aggregation and Renumbering. IETF Network Working Group. http://www.faqs.org/rfcs/rfc2874.html

[8] S. Deering, R. Hinden (1998). RFC 2460 - Internet Protocol, Version 6 (IPv6) Specification. IETF Network Working Group. http://www.faqs.org/rfcs/rfc2460.html

[9] Andrew Ebling (2002). Kernel Hacking HOWTO. http://www.kernelhacking.org/docs/kernelhacking-HOWTO/indexs03.html#kernel-kernelmodes

[10] Chris Edwards (2004). Internet Protocol Version 6 (IPv6). http://info.comp.lancs.ac.uk/year3/notes/options/353/notes/L12.pdf

[11] K. Egevang, P. Francis (1994). RFC 1631 - The IP Network Address Translator (NAT). IETF Network Working Group. http://www.faqs.org/rfcs/rfc1631.html

[12] Jim Frost (1996). BSD Sockets: A Quick And Dirty Primer. Software Tool & Die. http://world.std.com/ jimf/papers/sockets/sockets.html

[13] V. Fuller, T. Li, J. Yu, K. Varadhan (1993). RFC 1519 - Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy. IETF Network Working Group. http://www.faqs.org/rfcs/rfc1519.html

[14] Peter H. Gleick (2001). "Making Every Drop Count". Scientific American. http://www.sciam.com/issue.cfm?issueDate=Feb-01

[15] The GNU project. http://www.gnu.org/

[16] The GNU Project (1991). The GNU GPL. Free Software Foundation. http://www.gnu.org/copyleft/gpl.html,

[17] Michael Hauben. History of ARPANET. http://www.dei.isep.ipp.pt/docs/arpa.html

[18] R. Hinden, M. O'Dell, S. Deering (1998). RFC 2374 - An IPv6 Aggregatable Global Unicast Address Format. IETF Network Working Group. http://www.faqs.org/rfcs/rfc2374.html

[19] R. Hinden, S. Deering (2003). RFC 3513 - Internet Protocol Version 6 (IPv6) Addressing Architecture. IETF Network Working Group. http://www.faqs.org/rfcs/rfc3513.html

[20] IANA (2002). RFC 3330 - Special-Use IPv4 Addresses. IETF Network Working Group. http://www.faqs.org/rfcs/rfc3330.html

[21] IANA (2004). PORT NUMBERS. Internet Assigned Numbers Authority. http://www.iana.org/assignments/port-numbers

[22] IANA (2004). IANA: INTERNET PROTOCOL V4 ADDRESS SPACE. Internet Assigned Names Authority. http://www.iana.org/assignments/ipv4-address-space

[23] ISC. ISC BIND9. Internet Systems Consortium, Inc. www.isc.org/sw/bind/bind9.php

[24] Raj Jain (2001). IP Multicasting: RFCs. Ohio State University. http://www.cis.ohio-state.edu/ jain/refs/ipm_refs.htm#IP_Multicasting_RFCs

[25] D. Johnson, S. Deering (1999). RFC 2526 - Reserved IPv6 Subnet Anycast Addresses. IETF Network Working Group. http://www.faqs.org/rfcs/rfc2526.html

[26] Michael K. Johnson (1996). Device Drivers. http://en.tldp.org/LDP/khg/HyperNews/get/devices/

[27] E. Nordmark (2000). RFC 2765 - Stateless IP/ICMP Translation Algorithm (SIIT). IETF Network Working Group. http://www.faqs.org/rfcs/rfc2765.html

[28] Linus Torvalds homepage. http://www.cs.helsinki.fi/u/torvalds/

[29] Linux.org http://www.linux.org/

[30] The Linux Kernel Repository. http://www.kernel.org/

[31] LIDS (Linux Intrusion Detection System). http://www.lids.org

[32] The Netfilter/iptables project http://www.netfilter.org

[33] Netfilter. Mailinglists of he netfilter/iptables project. Netfilter Project. http://www.netfilter.org/mailinglists.html

[34] G. Malkin, T. LaQuey Parker (1993). RFC 1392 - Internet Users' Glossary. IETF Network Working Group. http://www.faqs.org/rfcs/rfc1392.html

[35] P. Mockapetris (1987) RFC 1035 - Domain names - implementation and specification. IETF Network Working Group. http://www.faqs.org/rfcs/rfc1035.html

[36] Jon Postel et al. (1981). RFC 791 - Internet Protocol. DARPA Internet Program. http://www.faqs.org/rfcs/rfc791.html

[37] Eric S. Raymond (2003). The Rampantly Unofficial Linus Torvalds. http://www.catb.org/ esr/faqs/linus/index.html

[38] Eric S. Raymond (2004). Definition of daemon. The Jargon File. http://www.catb.org/ esr/jargon/html/D/daemon.html

[39] Eric S. raymond (2004). Definition of deep magic. The Jargon File. http://www.catb.org/ esr/jargon/html/D/deep-magic.html

[40] Eric S. Raymond (2004). Definition of userland. The Jargon File. http://www.catb.org/ esr/jargon/html/U/userland.html

[41] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, E. Lear (1996). RFC 1918 - Address Allocation for Private Internets. IETF Network Working Group. http://www.faqs.org/rfcs/rfc1918.html

[42] J. Reynolds, J. Postel (1984). RFC 900 - Assigned Numbers. IETF Network Working Group. http://www.faqs.org/rfcs/rfc900.html

[43] R. Rivest (1992). RFC 1321 - The MD5 Message-Digest Algorithm. IETF Network Working Group. http://www.faqs.org/rfcs/rfc1321.html

[44] Alessandro Rubini & Jonathan Corbet (2001). Chapter 3: Char Drivers. Linux Device Drivers, 2nd Edition. O'Reilly. http://www.xml.com/ldd/chapter/book/ch03.html

[45] Paul 'Rusty' Russell (2000). Netfilter Tutorial: LinuxWorld: San Jose August 2000. http://www.netfilter.org/documentation/tutorials/lw-2000/tut-1.html

[46] Paul 'Rusty' Russell (2001). Linux Networking-concepts HOWTO. The Netfilter Project. http://www.netfilter.org/documentation/HOWTO//networking-concepts-HOWTO.html

[47] Paul 'Rusty' Russell, Harald Welte (2001). The Linux netfilter Hacking HOWTO. The Netfilter Project. http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.html

[48] Andrs Salamon (2004). DNS related RFCs. DNSRD. http://www.dns.net/dnsrd/rfc/

[49] Peter Jay Salzman, Ori Pomerantz (2001). The Linux Kernel Module Programming Guide. http://www.dirac.org/linux/writing/lkmpg/lkmpg.html

[50] sdybiec@humanfactor.com. Programming POSIX Threads. http://www.humanfactor.com/pthreads/

[51] Richard Stallman (2004). Linux and the GNU Project. The GNU Project. http://www.gnu.org/gnu/linux-and-gnu.html

[52] Andrew S. Tanenbaum, Albert S. Woodhull (1997). Operating Systems: Design and Implemenation. Prentice Hall, New Jersey. pg 37-38.

[53] Andrew S. Tanenbaum (1996). Computer Networks, Third Edition. Prentice Hall, New Jersey. pg 437

[54] Andrew S. Tanenbaum (1996). Computer Networks, Third Edition. Prentice Hall, New Jersey. pg 443

[55] G. Tsirtsis, P. Srisuresh (2000). RFC 2766 - Network Address Translation - Protocol Translation (NAT-PT). IETF Network Working Group. http://www.faqs.org/rfcs/rfc2766.html

[56] K. Tsuchiya, H. Higuchi, Y. Atarashi (2000). RFC 2767 - Dual Stack Hosts using the "Bump-In-the-Stack" Technique (BIS). IETF Network Working Group. http://www.faqs.org/rfcs/rfc2767.html

[57] Unknown. Big endian and Little endian computers. Rensselaer Polytechnic Institute. http://www.cs.rpi.edu/courses/sysprog/sockets/byteorder.html

[58] Unknown. Lab Exercise 2: Adding a Syscall. http://www.cc.gatech.edu/classes/AY2001/cs3210_fall/labs/syscalls.html

[59] Unknown. NAT-PT. Eurescom. http://www.eurescom.de/ public-webspace/P1000-series/P1009/doc3_1.html

[60] U.S. Census Bureau (2004). World POPClock. U.S. Census Bureau. http://www.census.gov/cgi-bin/ipc/popclockw

[61] Wayde (wallen@boulder.nist.gov). #3 - The UNIX philosophy. http://lug.boulder.co.us/docs/newbie03.html

[62] Harald Welte (2000). The netfilter framework in Linux 2.4. http://gnumonks.org/papers/netfilter-lk2000/presentation.html

[63] Harald Welte (2000). skb - Linux network buffers. http://gnumonks.org/ftp/pub/doc/skb-doc.html

[64] Harald Welte (2000). The journey of a packet through the linux 2.4 network stack. http://gnumonks.org/ftp/pub/doc/packet-journey-2.4.html