

MASTER OF ENGINEERING THESIS IN COMPUTER SCIENCE

Multihoming with Internet Protocol Version 6

Troels Walsted Hansen

December 21, 1999

Department of Computer Science

Faculty of Science
University of Tromsø

Preface

This work was created in the fall of 1999 and represents the final part of my Master of Engineering¹ degree in Computer Science at the University of Tromsø.

The work was done in the context of the PASTA and Vermicelli projects at the Department of Computer Science.

The PASTA project is a joint effort between researchers at the Departments of Computer Science at the Universities of Pisa, Italy, and Tromsø, Norway. The goal of the project is to understand the effects portability has on systems, using an experimental approach. The project explores the potential and integration of devices such as Personal Digital Assistants (PDAs), digital cellular phones and smartcards.

The Vermicelli project sprung from the PASTA project, to work on the integration of mobile machines into a static infrastructure. The name is an acronym for VERy MobiLe Clients Enhanced for Link Level Interaction. The infrastructure for the project is built with the next generation Internet protocol, IPv6, because of the increased flexibility offered by this technology.

Acknowledgments

First of all, a big thanks to my supervisors, assistant professor Tage Stabell-Kulø and research fellow Feico Dillema. I learned a lot from you this semester.

PASTA project member and research fellow Dag Brattli read the draft of the chapter on multihoming technologies and provided some very valuable comments. Thank you.

To my mother for always being an inspiration to do my best, and for providing encouragement when motivation was lacking. Thank you also for reading the draft and helping me improve the language.

¹Norwegian: “Sivilingeniør”.

Abstract

The Next Generation Internet Protocol, IPv6, is coming. In the IPv6 Internet, it will be common for nodes to have multiple addresses, to be *multihomed*. This creates a host of challenging problems which are only now being addressed.

In this thesis we provide an overview of the state of the art in IPv6 multihoming technology, by looking at potential benefits, problems and proposed solutions. We proceed to state requirements for a solution which will allow connections between two hosts to be migrated to a different link in the event of link failure or address changes. The most important requirement is that the solution is completely independent of router interaction.

In the design phase, we see that the requirements can be fulfilled in two different ways. The first is through a new user level API which provides an abstraction away from the underlying IP addresses of the connection, and connects to DNS names. Through a session layer protocol over TCP, it reconnects and resumes connections transparently.

The second solution is a kernel level modification to the transport protocol, TCP, which allows a set of alternative addresses to be exchanged at connection establishment. In the event of link failure or address change, it will automatically change to a different address and keep the connection alive.

Both solutions are implemented in a FreeBSD/KAME environment and compared in terms of functionality and performance. The kernel solution is shown to hold the upper hand in most areas, while still being dependent on parts of the API's functionality.

Contents

1	Introduction	1
1.1	Background	1
1.2	Problem statement	2
1.3	Method	2
1.4	Outline	4
2	Multihoming technologies	5
2.1	Multihoming coming into focus	5
2.2	Definition of multihoming	5
2.2.1	One network interface, multiple addresses	5
2.2.2	Multiple network interfaces	6
2.3	Potential multihoming benefits	7
2.3.1	Fault-tolerance	7
2.3.2	Load sharing	7
2.3.3	Provider selection	7
2.3.4	Eased renumbering transition	8
2.3.5	Enhanced mobility support	8
2.4	Problems associated with multihoming	8
2.5	Multihoming and DNS	10
2.6	Multihoming with IPv4	12
2.6.1	Router based solutions	12
2.6.2	Host based solutions	13
2.7	Multihoming with IPv6	14
2.7.1	Mobile IPv6	14
2.7.2	Router based solutions	15
2.7.3	Host based solutions	17
2.8	Summary	18

3	Requirements	19
3.1	System model	19
3.2	Functional requirements	20
3.3	Non-functional requirements	20
4	Design	23
4.1	Exploring different designs	23
4.2	The namesocket user space API and protocol	23
4.2.1	Functions in the API	25
4.2.2	The namesocket communication protocol	27
4.2.3	The namesocket packets	30
4.2.4	Summary	31
4.3	TCP connection preservation at kernel level	31
4.3.1	Fitting into the TCP state machine	32
4.3.2	The PREFIX_SYN and PREFIX_ACK header options	32
4.3.3	Demultiplexing IP packets for N-to-M connections	35
4.3.4	Address change policy	36
4.3.5	Summary	37
4.4	Summary	38
5	Implementation	39
5.1	The namesocket user space library	39
5.1.1	External API functions	39
5.1.2	Internal utility functions	43
5.1.3	Circular buffer	45
5.1.4	Logging facility	45
5.1.5	namesocket server	46
5.1.6	namesocket client	47
5.1.7	Summary	47
5.2	TCP connection preservation at kernel level	47
5.2.1	Adding state to the TCP control block (tcpcb)	48
5.2.2	Modifications to tcp_input()	48
5.2.3	Modifications to tcp_output()	48
5.2.4	Constructing IP6OPT_PREFIX_xxx headers	49

5.2.5	Changing address for a connection	49
5.2.6	Efficient demultiplexing of IP6 TCP packets	50
5.2.7	Summary	51
5.3	A comparison of the two implementations	51
5.3.1	User vs kernel space	52
5.3.2	Application modification	52
5.3.3	Dealing with address changes	52
5.3.4	Overhead	53
5.3.5	Interoperability	53
5.3.6	Limitations imposed	54
5.3.7	Portability	54
5.3.8	Backwards compatibility	55
5.3.9	Security	55
5.3.10	Summary	56
6	Demonstration	57
6.1	Demo configuration	57
6.2	Namesocket library	57
6.2.1	Summary	62
6.3	PreserveTCP kernel modification	63
6.3.1	Summary	65
7	Performance evaluation	67
7.1	Choosing measures	67
7.2	Making measurements	67
7.3	Expected results	69
7.4	Measured results and analysis	69
7.5	Summary	71
8	Conclusion	73
8.1	Achievements	73
8.2	Future work	73
	References	75
A	The CDROM	81

Contents

List of Figures

1.1	Our software development method.	3
2.1	The hierarchical DNS name space.	10
2.2	Example DNS tree down to host level, with associated AAAA Records.	11
2.3	Illustration of RFC 2260: Scalable Support for Multi-homed Multi- provider Connectivity	13
2.4	Simple multihoming routing, assuming direct ISP peering.	16
3.1	The system model.	19
4.1	Fitting the namesocket protocol into the TCP/IP protocol stack.	27
4.2	The namesocket protocol state diagram.	28
4.3	The namesocket protocol packet types and layouts.	30
4.4	TCP state transition diagram for connection establishment.	33
4.5	Generic IPv6 headers and a PREFIX_SYN header with two 64 bit prefixes.	33
4.6	Illustrating the change from 1-to-1 to N-to-M addresses.	36
5.1	The structure which holds namesocket state.	40
6.1	The demo host and network configuration.	57

List of Figures

List of Tables

4.1	Summary of namesocket API functions, inputs and return values.	25
4.2	Prefix option header values.	34
7.1	Results from performance measurements.	70

List of Tables

1 Introduction

1.1 Background

The mainstream Internet of today runs almost exclusively over a protocol known as the Internet Protocol Version 4 (IPv4) [32]. Since the original version [3], this protocol has survived 25 years of dramatic Internet growth, from just a few university sites to an estimated 56 million hosts of July 1999¹. More and more applications are abandoning their previous carrying protocols and moving to IP based networks, for example telephony and movie on demand solutions. Devices ranging in size and capabilities from mobile phones to mainframe computers can today communicate using IP. Unfortunately, IP is becoming a victim of its own success, the address space that probably seemed infinite at the time of the protocol's conception, is rapidly running out. IPv4 addresses are 32 bits in length, resulting in a theoretical address space of more than 4 billion. Unfortunately, due to the original hierarchical Class A, B and C structure of the addresses [27], as well as a very round-handed approach to handing out addresses early on, there are not many addresses left. To combat the problem, techniques like Classless Inter-Domain Routing (CIDR) [34] and Network Address Translation (NAT) [14] have been developed, and both have helped to stretch the lifetime of IPv4 further than what was previously predicted [22, p.312].

CIDR is a technique which breaks up the old fashioned hierarchical addresses into more manageable pieces, allowing more efficient address allocation and routing. Combined with a new address allocation strategy [17], CIDR has served to prolong to life of IPv4 significantly. Through aggregatable addresses, CIDR greatly decreased the size of the routing tables in the backbone of the Internet. Unfortunately, because of the way CIDR calls for addresses to be allocated out of the address space of the Internet Service Provider (ISP), additional routes must still be advertised into the backbone of the Internet when a site is connected to multiple providers.

NAT takes a different approach. Instead of allocating a unique address to every node in the network, only one node, the NAT device, is assigned an Internet address. The other machines in the network are assigned addresses which cannot be routed on the Internet, and can only communicate with Internet hosts through the NAT device. However, NAT based techniques do not come without a cost. Most importantly, they are only suitable for client-only hosts, and while

¹Source: Internet Software Consortium <http://www.isc.org>.

1 Introduction

these are definitely the most numerous in today's Internet, NAT has inherent problems which limit the kind of applications a host can be used for.

The solution, according to most experts in the field, is the next generation of the Internet Protocol, known as Internet Protocol Version 6 (IPv6) [9]. This protocol was standardized in 1995 and carries a 128 bit long address field. 128 bits of address space is equivalent to approximately 3×10^{38} unique addresses, hopefully enough for *at least* another 25 years of Internet growth.

While many implementations of IPv6 exist today, and large-scale testing is in place on the 6bone², IPv6 development continues at a rapid rate with solutions being developed to both old and new problems.

The problem which is receiving the most attention at the moment is how to deal with *multihoming*, the Internet term for assigning multiple addresses to a host. Challenges arise both when multiple addresses are assigned to a single network interface, and when there are multiple network interfaces with individual addresses. Both scenarios are expected to occur more frequently in the coming IPv6 Internet. The first as a result of the IPv6 concept of scoped addresses (link-local, site-local and global) and the second because of decreasing cost of Internet links. The IPv6 address architecture [21] is modeled on CIDR and consequently fails to handle multihomed routing without blowing up the routing tables, creating a demand for new solutions.

1.2 Problem statement

The problem statement which will guide this work is:

Survey current IPv6 multihoming technology, with focus on potential benefits, problems and proposed solutions. Develop host based solutions to help applications take advantage of multihoming benefits.

This is a fairly open problem statement, which leaves room for experimentation and for exploring different solutions to the problems. This is necessary to work in a rapidly developing field of study where different proposals are bound to appear and change along the way.

1.3 Method

There are many different software engineering processes proposed in the literature. Most of them, including the popular *waterfall* model [35], divide the development process into a set of distinct steps, and specify the order in which

²A world wide IPv6 network, largely routed through tunnels in the IPv4 Internet.
See <http://www.6bone.net>.

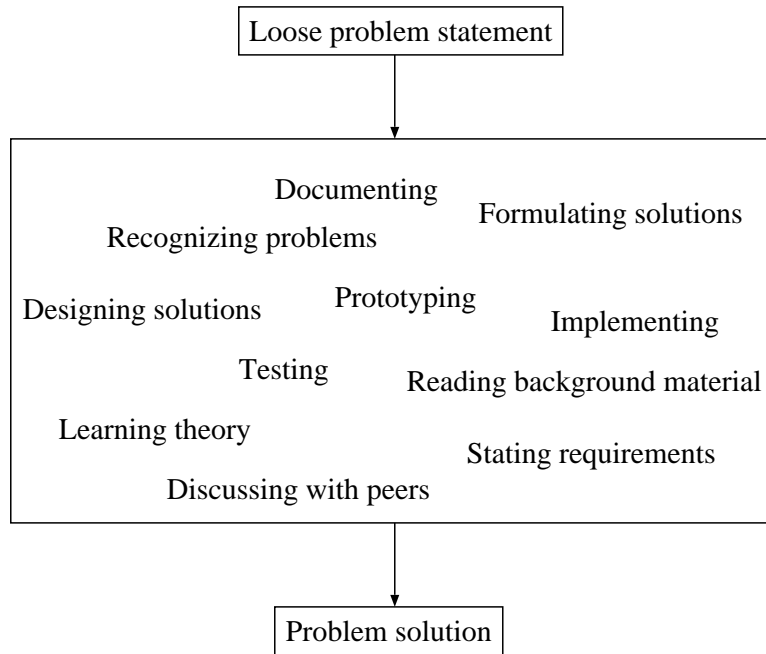


Figure 1.1: Our software development method.

they should be followed. This process is appropriate when the subject matter is well understood, or a detailed set of customer requirements is available. Neither of these two conditions are satisfied in our case. We will be mapping out the problem domain along the way, and the field is in rapid development. This requires a much more fluid development process, which can respond to changes more easily. The development model that we will follow is similar to the *evolutionary* model [35], but we will define the steps and the ordering ourselves.

Our software development method for this work is best described by Figure 1.1. While the process is divided into different steps, they will not be performed in any formally defined order. In fact, they will often be performed in parallel, and every step will feed back information to every other step, which will in turn help improve the final product. The steps are purposely arranged in a chaotic manner, to emphasize that there is no implied ordering. This does not mean that they will be performed in a *random* manner, far from it. It means that the decision on when to move from one step to another will not be made by some external requirement, but rather by the internal state of the project.

This kind of development process is possible due to the relatively small size of the project, the fact that only one person is working on it, and the fact that the overall problem to be solved is only loosely defined in a problem domain with many possible solutions.

1.4 Outline

Chapter 2 gives a detailed description of multihoming as it is currently practiced in the IPv4 Internet, and the progress made towards new solutions for an IPv6 Internet.

Chapter 3 states the requirements for a contribution to the field which will solve the problem of making a connection survive a network failure.

Chapter 4 details the design of two different solutions, one as a user space API and protocol and the other as a kernel based TCP modification.

Chapter 5 explains how the designs were implemented in a FreeBSD/KAME environment and compares the functionality provided by the two solutions.

Chapter 6 configures a suitable network environment and demonstrates the two solutions, through logging information and debug output.

Chapter 7 sets up a hypothesis regarding the performance of selected parts of the implementation, implements a custom tool to measure it and compares the result with the hypothesis.

Chapter 8 summarizes the achievements of the work, gives recommendations and ideas for future work and draws a conclusion on the work.

2 Multihoming technologies

2.1 Multihoming coming into focus

The development of IPv6 has been going on for several years, beginning with the basic foundations of an IPv6 packet header, an addressing architecture, and so on. Lately, multihoming has become a hot topic in IPv6 development circles. In particular, it has been a big item on the agenda at recent meetings and discussions of the IPng¹ group working under the IETF². Several different mechanisms have been suggested for how to handle a multihomed Internet, in this chapter we will present an overview of the most important problems and their proposed solutions.

2.2 Definition of multihoming

A host can be multihomed in two basic ways. The first is with a single network interface, which has been assigned multiple IP addresses, and the second is multiple network interfaces. We will discuss each instance in order. Most references to multihoming in this work applies to either type, exceptions will be noted.

2.2.1 One network interface, multiple addresses

The functionality to add multiple addresses to one network interface was introduced with 4.3BSD Reno (Net/2) and was originally intended to allow a smoother transition in the case of address changes of servers [41, p.66]. The server would continue to accept packets for the old address, giving clients a chance to learn the new address in a grace period decided by the system administrator. Recently, this has been exploited by World Wide Web (WWW) servers in particular to create “virtual servers”. With the Internet becoming more commercial and the increased focus on the WWW in particular, having your own domain name and associated WWW hostname has become much more important than previously. In many cases the load generated from a single WWW site is much less than what a modern computer can handle, and it makes sense to allow a single server to handle multiple domains. Unfortunately, version 1.0

¹IP Next Generation, the working name for IPv6.

²The Internet Engineering Task Force, the organization which sets and publishes Internet standards.

of the Hyper Text Transfer Protocol (HTTP) [16] did not allow this to be accomplished without assigning multiple IP addresses to a server, and having each virtual WWW server bind to a different IP address. HTTP version 1.1 remedied this with a “Host:” HTTP request header that allows a WWW server binding to a single IP to demultiplex connections to the appropriate virtual server. The practice is still prevalent though, due to implementations of other protocols such as File Transfer Protocol (FTP) [33] being much slower to adopt virtual server directives.

This particular type of multihoming is not of interest for our research into multihoming, since it has been faded out by the advent of HTTP 1.1, and the fact that it provides very limited opportunity for taking advantage of multihoming benefits. For example, it may be completely undesirable to attempt to preserve connections when an address is removed from an interface, since it would appear to come from a completely different server than the one the client thinks it is connected to.

IPv6 introduces the concept of scoped addresses, which means that link- and site local addresses are assigned to the same interfaces as global addresses. In both IPv4 and IPv6 hosts may also be connected to different providers through the same physical cable if more than one “prefix” is routed on the same link. This, of course, creates a single point of failure and as such is questionable from a fault tolerance point of view, but may be an advantage in terms of cost.

In general, the scenario of having one network interface with multiple addresses has not been the norm in the IPv4 Internet, but will be perfectly ordinary in the coming IPv6 Internet. Even if the additional addresses are not globally routable, it still creates a multihoming scenario where issues such as source address selection takes on new importance.

2.2.2 Multiple network interfaces

Although the bandwidth of network technologies in general has been rising faster than the speed of the computers they interconnect [37, p.569], increased network bandwidth is a common reason for equipping a computer with multiple network interfaces. The reason is cost, two low bandwidth connections are often cheaper than one high bandwidth.

In some cases, like modem or ISDN B-channel bundling, all (de)multiplexing happens at the link layer, thereby presenting a single link layer to the Internet layer. This avoids any of the issues that we intend to deal with in this work, and we do not consider a host implementing such a solution multihomed.

We will only concern ourselves with cases where a host is configured with multiple IP addresses. This is typical of links which are always up, such as ATM, Ethernet, DSL and cable.

As more and more networks are built, it is becoming more common for computers to serve as routers. A router is by definition multihomed because it copies packets between different networks based on their address. Usually this is a result of having multiple physical network interfaces attached.

2.3 Potential multihoming benefits

The current motivations for exploring multihoming can be roughly divided into five different areas, enumerated below. We will discuss each in turn.

1. Fault-tolerance.
2. Load sharing.
3. Provider selection.
4. Eased renumbering transition.
5. Enhanced mobility support.

2.3.1 Fault-tolerance

Fault-tolerance is probably the most important benefit in the sense that anyone investing in a multihoming solution will at least expect to gain this. The simplest form of fault tolerance consists of using one network connection during normal operation and should the link become unusable, migrate traffic to another. There are many reasons why a link might go down in the Internet, including physical wire cuts, router crashes, power outages and configuration errors. With the Internet becoming more and more important to both businesses and organizations, we expect that more and more people will be willing to invest in redundant Internet connections for purposes of fault tolerancy.

2.3.2 Load sharing

Load sharing is also a very important motivation. Bandwidth demands of Internet applications are increasing rapidly, some examples are WWW, multimedia on demand, IP telephony, file transfer, video conferencing and so forth. Very high bandwidth single-wire solutions like fiber optic cable are available, but often at prohibitly high cost. Multiple low bandwidth connections is a viable alternative which will in many cases create multihoming scenarios. Traffic load distribution can be realized at several levels, from simple random load sharing, to advanced calculated load balancing.

2.3.3 Provider selection

The third benefit, provider selection, is already present in the IPv4 Internet of today, but is expected to become even more of an issue in the IPv6 Internet of tomorrow. It is fairly common today for Internet users to have multiple dial-up Internet Service Providers (ISPs) configured and alternate between them. Particularly after the concept of free or semi-free ISPs came into existence. However, in these cases users tend to only use one ISP at a time, thus not creating a

real multihoming scenario. With connection prices constantly dropping, this is expected to change, and there is also a trend towards high bandwidth, always online connections such as DSL and cable Internet. In addition to expecting fault tolerance and load sharing, users may wish to alternate between different Internet links depending on such factors as time of day and current traffic load, in order to optimize the cost vs quality of service equation.

2.3.4 Eased renumbering transition

The fourth benefit is particularly interesting because of the new mechanisms for renumbering networks introduced with IPv6. Renumbering an IPv4 network is usually a major chore, involving work on every individual host. The process can be made easier if solutions such as Dynamic Host Configuration Protocol (DHCP) [12] are deployed, but it is never going to be as easy as it is with IPv6. During a network renumbering, hosts will have multiple addresses, creating a multihoming scenario where benefits such as migrating transport layer connections transparently would be very welcome.

2.3.5 Enhanced mobility support

Mobile IP networks almost invariably create multihoming scenarios. Both in the case of moving between heterogeneous networks such as Ethernet to wireless, or when moving from one wireless Local Area Network (LAN) to another. Often such a change involves an overlap period where the mobile host is attached to multiple networks with different addresses. Mobile IP is one the areas where the most growth is expected in the coming years, as more and more people acquire portable computers and Personal Digital Assistants (PDAs). Naturally, a lot of research effort is therefore going into this area, and better multihoming solutions are an important part of this.

2.4 Problems associated with multihoming

The benefits described in Section 2.3 do not come for free. Each benefit raises unique challenges which we will give an overview of in this section.

A common trait among the problems is that many of them can be solved at different levels, from the application to the Application Programming Interface (API) and down to the TCP/IP stack and the protocols it implements. Some problems can be solved both at the routing level and at the end-to-end host level, or a combination of these two. This creates a rich field of solutions which will hopefully allow the best ones to be picked.

Routing is one of the most discussed areas because it is fundamental to other benefits, and one where some lessons have been learned from the experiences with IPv4. The fear is that a profusion of multihomed sites will cause large numbers of routing table entries to be created in the backbone of the Internet,

thereby making it slower and more expensive to run. If a site is multihomed to a single ISP, there is no problem, the addresses will all be within the same block of aggregatable addresses, and the only extra routing done is entirely confined to the ISP. This solution is unlikely to satisfy sites which are really serious about fault tolerance, because after all, the ISP becomes a single point of failure. A big problem is therefore how to route multiple addresses reliably without causing a routing table “explosion” in the backbone.

A desirable property of a fault tolerant solution is to have transport layer connections survive a failure completely transparently. There are several ways to solve this problem, either by having routers continue to keep the same address routable after a link failure, or by modifying APIs and protocols to handle multiple addresses. Most current APIs such as the socket API, and connection oriented protocols such as the Transport Control Protocol (TCP) [31] are designed for a 1-to-1 address relationship which does not suit multihoming scenarios well.

Multihoming solutions may have to deal with challenges such as ingress filtering [15]. This is a mechanism recently introduced to the Internet because of problems with Denial of Service attacks with forged source addresses, preventing tracing and escalating damage. Ingress filtering is performed by the ISP on packets arriving from a customer. Any packets bearing a source address outside the range advertised for the link will be filtered out, effectively preventing any packets with a forged source address from being sent. Multihoming represents one scenario where packets might legitimately be sent with a source address from a different link. Because of the widespread use of ingress filtering today, multihoming solutions should avoid using this technique. It would be cumbersome and expensive for both ISPs and customers to require ingress filtering to be disabled on links to a multihomed site.

Load sharing and provider selection both depend on the same subject, namely address selection. This problem can be divided into source and destination address selection, but the two are closely related. A third member should be added to this set; interface selection. For a single connection, many different permutations of this set may provide working communication, but which is the desired one? The answer is dependent on the bits of the destination address, the time of day, the current level of traffic and other factors, and therefore difficult to answer. This is a good example of a problem which can be solved both by the choice of applications, APIs or even lower levels such as the TCP/IP protocol stack.

All the problems listed in this section are engineering challenges which will no doubt occupy Internet architecture engineers for a long time. Already, many solutions have been proposed and more are sure to follow. It is important to remember that there probably will not be one catch-all solution, rather there will be alternatives, suited to different budgets and architectures.

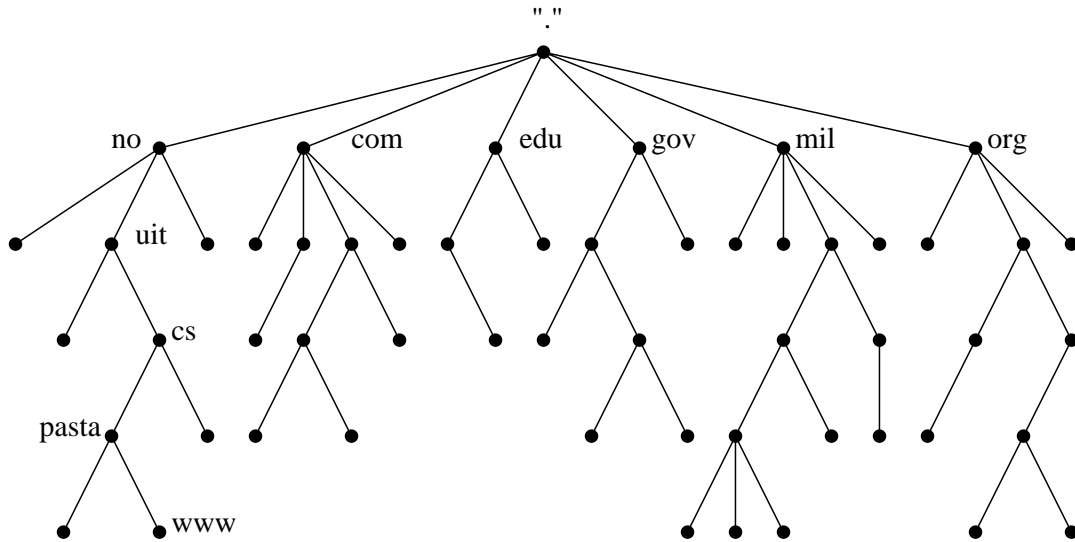


Figure 2.1: The hierarchical DNS name space.

2.5 Multihoming and DNS

A discussion of multihoming in the Internet could never be complete without a section dedicated to the Domain Name System (DNS) [29]. DNS is to be credited with a large part of the success that the Internet has today. Soon after the birth of the Internet, the developers realized that remembering numeric IP addresses up to 12 digits in length, was not something human beings were cut out to do, we prefer names. The first solution to this problem was a simple ASCII text file hosted on each machine on the Internet, and regularly updated through a file transfer service like FTP. This worked well for a few years, until the text file grew to many megabytes in size, and the task of administrating changes to it became a major chore. The solution was a hierarchical, distributed database known as DNS.

As shown in Figure 2.1, the DNS hierarchy is a tree growing down from a common root. The first level divides names into top level domains, based on institution or country. Below this, further divisions are made into sub domains, until reaching the leaf nodes which represent hosts. Different DNS servers are given control over different parts of the tree, referred to as their zone of authority. This means that there is no central authority to rely on, creating a more fault tolerant system. The naive approach to looking up a name would be to start at the root of tree by contacting one of the root servers, and working your way down the tree, querying the server whose address you learned from the previous lookup. It would be very inefficient if all applications went through this procedure and it would generate an enormous load on the root servers. Therefore DNS was designed such that local DNS servers can cache information for configurable periods of time, reducing both network traffic and server load.

DNS stores *records* for names, which can contain different kinds of information.

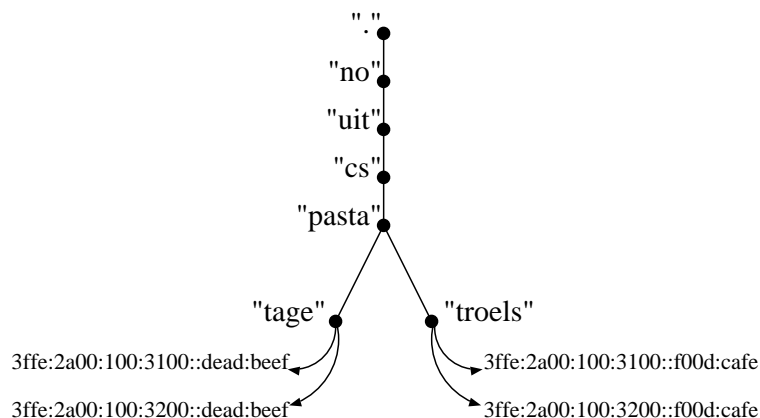


Figure 2.2: Example DNS tree down to host level, with associated AAAA Records.

The most popular record by far, is the A or Address Record. This maps a hostname to a single IPv4 address. A new record has been defined for IPv6 addresses, the AAAA Record [39]. While both of these records only contain a single hostname-to-address mapping, nothing prevents the creation of multiple such records to show the world multiple IP addresses for a single host. Figure 2.2 shows an example of two hosts with two IPv6 AAAA records each. Reverse lookups are also possible, by using the IP address to create a hierarchy of byte sized address parts, and looking up PTR Records associated with the IP address.

The AAAA Record is in the process of being phased out in preference of a new record called A6. The A6 Record is designed to take advantage of the hierarchical nature of IPv6 addresses under the current allocation rules by storing partial addresses and the DNS name that should be queried to get the next level of the address (if any). Forming complete IPv6 addresses therefore involves following variable length chains of A6 Records. This scheme provides better support for multihomed hosts, with less duplicate information needing to be maintained. Multiple addresses for a DNS name are formed whenever a query somewhere along the A6 Record chain returns multiple records. This creates a tree of addresses, which can split at any level, without duplicating the identical portions of the address. A draft is available with the specification of the A6 Record [7] which also relies on two additional new DNS features; DNAME Records [5] and binary labels [6].

There are two reasons why one would map a DNS names to a multiple IP addresses. The first, and probably most common reason today, is to replicate a service among multiple machines, to spread the CPU load. The second reason is the multihomed scenario where one machine has multiple addresses. If the addresses represent different physical networks, it can be used as a technique to spread the network load. An example of a DNS name mapping to multiple IP addresses is *1.fes.mirabilis.com* (a server for the highly popular ICQ protocol) which has 10 A Records. It is unknown whether these addresses represent different machines or network interfaces.

We are only concerned with the case where a host is multihomed, but both reasons represent attempts to do load sharing and create a more fault-tolerant service. The success of these attempts depends largely on the *client*, it must try to connect to all addresses returned in the DNS lookup. Unfortunately, even though all socket APIs today return a list of addresses in response to a DNS name lookup, many applications still only attempt a connection to the *first* address returned. This may be due to ignorance on the part of programmers, lack of documentation, historical reasons or sheer laziness. The observant reader will note that the DNS server has a chance to influence the choice though, by changing the order that addresses are returned in. In some cases, this solution can be defeated by resolver or application name caching, if the caches are not implemented properly.

Algorithms for address rotation are another subject of discussion. Arguments can be made for both round robin, random and sorting of addresses based on the address of the host doing the lookup. The latter solution was previously implemented by the de facto standard UNIX DNS server, BIND³, but was removed in later versions. The BIND authors felt that address sorting would more appropriately be done in the resolver, and consequently the resolver shipping with BIND 8 supports `resolv.conf` directives to sort addresses according to network [1, p.240]. For example, a `resolv.conf` directive of “`sortlist 128.32.42.0/255.255.255.0`” combined with a DNS lookup reply containing the two addresses 128.32.1.1 and 128.32.42.1 would prompt the resolver library to exchange the two addresses before handing them over to the application.

If this client side load sharing is desired in an environment where addresses change frequently, updates must be made to the DNS database. This has become easier recently with standards such as “Dynamic Updates in the Domain Name System” [40], but because of the nature of DNS it involves a tradeoff. If the lifetime of DNS address information is set low, updates will propagate quicker, but at the cost of higher network traffic. Vice versa, the lifetime can be set high, to lower traffic, but this increases the time for updates to propagate.

2.6 Multihoming with IPv4

It is only natural to begin a comparison of multihoming technologies with a description of current practice in IPv4 environments. The very least we should expect from IPv6 techniques is that they are equivalent to current IPv4 practice, hopefully taking the opportunity to improve where necessary.

2.6.1 Router based solutions

In the original Internet, addresses were routed according to class; A, B or C [32]. Because of exponential routing table growth in the backbone “default-free” zone, and the fact that class B networks were being rapidly used up, a new kind

³The Berkeley Internet Name Daemon, see <http://www.isc.org/products/BIND>.

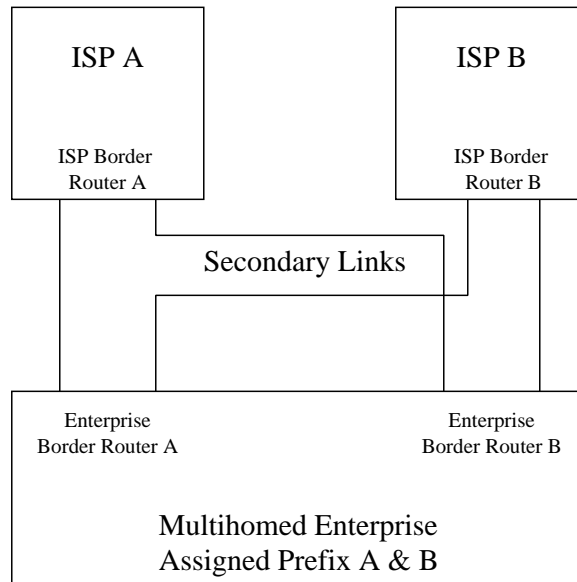


Figure 2.3: Illustration of RFC 2260: Scalable Support for Multi-homed Multi-provider Connectivity

of routing scheme had to be introduced. The Classless Inter-Domain Routing (CIDR) [34] scheme solved the problem by collapsing class C networks into larger aggregatable blocks. Unfortunately, CIDR cannot be used for sites multihomed to providers with different address blocks, and specific routes must be advertised into the default-free zone. Because the number of multihomed sites is much smaller than the number of singlehomed sites, the problem has not become big enough to warrant the same kind of attention that the introduction of CIDR got.

This is not to say that it has been completely overlooked though. An RFC⁴ exists which describes “Scalable Support for Multi-homed Multi-provider Connectivity” [2]. The basic idea is to establish a secondary link (usually in the form of an IP-over-IP tunnel) for each primary physical link. The secondary link must use a different physical link than the primary link. If the primary link fails, connectivity will be maintained through the secondary link. This is accomplished by advertising routes through both primary and secondary link, the latter with lower priority. The scheme is illustrated in Figure 2.3.

It is unknown how widely implemented this solution is, but the impression gathered from the discussions of the IPng Working Group is that it is very sparsely deployed and generally considered complex and difficult to manage.

2.6.2 Host based solutions

Few host based solutions to improve IPv4 multihoming exist. The most basic of all host operations to support multihoming is to try all addresses returned by

⁴Request For Comments, the document format for Internet standards.

DNS lookup. The ratio of applications supporting this is unknown in general. One of the current IPv6 multihoming drafts mentions in passing that “in BSD 4.4 derived Unixes we have found only one standard application trying only the first returned address” [13].

The `connect` operation in the traditional BSD socket API takes a single IP address as argument to specify which host to connect to. Most other TCP/IP APIs have copied this behavior, but there are some APIs available which support connecting to a DNS name, allowing transparent support for trying all IP addresses. Examples are the MacOS⁵ network API and the `java.net` Java standard library.

2.7 Multihoming with IPv6

Most of the current IPng Working Group proposals concern routing solutions to avoid the routing table explosion problem discussed in Section 2.4. There are also relevant Mobile IPv6 proposals, and some work on connection preservation and APIs. This section will give overview of the current state of affairs. Much of the information presented is summarized from minutes of the three most recent IPng Working Group meetings in July [28], September [19] and November [20] 1999, as well as discussions on the IPng mailing list⁶. The reason that we have to use these sources is of course that IPv6 is still in rapid development, especially in this area.

2.7.1 Mobile IPv6

The Mobile IPv6 specification is currently available in the ninth draft [26], and seems to be almost completely ironed out. It is interesting in relation to multihoming because it encounters and solves some of the same problems. Mobile IPv6 uses a concept of a *home* address to give a mobile node a permanent address no matter which *care-of* address it is currently using. Traffic from a correspondent to a mobile node is initially sent to the home address, where a *home agent* picks it up and tunnels it to the care-of address. The mobile node replies from its care-of address, but includes a header option specifying the home address. The correspondent uses this home address to find the binding that the reply belongs to, and delivers the data to the application. The application will never notice that the mobile node is actually communicating from the care-of address rather than the home address.

We can see that this is in fact a multihomed scenario where the mobile node has at least two addresses and solves the problem of connection preservation with a home agent and a header option in packets.

⁵The standard operating system for Apple Macintosh computers.

⁶Subscription details and archives are available at <http://playground.sun.com/pub/ipng/html/instructions.html>.

Mobile IPv6 also introduces *binding update* Internet Control Message Protocol Version 6 (ICMPv6) [4] packets to make a correspondent send traffic directly to the care-of address, instead of having it tunneled through the home address. This can potentially be used to have the correspondent of any multihomed node redirect traffic through another, working link, thereby solving the fault tolerance problem as well. Mobile IPv6 does not inject any additional routes into the network, so it also avoids any routing problems.

In summary, Mobile IPv6 solves, or has mechanisms which can potentially be used to solve, many of the problems identified with multihoming. There are several reasons why the IPng Working Group is pursuing other options as well. First of all, Mobile IPv6 is not completely specified yet, there are still some security related issues to work out. Second, it would be unfortunate to place all eggs in the proverbial basket; if Mobile IPv6 is not deployed, it would hinder multihoming deployment as well. Furthermore, Mobile IPv6 puts a fair number of requirements on both routers and hosts, it may be interesting to explore options which only depend on *either* router or host changes.

2.7.2 Router based solutions

The IPv6 address format [21] is designed to promote the use of aggregation for routing. This is a lesson learned from the IPv4 Internet, non-aggregated addresses proved to cause exponential growth of routing tables, beyond what the routers were able to handle. Nobody knows how many IPv6 sites will want to multihome in the future. If this number is low, or routing technology improves sufficiently, the IPv4 “solution” of using non-aggregated addresses for such sites could work. The IPng Working Group expects that the number will rise significantly and is therefore working on multihoming solutions under the assumption of aggregated addresses. Some of these solutions are pure routing schemes with no host changes required, and these are presented in this section.

The first router solution we will discuss is based on a previously published suggestion [2], updated for IPv6. It is available in draft format [24] and has been discussed at several IPng Working Group meetings. The core idea of the proposal is unchanged from the RFC already described in Section 2.6.1, and will therefore not be repeated here.

The solution avoids injecting any additional routes into the backbone, but has quite complex configuration and requires a good deal of ISP cooperation and effort. The only benefit it delivers is tolerance against ISP link failure, it makes no attempt to do any kind of load sharing across the links.

The complexity of the first approach inspired the second approach which we will describe. This too is available in draft form [42] and is being discussed. The idea is that addresses of Prefix A are obtained from a single, primary ISP A, while links are maintained to any number of alternative ISPs. In the stable situation with all links working, outbound traffic from the site will flow through any of the links, depending on configuration. Inbound traffic will always use the path through the primary ISP. Should the link to the primary ISP fail, inbound

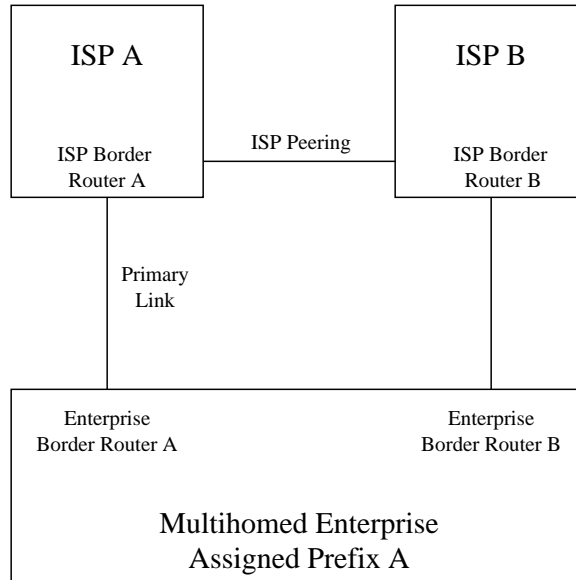


Figure 2.4: Simple multihoming routing, assuming direct ISP peering.

traffic will be routed from the primary ISP to an alternate ISP and from there to the site. See Figure 2.4 for an illustration of the configuration.

Since only one address prefix, the one that belongs to the primary ISP, is advertised, there is no increase in backbone routing table size. A limited form of load sharing can be accomplished by sending outbound traffic through any of the alternate links. Inbound traffic will always use the same route though. For efficient operation, the primary and alternate ISPs must be directly peered. The proposal is definitely simpler to configure and use than the first one, particularly because of the lack of tunnels. Unfortunately, the primary ISP becomes a single point of failure for the site which may be unacceptable to customers of multihoming solutions.

A variation on the second scheme is presented in another draft [13], under the name “Mutual Backup”. In this case, two ISPs which both have a link to a site, advertise routes towards each other such that the direct link will be preferred, but traffic will go through the other ISP if one ISP’s link fails. The proposal assumes a willingness to cooperate between the two ISPs, and it does not expand easily to more than two ISPs.

The draft contains more proposals, one of which is labeled “Broken Path”. This proposal assumes new connections to the multihomed node will succeed by trying all addresses stored in DNS, and suggests that the border router handling a broken link should deprecate addresses associated with this link using the router renumbering protocol. This will stop hosts in the multihomed site from using that address for new connections. Existing connections will be migrated using the Mobile IPv6 binding update mechanism and home address destination options header. If we tolerate losing existing connections when a link goes down, the proposal is a pure routing scheme, assuming no additional mechanisms and

requiring almost no implementation changes.

2.7.3 Host based solutions

Both router and host based multihoming solutions are being pursued in parallel. Many of the router solutions are limited to how well they deal with different kinds of failure, several only handle the case where the actual link is down, and not other cases like routers going down. This is an important limitation, and host based end-to-end solutions have the potential to do better, by completely ignoring the reason for failure, as long as one viable route is available.

The disadvantage of host based solutions is that they will most likely require support at both ends to function. At least, no one has yet made a proposal which avoids this. Because IPv6 deployment has yet to start in earnest, a host based approach selected at this point has a good chance of being widely implemented.

In this section we will look at the current host based proposals, divided into three categories; address selection, connection preservation and a new API.

Address selection

Selection of both source and destination address for a connection is important, because it can help create better, more robust routes, for example by reducing the likelihood of asymmetric routes. This is one of the areas that the IPng Working Group has prioritized for the near term, and the draft [10] which is available is probably the most mature of the several multihoming related drafts published lately. The draft specifies two different algorithms, for the two cases of address selection. A list of potential destination addresses is ordered without any additional information, while a source address is selected based on the destination address. This reflects the common Internet application model of looking up the addresses for a host and letting the stack select the source address. Choices made by the algorithms can be overridden by applications through system calls, or by administrators through a policy table.

Preserving transport layer connections

The argument has been made that preserving transport layer connections does not need to be a major goal of multihoming solutions, since mission critical applications should be designed to cope with reconnecting and resuming themselves. It is an interesting problem though, and an attractive benefit for many applications, so work is underway in this area too.

One host based mechanism has already been mentioned in both Section 2.7.1 and 2.7.2, namely Mobile IPv6. Another mechanism has been proposed, and received favorably by the IPng Working Group. A draft is being written, but currently the only documentation consists of material from the September 1999 IPng Working Group meeting, and the author's homepage [38].

The basic idea of the proposal is to exchange a number of alternative prefixes at the beginning of a TCP connection establishment, and, if the connection fails, migrate to one of these. The mechanism is quite simple in idea, but unlike the Mobile IPv6 approach requires modification to TCP. The mechanism creates more state at the hosts than a minimal Mobile IPv6 connection preservation implementation, because unlike Mobile IPv6 it does not add overhead to the packets. It may be possible to use the home address destination option from Mobile IPv6 to reduce the added state at the expense of packet overhead if so desired.

New API

The idea of a new API designed to improve multihoming support has been discussed at several IPng Working Group meetings, but no concrete specification or draft is available yet. The core idea of the proposal is to connect to a DNS name or list of addresses, instead of a single address. This will give the API and the TCP/IP stack greater flexibility in establishing a connection.

The underlying layer will for example be able to change the order of addresses to connect to, or select a source address and interface with more information available than just a single address.

The idea has also been floated that such an API could be used to solve the problem of preserving transport layer connections. One way to do this would be to introduce a new protocol layer on top of the transport protocol. Such a protocol layer would strongly resemble the *session* protocol layer of the ISO OSI network architecture reference model [8] by presenting a persistent connection to the application, even when the underlying transport layer connection changes.

2.8 Summary

We started this chapter by looking at the potential benefits of multihoming and the problems that need to be solved. We proceeded to explain how the problems are addressed in the IPv4 Internet, and what proposals have been made for IPv6 solutions. Multihoming will be commonplace in the IPv6 Internet and it is necessary to develop new solutions to take full advantage of it. We divided the solutions into router and host based categories, but also explained hybrid solutions like Mobile IPv6. We saw that one of the biggest and most interesting challenges is how to keep transport layer connections alive in the face of link failure or address changes.

3 Requirements

Based on the current multihoming technologies described in the previous chapter, we can tell that this a field with many challenging engineering problems to be solved. Based on our problem statement, we will create a host based solution which solves the problem of preserving transport layer connections across link failures. We begin by presenting a system model to give an overview of the problem domain and continue by listing functional and non-functional requirements for the solution.

3.1 System model

The system model for our contribution is best described by referring to Figure 3.1. Assume that Host A is connected to the Internet through two or more links, each with a separate global Internet address. In the illustration, two such links are shown, Link A-1 and A-2. Host A is, according to our definition in Section 2.2, therefore *multihomed*. Assume that Host A has a transport layer TCP connection to Host B. Host B may, or may not be multihomed, in the figure it is shown as singlehomed. At the beginning of our scenario, this connection uses the address assigned to Link A-1 and consequently data is transmitted and received over this link. At some point after the establishment of the connection, Link A-1 fails, and all traffic attempted sent over the link is lost, with or without an error being reported. We want to create a solution which will detect this condition, and migrate the connection to another working link, such as Link A-2.

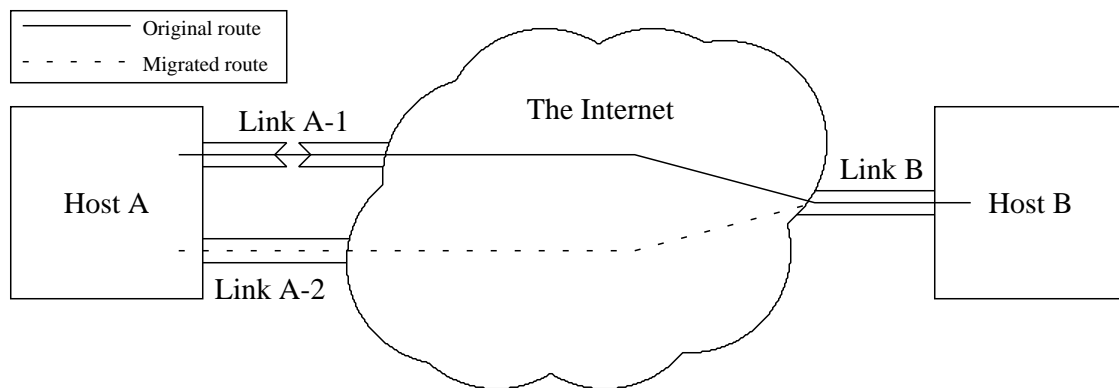


Figure 3.1: The system model.

From this outline, we can tell that our solution will be mostly aimed at providing the first and the last multihoming benefit mentioned in Section 2.3, fault tolerance and smooth renumbering transitions. The figure and the above explanation only dealt with a link failure scenario, but we address renumbering as well, because removing an existing address from an interface (or the interface itself) has a similar effect to the link going down. The other two benefits, load sharing and provider selection, are a bit different and more likely to be provided by other mechanisms which change the conditions at the time a connection is established.

3.2 Functional requirements

As described in Section 3.1, the most important functionality of the solution will be to migrate connections to alternative links. This migration must happen completely transparently, without application participation. This will allow the solution to benefit the large base of existing applications, without any changes.

When making an initial connection to a remote host, the solution must cycle through and try all the addresses found in DNS. Without this requirement, a failed link could prevent a host from making any connection at all.

3.3 Non-functional requirements

The contribution should be completely host based, and expect no router interaction. This is important because it makes implementation cheaper and deployment easier. Routing solutions like the ones presented in Section 2.7.2 usually require ISP cooperation and configuration, which they are sure to charge customers for. The solution should not require any additional routes to be injected into the backbone of the Internet, each link should simply be numbered out of the aggregatable address block of the ISP that it connects to.

The solution should build on the assumption that all alternative addresses for a host are stored in DNS. Without this assumption, it would be very hard, if not impossible, to create a purely host based solution.

The contribution should add a minimum of additional overhead compared to existing systems. Overhead should here be interpreted in the widest sense to include CPU cycles, memory and other storage, plus network bandwidth and latency. It is particularly important to keep the overhead as low as possible when the implementation is not in active use because this will be the case most of the time. Necessary overhead can for example better be tolerated when connections are in the process of being migrated as opposed to when they are stable.

Backwards compatibility and interoperability should be kept as high as possible. A solution which is backwards compatible (in the sense that it can easily be integrated into existing systems) is likely to be deployed quicker than one which

is not. A solution which does not interoperate with existing systems is not likely to be deployed at all.

The development platform for the solution should be a BSD derivative with the KAME¹ IPv6 extensions. There are several reasons for requiring the use of a BSD platform. Most importantly, the TCP/IP protocol stack was first implemented on BSD, and it is therefore in many ways considered the reference implementation. Many other TCP/IP implementations derive from BSD, either as a direct port, or by copying APIs. By sticking to BSD, portability will be as high as possible, which is another requirement for quick deployment.

A multihomed host may have many different links to the network. The solution should make no assumptions regarding the number of such links. However, it must be flexible in the way it selects alternative links for connections, to allow for experimentation.

Security will not be considered a requirement for the solution. Security is such a large and complex subject that we cannot make a complete analysis in the time frame of this work. The primary evaluation criteria will be functionality, with performance given second priority.

¹KAME is the de facto standard IPv6 kernel for BSD based Operating Systems, developed by a consortium of Japanese companies, see <http://www.kame.net>.

4 Design

In this chapter we present solutions which satisfy the requirements stated in the previous chapter. Here we aim to describe how the requirements are met, without going down to a level of detail where we show the actual code. The implementations will be described in the next chapter.

4.1 Exploring different designs

There are two primary functional requirements which dictate the design process.

1. When making an initial connection to a remote host, all addresses found in DNS must be attempted.
2. If an existing connection breaks due to link failure or address changes, migrate it transparently to another link.

The first requirement will be met with a change at the API level. The `connect` function of the original BSD socket API is unsuited for multihoming scenarios because it identifies the remote host by a single IP address. A replacement will be created which instead uses a DNS name as identifier. This will allow the API to internally try all available addresses for the remote host, greatly increasing the chance of making a successful connection in a multihomed scenario.

The second requirement can be met in at least two different ways. Within the context of the API, a new *session* protocol layer can be introduced, which handles the task of changing the underlying TCP connection without notifying the application. Alternatively, the requirement can be met by modifying the TCP protocol itself, to make it aware of multiple peer addresses. This will also allow connections to be migrated transparently.

Both of the two different solutions will be pursued in this work. This will give us plenty of opportunity to make comparisons about how well the two meet both functional and non-functional requirements. The API and the protocol layer it introduces will be described first, followed by the kernel TCP modification.

4.2 The namesocket user space API and protocol

The design of namesocket is, for better and worse, modeled closely after the original socket API introduced in the BSD operating system. The advantage is

that programmers who are familiar with the socket API can immediately use the namesocket API as well. The disadvantage is that it inherits some problems, such as the awkward `select` operation.

The name of the API was chosen because the applications which use it deal solely with DNS names. There is no access to the underlying IP addresses involved. This abstraction is, as we shall see, exactly what allows an implementation of the API to resume a connection which has failed.

Another convenient result of the abstraction away from IP addresses is that it shields users against the differences between IPv4 and IPv6, such that they can transparently use either or both. It also makes the applications more robust against future protocol changes. The API is general enough to run over any reliable transport mechanism. Using the namesocket API can also help improve portability. The socket API is closely tied to the BSD operating system, and although widely implemented, there are differences between implementations which the library can shield an application from.

Ideally, we could have wished to replace the standard socket API with our own identically named functions. That way, we could have offered the benefits of the namesocket library transparently, without having to adapt applications. There are two levels of such compatibility, executable and source level. Executable level compatibility would mean that existing binaries could be used and benefit from the namesocket features. Source level compatibility would mean a recompilation was required to get applications to take advantage of the new features. The latter would be a smaller obstacle in UNIX environments compared to other platforms, but there are always commercial products where source code isn't available. Both levels are name space problems; how to make sure that our identically named functions are always called. This can be very tricky, especially when replacing standard library and system calls. We avoid this problem completely by creating our own name space of uniquely named functions, losing both levels of compatibility. Applications will need to be partially reprogrammed to use the library. Since we only expect to use the library with our own experimental programs, this does not represent a problem.

Because the input to the namesocket `connect` function is a DNS name, it will be able to implement and experiment with both ordering of destination addresses returned from DNS lookup and selection of source addresses to match. As explained in Section 2.7.3, careful address selection can result in better performance in multihomed scenarios.

This solution will reside entirely in user space, and not require any changes to the operating system kernel to function. This is a fundamental difference because it creates some challenging problems to solve, and offers some attractive benefits. On one hand, it will lack direct access to much of the state information about a connection. On the other hand, it should be much more portable between different operating systems.

The API is designed to be a prototype which will allow experimentation with reconnecting and resuming connections between multihomed hosts. As such, issues such as security have not been considered at all.

Function name	Inputs	Returns
nsocket	flags	nsd or error
nlisten	nsd, service, backlog	error
naccept	nsd	nsd or error
nselect	read, write and exception nsd's, timeout	# ready nsd's or error
nconnect	nsd, hostname, service	error
nsend	nsd, data, data length, flags	bytes sent or error
nrecv	nsd, buffer, buffer size, flags	bytes received or error
nclose	nsd	error
nshutdown	nsd, direction	error

Table 4.1: Summary of namesocket API functions, inputs and return values.

4.2.1 Functions in the API

An application wishing to use the namesocket API will call upon it through the use of the functions shown in table 4.1. The “nsd” mentioned as both an input and return value is a namesocket descriptor, which identifies a single namesocket. Each function will be described in turn in this section, specifying assumptions and effects.

nsocket

This function creates and returns a new namesocket, initialized with the flags given. An error will be returned if the namesocket could not be created, perhaps because of lack of resources.

nlisten

After creating a namesocket, servers call this function on it. The function will turn the namesocket into a server namesocket, listening on the port of the service given. The maximum backlog of pending connections will be set to the value passed.

naccept

Calling this function on a server socket which has a pending connection, has the effect of accepting the connection and creating a new namesocket for it. The new namesocket identifier will be returned. If no connection is pending, the function will block, waiting for one. A server will usually want to service existing clients while still waiting for new ones, and nselect is provided for that purpose.

nselect

If an application uses multiple namesockets and wishes to avoid blocking while waiting for a condition of one, it must use the **nselect** operation. It takes three arrays of namesockets as inputs, one for each of the possible conditions read-ready, write-ready and exceptional condition pending. It will return when at least one of the namesockets has experienced a relevant condition, or the timeout specified expires. On return, the namesockets which have a condition pending will be marked in the arrays passed in, so these arrays are used for both input and output.

The read-ready condition is also used to indicate two other conditions. A name-socket which has been put into listening mode with **nlisten** will be marked read-ready when a new connection is pending. When the peer closes a connected namesocket of any type, it will be marked read-ready, but return a count of 0 bytes when **nrecv** is called on it.

nconnect

A client will call this function on a newly created namesocket to connect to the specified host and service. The function blocks until the connection has completed, or an error has occurred.

nsend

This function attempts to send the given data on the namesocket. The flags parameter passed is used without modification in the underlying socket API **send()** call. The function may block if the namesocket is not ready for writing, use **nselect** to multiplex writing between different namesockets. The function returns the actual number of bytes transmitted.

nrecv

Similarly to **nsend()**, this function receives a number of bytes into the given buffer, up to the maximum specified. The flags parameter is also passed straight through to **recv()**. The function blocks if no data is available to be read, use **nselect()** to multiplex reading. The return value is the number of bytes written into the buffer. 0 will be returned if the peer has closed the connection.

nclose

This function simply closes the namesocket and any associated connection, and deallocates all resources allocated for it. After calling it, the namesocket is no longer a valid input for any other API functions.

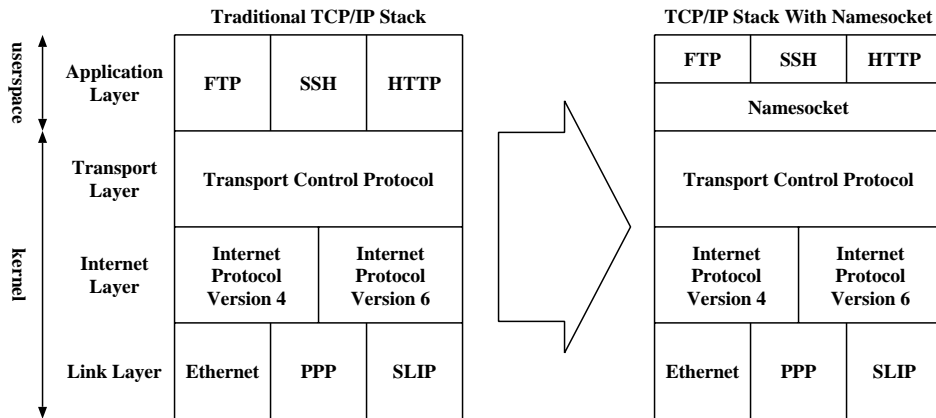


Figure 4.1: Fitting the namesocket protocol into the TCP/IP protocol stack.

nshutdown

Using this function, a namesocket can be closed down in one direction only, depending on the parameter. Functions to send or receive data will return error if they are called on a namesocket which has been closed down in that direction.

4.2.2 The namesocket communication protocol

In order to offer the desired reconnection and resume properties of the namesocket API, a new protocol layer is injected, between the application and the TCP transport layer, see Figure 4.1. In this section we will describe the operation and internal actions of the namesocket protocol through the finite state machine paradigm commonly used to describe network protocols. From a given starting point, the possible actions for a namesocket are shown as arrows, with descriptions of the actions written next to them. Often, transition from one state to another also causes some action to be taken, such as sending a packet. This too is written next to the arrows.

The state diagram in Figure 4.2 uses a pseudo-state called **CLOSED** as a starting and ending point for the life of a namesocket. By pseudo-state, we mean that there is no actual namesocket allocated, it represents the state before the initial allocation and after the final deallocation. It follows that the only way to move on from the **CLOSED** state is to call `nsocket()`, which gets you to the **CREATED** state. The initial type of socket is simply **NEW**, as indicated in the diagram. Assuming that the application follows standard procedure, the next function called will be either `nconnect()` or `nlisten()`, which sets the type to **CLIENT** or **SERVER** respectively. The states possible for each of these two types of namesockets will be described in separate sections below.

Error handling will not be explicitly addressed, any error conditions not otherwise explained will cause the API to back out of whatever it is doing with an error and let the application call `nclose()` to terminate the namesocket and

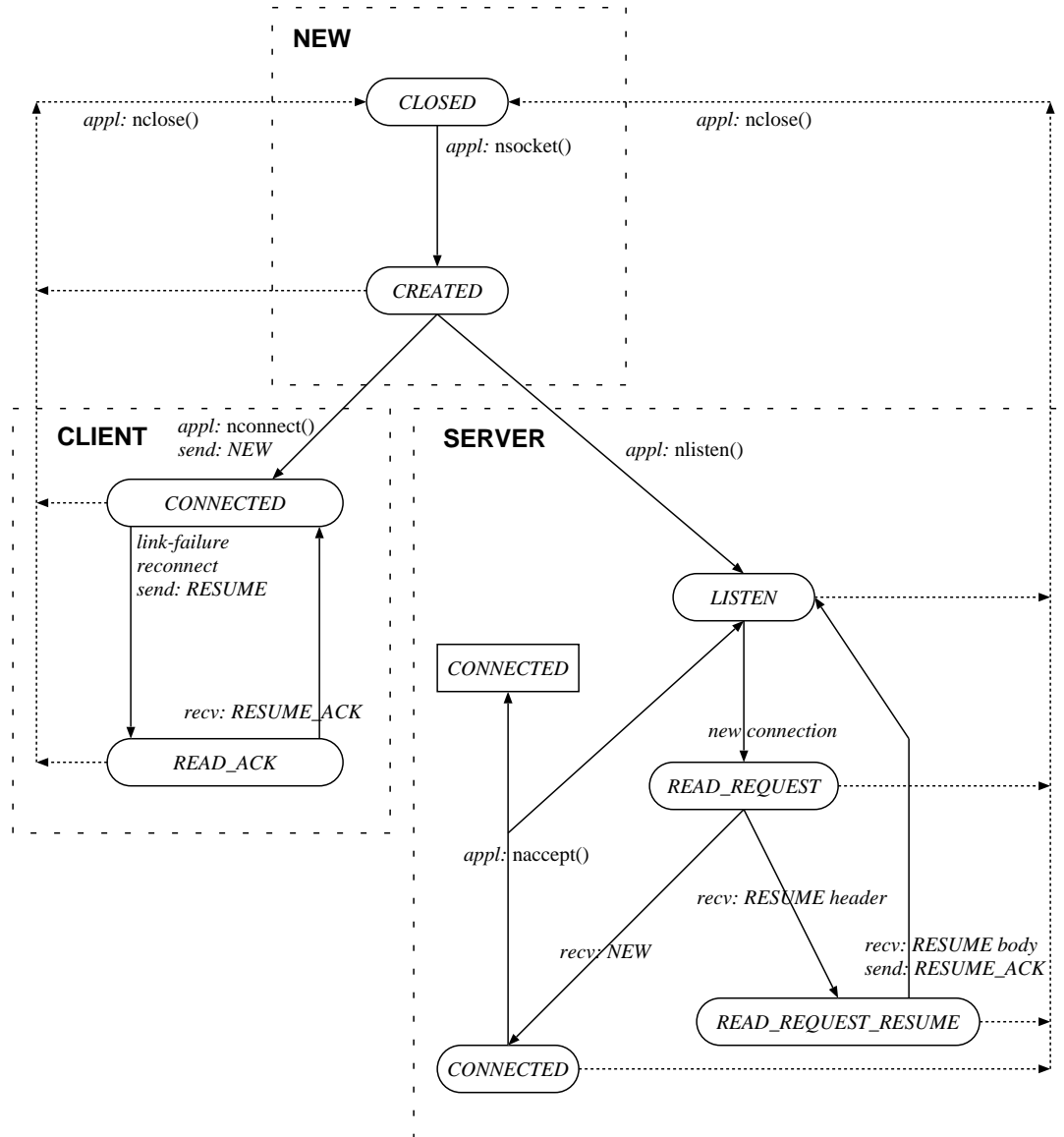


Figure 4.2: The namesocket protocol state diagram.

return to the `CLOSED` state. Of course, the application may call `nclose()` at any time, not only when there is an error, to return to the `CLOSED` state.

Client namesockets

Calling `nconnect()` on a namesocket of type `NEW` initiates a DNS lookup to find the addresses of the server, and then establishment of a standard TCP connection to this address and the service port provided. If the connection is successful, a packet of type `NEW` (not to be confused with the namesocket type `NEW`) is sent off, and the namesocket is changed to type `CLIENT` and state `CONNECTED`.

If a function in the API detects that the connection has been severed, with a cause which is likely to indicate a link failure or a change of address, it will take action to reconnect and resume. The first step is to make a new connection, to the same server port as the original connection was made to. The remote address used may be the same as the one used for the original connection, but greater chance of success is expected with an alternative address, obtained from the list that is returned by a DNS lookup of the remote host's name. If the connection is successful, a `RESUME` packet, as shown in Figure 4.3, is sent off. The namesocket enters the `READ_ACK` state and reads a `RESUME_ACK` packet, also in Figure 4.3, from the socket. If the `data_rcvd` field in the `RESUME_ACK` packet does not indicate a data loss greater than we can cover from our retransmission buffer, we go ahead and retransmit, and finally put the namesocket back in the `CONNECTED` state.

Server namesockets

The `nlisten()` function puts the underlying TCP socket into listening mode on the given service port, changes namesocket type to `SERVER` and moves the state to `LISTEN`. The namesocket does not proceed from this state until a new connection is pending and accepted on the TCP socket. At that point, it moves to the `READ_REQUEST` state, during which a namesocket packet header is read. The type of this packet header decides the next transition. If the type is `NEW`, the namesocket becomes `CONNECTED` and will remain in this state until the application notices and calls `naccept()`. This action causes a new namesocket to be created with state `CONNECTED`, containing the newly connected TCP socket, and the original namesocket is returned to the `LISTEN` state. This is indicated in the state diagram by a *square* state.

If, however, the packet is a `RESUME` packet (see Figure 4.3), we advance to the `READ_REQUEST_RESUME` state and read the rest of the packet. When we have the entire packet, we have to search the list of active namesockets to see if we can find a namesocket which matches the client and server addresses and ports specified. If that is successful, we use the `data_rcvd` field of the `RESUME` packet to calculate the data loss, and see if we have buffered enough data to retransmit everything that is lost. If all is ok, we send a `RESUME_ACK` packet

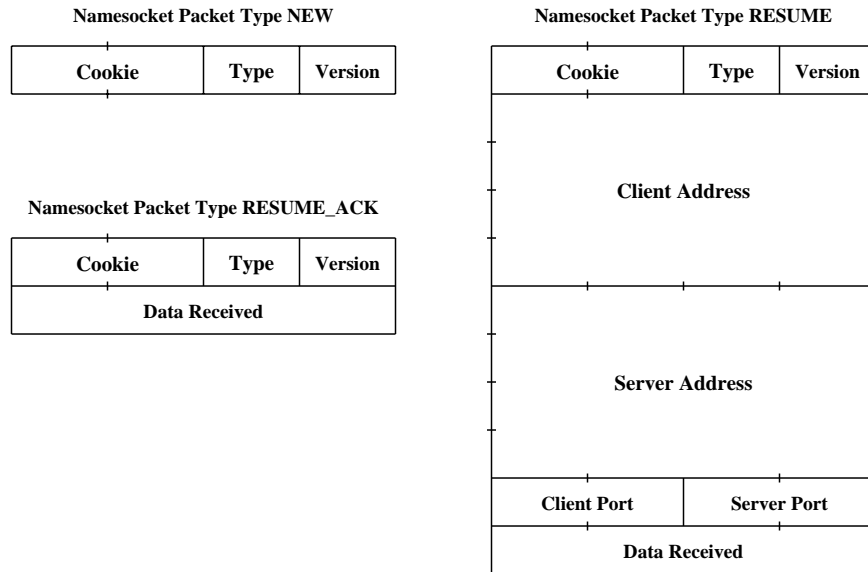


Figure 4.3: The namesocket protocol packet types and layouts.

with our own data counter, retransmit any lost data and update the resumed namesocket to use the newly connected TCP socket. The **SERVER** namesocket which we received the connection on is returned to the **LISTEN** state.

4.2.3 The namesocket packets

The packets exchanged by two namesockets hosts are shown in Figure 4.3. The packets are shown with 32 bits across and vertical notches to indicate 8 bit byte boundaries. All multi-byte fields are in network byte order, i.e. big endian. The layout is designed for optimal alignment on 32 bit CPUs, without adding additional overhead. The two 16 bit port numbers are for instance packed together to avoid ruining alignment for the 128 bit address fields. The protocol is designed to be extensible through a an 8 bit Type field, as well as an 8 bit Version field. The packets specified here will all use Version 1, and an implementation should reject all packets of a version it does not understand. The packets carry a 16 bit “magic cookie” at the beginning, to make it more robust against interpreting data from another protocol as part of its own. This is very much a danger if a namesocket client contacts a non-namesocket server. There is no checksum in the packets, the protocol is only designed for use over a reliable transport such as TCP, so a checksum would be superfluous.

Note that the **RESUME** packet shown is specific to IPv6 because of the 128 bit address fields, while the other two are generic enough to be used with IPv4 as well. This does not lessen the IP layer independence of the protocol as a whole; a new IPv4 **RESUME** packet could simply be defined, with 32 bit address fields.

The field labeled “Data Received” is used by the receiving host to calculate whether it has buffered enough data to retransmit lost data. Because the field

is an unsigned 32 bit integer, it will wrap around after 2^{32} (approximately 4.3 billion) bytes have been sent. The problems arising from this are well studied, because they are similar to what a protocol such as TCP encounters when its sequence numbers wrap around [41, p.810].

4.2.4 Summary

The first half of this chapter has dealt with the design of the namesocket API and protocol which provide an abstraction away from IP addresses, in order to allow connections to be resumed without the application's knowledge. The functions and semantics of the API and the states and packets of the protocol were all presented in order.

4.3 TCP connection preservation at kernel level

This section describes a kernel based scheme to transparently preserve TCP connections between multihomed hosts in the face of link failure or address removal. The design was submitted by Peter R. Tattam of Trumpet Software International Pty Ltd.¹ to the IPng Working Group mailing list in September 1999 as a proposal. The references used for this implementation are the author of the proposal's webpage [38] and minutes of discussions over the proposal from the interim IPng Working Group meeting in Japan, in September 1999 [19]. At the time of writing, the author of the proposal was not aware of any other implementations besides the one presented here, but is working on a draft.

The idea behind the proposal is to expand the current 1-to-1 address relation of a TCP connection to become N-to-M. If a host knows a set of alternative addresses for the peer of a connection, it can change to a different address if the current one cannot be reached.

The proposal suggests one method for exchanging the necessary address information, which is also the one implemented in this thesis. As the author states, there are other ways to exchange address information, this is only one suggested mechanism. The mechanism transmits address information in the IPv6 header of the packets which carry the SYN and ACK of SYN TCP packets. Using the IPv6 header might seem a curious choice, since the information is used by the TCP layer. Unfortunately, the TCP options header is too short to carry the long IPv6 addresses, and is commonly used for other things. The clever design of IPv6 sets no practical restrictions on the amount of headers which can be added (a truth with modifications, see Section 4.3.2), other than the total packet size.

Other than to make the initial connection, this solution will be completely independent of DNS, since the set of addresses used to reroute defunct connections will be negotiated directly with the peer.

¹Makers of the Trumpet TCP/IP stack for Microsoft Windows.

This allows us to change the address of a connection completely transparently, as if nothing had happened. No explicit higher level API action needs to be taken to change the address. This will give the solution a very high degree of backwards compatibility with applications.

This solution will inherently have low portability, because it resides so deep in the operating system. That is the direct result of an almost universally applicable rule of software; the lower you move in a system, the less it looks like any other system, even if they may look the same on the surface.

4.3.1 Fitting into the TCP state machine

To explain how this solution fits into the TCP finite state machine, we must first establish some terminology regarding the protocol. A packet in TCP is usually referred to as a *segment*, and the header of a segment contains various flags. Two of these flags are known as *SYN* and *ACK*, meaning synchronization and acknowledgment, respectively. It is common to refer to a segment with the SYN bit set as a *SYN segment*, or even abbreviate it to just *SYN*.

SYN is only set for the first segment transmitted in a TCP connection. The usual behavior for a host receiving a valid SYN for a listening TCP socket is to respond with a TCP segment with both the SYN and the ACK bit set. The original host responds with an ACK segment and the connection is considered established. This packet exchange is known as the TCP 3-way handshake and the purpose of it is to synchronize the starting sequence numbers.

Our kernel solution ties into the TCP 3-way handshake by adding a PREFIX_SYN header to the initial SYN, and a PREFIX_ACK header to the ACK of SYN. The purpose of piggy-backing on precisely these packets is, according to the author of the proposal, to attempt to keep the security risk roughly equivalent to that imposed by the existing 1-to-1 address scheme.

Figure 4.4 illustrates where the new IPv6 headers will go by showing the TCP state diagram up to the point of connection establishment (the figure is taken from [41]). PREFIX_SYN must be sent everywhere it says “*send: SYN*”, and likewise, PREFIX_ACK must be sent everywhere it says “*send: ACK*”.

4.3.2 The PREFIX_SYN and PREFIX_ACK header options

The headers used to exchange address information do not have to contain complete 128 bit IPv6 addresses, because in the IPv6 address architecture [18] addresses are built up from a (usually 64 bit) prefix and a host identifier part. The host identifier part is designed to be globally unique, and already known to the peer of a TCP connection through the initial address used for the connection. Consequently, it is only necessary to transmit alternative prefixes for the host identifier, which is precisely what the PREFIX_SYN header contains. The PREFIX_ACK header is simply an echo of the PREFIX_SYN header received.

Option	Action	Change	Rest	Hex	Decimal
PREFIX_SYN	00	0	01011	0x0B	11
PREFIX_ACK	00	0	01100	0x0C	12

Table 4.2: Prefix option header values.

It might be argued that one PREFIX_SYN header sent in each direction with alternative prefixes would yield sufficient information to preserve TCP connections. However, the PREFIX_ACK header serves two purposes, both to show that the peer understood the PREFIX_SYN header, and that it agrees on the set of addresses.

The identifiers and header layouts for the prefix headers were not specified in the informal draft available, and were therefore chosen by this author. Thus, compatibility with implementations based on the forthcoming formal draft cannot be expected, but the changes required for compliance should be minimal.

Before presenting the values and encoding, a brief overview of the IPv6 packet header is appropriate. A packet always begins with a 40 byte static header. One of the fields in the header is an 8 bit one called “Next Header”. For a vanilla TCP packet, this field will hold IPPROTO_TCP (6) and the TCP header begins immediately after the IPv6 header. If we fill in this field with another value, such as IPPROTO_DSTOPTS (60), we can insert *extension headers* between the IPv6 and TCP headers. The general structure of an extension header is shown in Figure 4.5. These headers all begin with the same two fields, namely “Next Header” and “Length”. This allows multiple extension headers to be strung together in a daisy chain. The daisy chain can only be followed if the protocol stack knows which IPPROTO_XXX values represent extension headers, and which are upper level protocols such as TCP. Given no other choice, unknown values must be assumed to be upper level protocols and terminate header processing.

This begs the question of how new values for new headers are defined. The answer is that new headers are embedded within extension headers such as IPPROTO_DSTOPTS and given the name *option headers*. Multiple option headers can coexist within a single extension header, due to the way they are encoded. The first byte gives the type of the option, and the second gives the length of the data following it. The type also specifies the action that the IPv6 stack should take if it does not recognize it, and whether it may change in transit. See Figure 4.5 for an illustration of a general option header.

There are two kinds of extension headers currently defined. The *hop by hop* and the *destination* options header. We want our prefix option headers to be carried in the destination options header, because it is of no interest to intermediary routers. The values for the option header types have been chosen based on the types currently registered [25] with IANA², and are summarized in Table 4.2.

Both Action bits being 0 imply that stacks should ignore the the option if they do not know it, and the 0 Change bit means the option does not change in

²Internet Assigned Numbers Authority, see <http://www.iana.org>.

transit.

The layout of the prefix option headers was chosen to encode prefixes as in as little space as possible, at the cost of impractical alignment for 32 and 64 bit CPUs. However, the performance impact of the misalignment should be negligible, given that the options will only be processed when establishing a TCP connection.

The prefixes are placed consecutively after the option header, each preceded by a byte giving the length in bits. Prefixes are theoretically from 0 to 128 bits long, although either extreme makes no practical sense. The length of a prefix could therefore be encoded in a 7 bit value, but that would be taking header length concerns rather overboard. Since prefixes are not guaranteed to be an integral number of bytes long, we round up the number of bits to the nearest byte integral, to at least keep byte alignment in the header. This means for example a prefix of 30 bits would be sent as 4 bytes, with the last two pad bits taken from the host identifier. The choice of padding is important, because it allows the host processing the prefix to avoid manipulating bits to create a complete alternative address. Even when padding is used, the length field should carry the true bit length of the prefix.

The total length of an option header is limited by the 8 bit length field which specifies the length of the data part in bytes, thus giving a maximum size of 255 bytes. This is enough to store for example 28 64 bit prefixes, which is expected to be sufficient for all practical purposes. Theoretically, multiple prefix header options could be placed in the destination header, because the maximum length of that is no less than 2048 bytes. That will not be allowed in this design though, prefix header options following the initial one should be ignored.

Figure 4.5 shows an example of a PREFIX_SYN header, embedded within a destination header. The header encodes two 64 bit prefixes, and is followed by a PadN option which fills the last two bytes. The padding is required because extension headers must be an integral number of 64 bit units long, in this case 3.

4.3.3 Demultiplexing IP packets for N-to-M connections

Figure 4.6 is an attempt to illustrate the way the number of address combinations for a connection rises from just one, to $N \times M$, when we introduce the prefix exchange scheme to TCP. The author of the proposal suggests a barrel metaphor to visualize the situation. Each end of the connection is a barrel with a hole for each address it has. The connection flows through two holes, and should it dry up, we rotate the barrels at either end in an attempt to get the packets flowing again.

Each end of the connection must therefore accept packets for any of the $N \times M$ possible address permutations. This quickly becomes a big number if either end has many addresses. A scheme which is both CPU and memory efficient must be developed to demultiplex the incoming packets. Possibly, hosts should impose

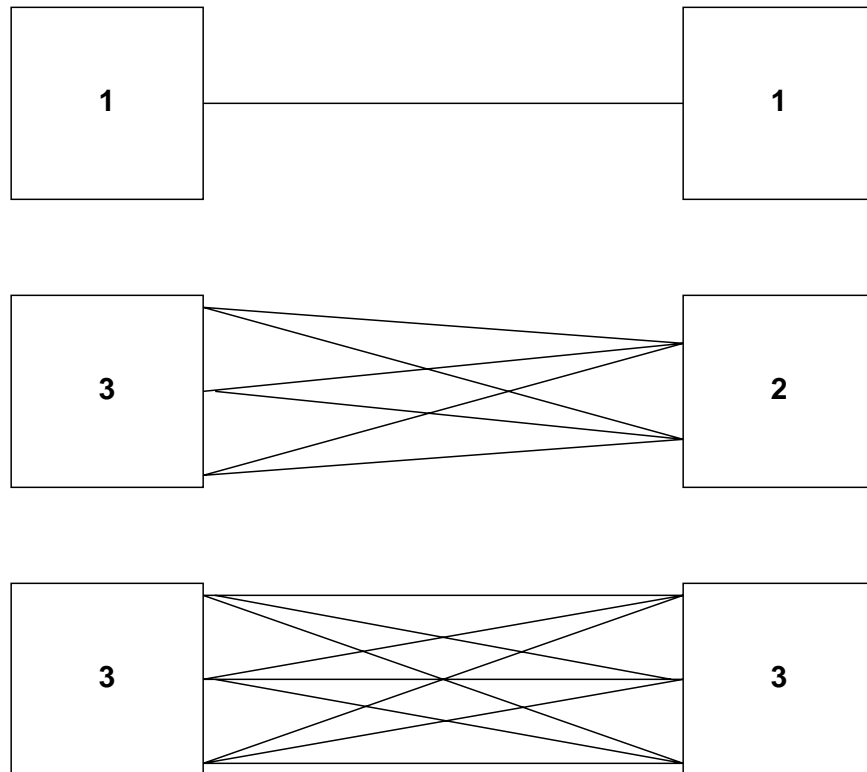


Figure 4.6: Illustrating the change from 1-to-1 to N-to-M addresses.

limits on the number of alternative prefixes they will present or accept, or they could be at risk from Denial of Service attacks by malicious hosts advertising a large number of non-existent prefixes, only to tie up CPU and memory at the victim.

4.3.4 Address change policy

We have described in detail mechanisms to exchange alternative address information with a peer, as well as setting up the necessary additional address mappings. We still have not discussed when to actually make an address change. As the author of the proposal outlines, there are many possibilities.

1. Use ICMPv6 unreachable messages to force an alternative destination address to be used.
2. Let the TCP retry mechanism determine an address change.
3. Let a received TCP segment update the current best path.
4. Internal TCP protocol changes to transmit information.
5. Use IPv6 header options to pass information.

We will implement policy 2 and 3, as well as a policy of changing destination address if the IPv6 layer reports an immediate error because no route can be found to the host. As can be seen from this list, a multitude of options exists and it is outside the scope of this thesis to investigate more than the three mentioned. As the author of the proposal states, it is expected that good methods for converging on a set of addresses will evolve over time.

Almost as important as when to change address is which address to change to. If the change is due to a received packet, the choice is obvious, but for all the other cases any of the alternative addresses could be chosen. Should addresses be retried multiple times? Perhaps sorted, or given scores for reliability. There are many potential schemes, so this is another area where experimentation is required — see Section 8.2.

The straightforward design is to just use addresses in the order sent by the peer, and only try each once. This allows the peer a primitive kind of control over the order in which addresses are tried; it would be an advantage to sort high capacity networks first. If multiple networks of equal capacity are available, they should be randomized in order, so that one network failure will not move all connections to the same network.

When we change destination address for a connection, we have a choice whether we should change the source address as well. If we do change the source address, we must make sure it is among the set acknowledged by the peer in the PREFIX_ACK options header. If it is not, the peer will not consider the new address pair to belong to the same connection. The initial source address is usually picked by the kernel depending on the destination address. A few applications choose their own, using the `bind()` system call. There is a good reason for picking a source address to match the destination address as it might make the communication more efficient by ensuring that traffic in both directions follow the same path. There is no guarantee for such symmetric routes in the Internet though, asymmetric routes are perfectly legal as well. Depending on where the link failure or address change has occurred in the network, changing source address could be both an advantage and a disadvantage. The safe choice is not to change it, because if the reply packets are not delivered, the peer will automatically change its destination address, the same way we did.

4.3.5 Summary

The second half of this chapter has explained a kernel based solution which allows TCP connections to survive network outages between multihomed hosts. The origin of the design was explained, followed by detailed looks at how and where it fits into the current TCP model. A short overview of IPv6 extension headers was given, in order to explain the layout of the PREFIX_SYN and PREFIX_ACK headers. Finally, we looked at how the design changes the fundamental 1-to-1 address assumption of TCP, and some suggestions for when the address of a connection should be changed.

4.4 Summary

From the requirements stated in the previous chapter, we have presented two designs which both solve the fundamental problem of migrating an existing connection on link failure or address change. The first design is based on a new user level API which introduces a session layer between the application and the transport layer. If the existing transport layer connection is broken, a different one can be created without the application's knowledge. The second solution is a modification to the transport layer protocol itself which uses an address exchange mechanism to migrate an existing connection away from a broken link.

5 Implementation

This chapter will describe in detail the implementation of two different approaches to solving the same problem, namely how to make a transport layer TCP connection survive a link failure between multihomed hosts. Both implementations were done on Hewlett-Packard Kayak XU and AMD Athlon based workstations, running FreeBSD 3.2 with the KAME IPv6 extensions. All programming was done in ANSI C and the GNU C Compiler, *gcc*, was used for all compilations.

5.1 The namesocket user space library

The namesocket API is implemented as a statically linked library, called `libnamesocket.a`. There is no reason why it cannot be converted to a shared library on platforms which support this. There is no support for IPv4, the library only supports IPv6 connections. This is merely an implementation choice to keep it simple, there is nothing in the API which limits it to IPv6.

The library uses the same opaque identifier as the socket API, namely a small positive integer. However, users of the library need to be careful never to mix socket and namesocket descriptors. Since namesocket is implemented at the user level, namesocket descriptors are obviously *not* file descriptors, and *only* namesocket API functions may be used to handle them. This identifier is used in the implementation to look up in a table of pointers called `namesockets`. These addresses point to a `struct namesocket` each, which contains all the state for a namesocket, see Figure 5.1.

The mapping of namesocket descriptors to namesocket structures is complemented by another mapping, this time from socket to namesocket. This mapping is necessary in order to implement the `nselect` operation for namesockets and is maintained in the array `socket2namesocket`.

5.1.1 External API functions

The namesocket API exports 9 different functions in the file `namesocket.h`. This is the only file an application needs to include to use the library. As experienced socket programmers will quickly see, all functions are the most commonly used functions from the familiar socket API, with an `n` prepended.

```

struct namesocket
{
    /* client/server common fields */

    int type;                /* NST_xxx */
    int flags;               /* NSF_xxx */
    int state;               /* NSS_xxx */
    int socket;              /* connected or listening socket */
    struct cb data_cb;        /* circular buffer for resending */
    int data_sent, data_rcvd; /* counters for resuming connections */
    struct sockaddr_in6 client; /* address and port of client end */
    struct sockaddr_in6 server; /* address and port of server end */

    /* client only fields */

    struct addrinfo *ai;      /* list of destination addresses, from DNS */
    int num_addr, curr_addr;  /* number of and current address pair */
    struct in6_addr *src_addr; /* table of source addresses, paired with ai */

    /* server only fields */

    int new_socket;          /* new_socket for new/resumed connections */
    struct ns_packet_resume nsp; /* largest packet, for reading packets into */
    int nsp_rcvd;            /* how many bytes received so far */
};

```

Figure 5.1: The structure which holds namesocket state.

The functions all have the same calling convention as well. The first argument is usually a namesocket descriptor, and the return value is an integer, either a namesocket descriptor (or -1 for error) or simply positive for success and negative for error. The cause of an error can be found in the standard manner by inspecting the `errno` variable after a function has failed. Below, the implementation of each function will be described.

int nsocket(int flags)

This function simply allocates, initializes and returns a new namesocket. A new socket is created and held internally in the namesocket state. The flags argument is merely stored in the namesocket for later use.

int nlisten(int nsd, char *service, int backlog)

The second function to be called by a namesocket server, this function first looks up the port number corresponding to the service name using `getservbyname()` (optionally the service name can also be a number in an ASCII string, such as “21”). It then enables the `SO_REUSEADDR` socket option on the socket. Highly suggested for servers, but often forgotten by application programmers, the option allows the server to restart quickly and bind the same socket again, even if it is still lingering around. The socket is then bound to the port. Note that there is no application control over the address bound, the function will always bind to the wildcard address, allowing connections on all IP addresses. This is consistent with our philosophy not to bother the application with IP addresses, and also necessary to allow connections to be resumed. Finally `listen()` is called to set the listen backlog, the number of pending connections allowed for the socket.

int naccept(int nsd)

This function first creates a new namesocket, without an associated socket. Then it loops on the state of the supplied server namesocket until a new connection is fully negotiated. When this is done, it copies the newly connected socket to the newly allocated namesocket, initializes some more state, updates socket mappings and returns the new namesocket descriptor. Notice that because of the loop, the function will block until a new connection is ready (which could be immediately).

int nselect(int ns_nfds, fd_set *ns_readfds, fd_set *ns_writefds, fd_set *ns_exceptfds, struct timeval *timeout)

Probably the least elegant function of the namesocket API, `nselect()` copies the slightly awkward `select()` behavior and adds complexity because of the requirement for mapping namesockets to sockets, and vice versa. Just like the

5 Implementation

socket function, it takes a bitmap of descriptors which we want to wait for events on. However, the bits in the `fd_sets` passed in are of course namesockets and not file descriptors, and we must therefore map them into their equivalent socket descriptors before we can call `select()`. It is a bit unfortunate for this function to take `fd_set` arguments, given that the bits do not represent file descriptors at all. This is merely for convenience, to allow use of the familiar `FD_XXX()` macros to manipulate the bits.

After the call to `select()`, the returned sockets must be mapped back to namesocket descriptors so the returned `fd_sets` can contain the proper bits. This is where the `socket2namesocket` mapping maintained comes in handy. For read sets, we must pay special attention. If the namesocket which has been signaled as ready for reading is a server socket and has not been fully negotiated yet, we cannot return it as read ready to the application yet, since it could be a request to resume an existing connection.

int nconnect(int nsd, char *host, char *service)

A replacement for standard socket code to lookup the address of a host and connect to a specific port on that host, this function takes the hostname and service as string arguments. Similarly to `nlisten()`, the service can be either the name of a service or an ASCII number. The function `getaddrinfo()` which is used to obtain a list of addresses for the server, is not part of the classic socket API, but is part of a set of new functions that are better suited to handle the different structures used for IPv6. A function from the sample code of [36], `get_ifi_info()`, is used to get a list of local interface addresses. For each server address returned, the most appropriate local address is chosen, according to the algorithm described in a draft IPng Working Group document [11]. The algorithm is designed to pick the address which most closely matches the remote address, in terms of scope, address flags and bitwise equality.

The table of matched local and remote addresses is stored in the namesocket, and immediately put to use by attempting a connection to each of the remote addresses in turn, with the socket bound explicitly to the chosen local address. Once a connection is made, an `NSPT_NEW` packet is sent to the remote server to indicate that we are establishing a new connection.

int nsend(int nsd, const void *msg, size_t len, int flags)

Just a small wrapper for `send()`, this function takes care to store the transmitted data in the circular buffer of the namesocket, in case the namesocket is later reconnected.

int nrecv(int nsd, void *buf, size_t len, int flags)

This function is also a short wrapper function, but it has the additional ability to detect a socket error due to a link failure and starting the process of reconnecting

to resume the connection. All the work of resuming a connection is done by the `resume_connection()` utility function, which is described in Section 5.1.2. On successful data reception, the namesocket `data_rcvd` counter is incremented to allow retransmission negotiation in case of reconnection.

int nclose(int nsd)

This function simply closes the socket(s), if any, associated with a namesocket, and frees all allocated memory. It also invalidates the mappings associated with the socket(s) and namesocket.

int nshutdown(int nsd, int how)

This is a very short wrapper function to allow a namesocket to be shut down in either direction, without stopping transmission in the other.

5.1.2 Internal utility functions

Many of the internal functions called upon by the external API functions deal with obtaining the local interface addresses through `ioctl()` calls and are almost completely unmodified code from [36]. The source code has been kept in the original source files: `config.h`, `get_ifi_info.c`, `get_rtaddr.c`, `net_rt_iflist.c`, `unp.h`, `unpifi.h` and `unproute.h`. This code is all standard socket code, and will not be described in any further detail. The utility functions in `cb.c` and `cb.h` are described in section 5.1.3. The final source file is `namesocket_util.c`, and as the name indicates, contains some more utility functions. The first ones are small, uninteresting functions used for matching addresses and looking up mappings in the socket and namesocket tables. The last three functions are more interesting, they handle making and resuming connections, as well as the state machine of a server socket receiving a new connection. These three will be described here.

int make_connection(int nsd)

This function is called upon in two cases, first by the initial `nconnect()` and possibly at a later stage by `resume_connection()` if the connection has been cut off. The function loops through the address pairs stored in the namesocket structure, trying each in turn. First, it binds the local socket, and then it tries to `connect()` to the remote one. If a connection is successful, it records the local and remote addresses and ports in the namesocket structure, they will be required if the namesocket ever needs to be reconnected and resumed.

int resume_connection(int nsd)

This function is designed to be called immediately after some other socket function has failed, because the first thing it does is to inspect the global `errno` variable which indicates the reason for the failure. The namesocket it is called to handle must be of the `CLIENT` type, and it must have previously been connected. If the error seems to indicate some form of link failure such as `ETIMEDOUT` or `EHOSTUNREACH`, it will attempt to reconnect and resume the socket.

The first thing we do is to create a new socket. Unfortunately, the socket API does not allow reusing a socket which has experienced a failure. Then we call on `make_connection()` to do just that, whether to the same or a different IP address than our previous connection does not matter. If successful, we send off an `NSPT_RESUME` request packet, containing the addresses and ports of the old connection, as well as the value of our `data_rcvd` counter. This information is sufficient for the server to locate the old namesocket and determine whether it has enough data stored in its retransmission buffer to resume the connection.

After having read the `NSPT_RESUME_ACK` sent by the server, we need to check that the data lost in transmission to the server is less than or equal to the amount stored in our retransmission buffer. If the lost data is successfully retransmitted then the function is done and returns success.

int handle_server_connection(int nsd)

Probably the biggest single function in the library, this one implements the server side state machine that handles new connections. It is intended to be called every time a server type namesocket is ready for reading, and as is typically the case for a state machine it begins with a switch/case construct on the state variable of the namesocket.

For listening sockets, read readiness indicates a pending connection, so the first thing we do is to `accept()` it. If the connection is an IPv6 one, we advance to the `NSS_READ_REQUEST` state, and record the necessary socket mapping for the newly connected socket.

Data on a namesocket in the `NSS_READ_REQUEST` state is read, but only up to the size of a request packet size. When we have a complete packet, we verify the magic cookie and version number in the header, and check the type of the request. `NSPT_NEW` requests are easy, we just change to the `NSS_CONNECTED` state and leave it to `naccept()` to dish out a new namesocket for the newly connected socket. `NSPT_RESUME` requests are trickier, we have to advance to the `NSS_READ_REQUEST_RESUME` state so we can read the rest of the packet.

If the namesocket is in the `NSS_READ_REQUEST_RESUME` state, we read the rest of the resume request and try find the namesocket it refers to in our table of namesockets. If found, we have to check that we have stored enough data to retransmit, and if so, send an `NSPT_RESUME_ACK` packet with information about how much data we have received on the namesocket. If this transmission, along

with transmission of lost data appears to be successful, we can change the socket in the resumed namesocket to the newly connected one, and continue normal operation.

For `NSS_CONNECTED` namesockets, there is nothing to be done, and the function should never be called for such namesockets.

5.1.3 Circular buffer

The circular buffer is used to store data for retransmission, and is by far the biggest bottleneck of the namesocket user space approach to reconnecting and resuming connections. Unfortunately, it is also necessary, because `send()` will gladly return success for large amounts of data, without the data ever making it further than into a kernel mbuf. Because of this, effort was spent trying to make it efficient.

There are just four functions available, they allow you to initialize, free, put data into and get data out of a circular buffer. The state, and the pointer to the dynamically allocated data buffer is stored in a `struct cb`, supplied by the user.

`cb_put()` stores data, by calculating offsets and lengths and writing the data with `memcpy()`. The function is split in two distinct parts, the first is executed when the amount of data to be stored is greater than or equal to the size of the data buffer. In this case we just do a single `memcpy()` of the tail of the data and adjust the offsets. In the second case, the buffer has room for all the data, but storing the data may require wrapping at the end to store the rest at the front of the data buffer. In any case, the copy is done with either one or two `memcpy()` operations and the offsets are updated.

Retrieving data from the buffer is slightly awkward, but programmed in this manner to use the buffer to the fullest and do not do any unnecessary dynamic allocations. `cb_get()` returns the requested amount of data, or an error, not as a pointer to a single contiguous area, but as two pointers, and two integers indicating the size of the data areas. This is necessary because the data may not be stored contiguously in the data buffer, but may wrap at the end. If the data should be contiguous, the second pointer will be `NULL` and the second size 0. The only penalty for this way of returning the data is two consecutive `send()` calls the two places in the library where lost data is retransmitted.

5.1.4 Logging facility

In order to more easily debug and monitor the actions of the namesocket library, a logging facility was implemented. The primary interface to this is a `log()` function, taking `loglevel` and `format` string parameters as well as a variable amount of arguments. The function works much like the `printf()` standard output function, except that the output is written to a specific filehandle, and only if the urgency level indicated by the `loglevel` parameter is greater than or

5 Implementation

equal to a globally configured minimum urgency level. Three levels of urgency are defined; *Error*, *Info* and *Debug*. They are listed in order of decreasing urgency and increasing verbosity.

A typical call to the logging facility looks like the following.

```
log(LOG_INFO, "Successfully resumed at server\n");
```

This, in turn, produces the following output to the designated log filehandle (by default `stderr`, the standard error output filehandle).

```
Info: 646 - Thu Dec  9 01:53:25 1999 - Successfully resumed at server
```

The number after the urgency level is the process number, obtained with the `getpid()` system call.

An alternative logging function called `logerror()` is also available, this one replaces the commonly used `perror()` standard library function. The function simply prints a banner and the textual equivalent of the error indicated by `errno`. One usage is the following.

```
logerror(LOG_ERROR, "connect");
```

If this was called after `connect` had failed and `errno` was set to `EHOSTUNREACH` (65), this would produce debug output like the following.

```
Error: 415 - Thu Dec  9 01:48:14 1999 - connect: No route to host
```

5.1.5 namesocket server

To test the namesocket library, a simple server was implemented. This is a typical select based server, i.e. it handles multiple connections in a single thread of control. The implementation is stored in the source file `nserver.c`.

For simplicity, the server uses a fixed size table to store the namesockets associated with each client, called `client2nsd`. Initially, this table is filled with -1, to indicate that all of the client slots are available.

The server creates a single namesocket, puts it into listening mode with `nlisten()` and then proceeds into an `nselect()` based input loop. A read `fd_set` is created, with a bit set for the listening socket and each connected socket. If the listening socket is ready for reading, it means a new client has connected. The connection is accepted with `naccept()` and a free slot in the client table is located. The namesocket of the new connection is recorded in this slot.

If a connected socket is ready for reading, the data is read, and on success, echoed back to the client. If `nrecv()` returns 0 bytes read, the socket has been closed, and we can close our end of the socket and free the client slot.

5.1.6 namesocket client

To match the namesocket server, a client was implemented in `nsclient.c`. The client creates a single namesocket, and connects it to the server and service given on the command line. It then enters a loop which does three things. First, it reads a line of text from the standard input, then it sends this line to the server and finally it reads the reply. Coupled with the server described in Section 5.1.5 this gives a simple echo client with both local and remote echo of input.

5.1.7 Summary

In the first half of this chapter we have given a detailed description of the implementation of the namesocket API and protocol. We started off by discussing the presentation of the API as a library, and explained data structures and general calling conventions used. Following this came a description of each API function, and the internal library functions called upon. Finally, we rounded off with a short overview of the server and client implemented to test the library.

5.2 TCP connection preservation at kernel level

The kernel based implementation was done on FreeBSD 3.2 using the KAME snapshot from October 12th, 1999. The KAME IPv6 kernel is fairly uniform across all the BSD architectures supported, but FreeBSD is a little different because the same TCP code is used for both TCP over IPv4 and TCP over IPv6. The central `inpcb` structure which contains all state for the IP level of a connection is also totally integrated, using unions to store IPv4/v6 related information in an overlaid fashion. For other BSDs, there is separate TCP4 and TCP6 code, and state is stored in either an `inpcb` or an `inpcb6` structure. The biggest impact this has on my implementation is that it is slightly more complex, since all code has to check that the layer below is IPv6 before proceeding. Portability is also reduced, a bit more effort is required to merge it into other BSDs.

To put things a bit into perspective, the total number of source and header lines for files in the `netinet` and `netinet6` directories (the IPv4 and IPv6 parts of the kernel source tree) amount to over 90000 lines, spread across 168 files. While the diff to implement this modification only amounts to around 1300 lines of code with modifications to 14 files, there is a lot of inherent complication in modifying an existing system the size of the FreeBSD/KAME TCP/IP kernel. The level of documentation is also a major obstacle, without [41] (which of course only covers up to 4.4BSD-Lite from which FreeBSD is derived) it would have been a *great* deal more difficult.

5.2.1 Adding state to the TCP control block (`tcpcb`)

For every TCP connection in the kernel, two control structures are maintained, containing the entire state of the connection. The `inpcb` structure contains (mostly) IP level state, such as the local and foreign IP address. Above that, the `tcpcb` structure holds sequence numbers, window sizes, TCP state, and other TCP related information. To implement this change, we need to add some information to the `tcpcb`. Any such additions should be carefully reviewed, since busy hosts can easily have hundreds, if not thousands of concurrent connections, and a `tcpcb` will be allocated for each. The variables added were two pointers and two integers. The increased size of the structure depends on the architecture, on a 32 bit only architecture it amounts to 16 bytes.

The first pointer is used to hold a dynamically allocated copy of an `IP6OPT_PREFIX_SYN` option received and changed to `IP6OPT_PREFIX_ACK`. This is the easiest way to echo back the prefixes sent in the SYN packet. The pointer is `NULL` when there are no prefixes to echo back.

The second pointer is also dynamically allocated, it holds an array of IPv6 addresses, the alternative foreign addresses for the connection. This is allocated and filled when we receive an `IP6OPT_PREFIX_SYN` with the initial TCP SYN packet. The two integers are related to this array, one holds the total number of entries, and the other holds the index of the current address in use.

5.2.2 Modifications to `tcp_input()`

`tcp_input()` is the function called to handle incoming IP packets which are deemed valid by the IP layer, and identified as TCP packets. Our first modification comes closely on the heels of the code which looks up the `inpcb` and `tcpcb` for the connection, based on the local and foreign addresses and ports. We compare the packet's local and foreign addresses with the ones stored as primary in the `inpcb`. If these addresses differ, we can tell that the peer has changed address or addresses for the connection, and we follow suit.

The second modification consists of inserting calls to functions which search for and handle `IP6OPT_PREFIX_SYN` and `IP6OPT_PREFIX_ACK` options in incoming TCP packets. This is less trivial than it might appear when looking at a neat TCP state diagram where you can pinpoint exactly the transitions where the code should be inserted. The reason is that the complexity of `tcp_input()` is quite high, with more than 2000 lines of code, 8 goto labels and countless goto statements. Each of the two `tcp_handle_prefix_xxx()` calls are inserted in two places, to catch all cases.

5.2.3 Modifications to `tcp_output()`

Three modifications were made to `tcp_output()`. The first was a simple optimization of the whole function by storing a pointer to the `inpcb` for the connection in a local variable. The `inpcb` was accessed many times in the original

function, but always through the back pointer stored in the `tcpcb`. This is inefficient because it causes the calculation of the address to be performed repeatedly.

The second modification was insertion of `IP6OPT_PREFIX_xxx` options in the packet due to be transmitted. Luckily this code only had to be inserted in one place, and was simply made dependent on the flags of the outgoing packet. If the SYN flag is set, we add the `IP6OPT_PREFIX_SYN` option. If the ACK flag is set and we have stored `PREFIX_ACK` data in the `tcpcb`, the `IP6OPT_PREFIX_ACK` option is added.

The third and final modification consists of some code following the `ip6_output()` call. This function may return an error, for example if the routing tables have changed, and it cannot find a route to the destination address in the packet. Usually such an error condition is just stored as a `softerror` in the `tcpcb` (which is delivered to the application after a timeout), but if our TCP preservation code is active, and we have stored alternative addresses for the peer, we can simply change addresses and hope for better luck along another route.

5.2.4 Constructing `IP6OPT_PREFIX_xxx` headers

Constructing an `IP6OPT_PREFIX_ACK` header is entirely trivial in our implementation. Simply make a copy of the received `IP6OPT_PREFIX_SYN` and change the first byte to the correct identifier.

Making an `IP6OPT_PREFIX_SYN` header is a little more work. In this case we need to loop through all the available interfaces and their associated addresses and compare them with the primary local address chosen for the connection. Three conditions must be valid for the prefix to be added to the header:

1. The scope of the interface address is greater than or equal to the scope of the local address.
2. The prefixes must be different.
3. The host identifiers must be equal.

Some additional housekeeping is done to make sure the new total length of the header does not exceed the maximum length (the IPv6 option header length is encoded as a single byte, so the maximum length is 255 bytes) and to remove the KAME special interface index encoded in the second 16 bit quantity of the IPv6 address.

5.2.5 Changing address for a connection

Two out of three places where the address is changed have already been mentioned, in `tcp_input()` on peer address change, and in `tcp_output()` on transmission failure. The third and last place is in the `tcp_timers()` function which

is called to update the various timers in all active `tcpcb`s every 500 ms. When the `TCPT_REXMT` retransmission timer expires after `TCP_MAXRXTSHIFT` (12) times, TCP usually gives up and declares the connection dead. If there are alternative peer addresses available for the connection, we can instead change foreign address and reset the timer for a second chance.

When changing address in response to a timeout or transmission error, only the destination address is changed, the source address remains the same. This is an implementation choice based on the considerations discussed in Section 4.3.4.

All code to change address or addresses call upon the same function, `tcp_change_addrs()`, stored in `tcp_subr.c`. The first thing this function does is to modify the alias `inpcb` structures for the connection. Please refer to Section 5.2.6 for an explanation of this. The next thing on the agenda is changing the addresses in the `inpcb`, and equally important, in the IPv6 header template that TCP maintains for efficiency. Finally, we delete any cached route from the `inpcb`, as this is invalid after the address change.

5.2.6 Efficient demultiplexing of IPv6 TCP packets

As explained in Section 4.3.3, the really big conceptual change that this kernel modification introduces is to change TCP from a 1-to-1 to N-to-M address relationship. Each end of a connection will need to accept packets to and from potentially many different addresses. This impacts on the TCP/IP kernel's code to demultiplex incoming IP packets to belong to different connections. In the classic 4.4BSD-Lite stack this was done rather inefficiently as a linear search in the table of TCP control blocks. FreeBSD incorporates enhancements to use a hash table instead. The hashing was of course designed for IPv4 addresses which are 32 bit quantities. KAME has not really updated the hashing for IPv6, and judging by the "XXX" comment following the code to hash on the last 32 bits of the 128 bit IPv6 address, the KAME programmers are not really happy with the solution either.

The big question is then, how do we add N-to-M capabilities to the demultiplexing code without reducing it to a slow linear search again or adding nasty multilevel matching to the hashed search? The solution chosen was to introduce "alias" `inpcb` structures, minimal `inpcb` structures which are only present to allow lookups of their addresses to return the real, main `inpcb` which has different addresses stored. In practical terms, this meant adding a new `alias_to` pointer to the `inpcb` structure. When this pointer is non-NULL, it is an alias for the `inpcb` that it points to. Adding state to the `inpcb` comes with all the concerns described in Section 5.2.1, but even more so, because an `inpcb` is allocated for ALL protocols layered above IP, not just TCP. With this in mind, an attempt was made to identify alias `inpcb` structures by a flag in the existing flag member, and abusing the `inp_ppcb` pointer (which usually points to the protocol control block, in the case of TCP, the `tcpcb`) to point to the real `inpcb`. This attempt was abandoned because of the wide implications of pointing the `inp_ppcb` at

something other than a protocol control block or NULL, resulting in a multitude of crashes.

Demultiplexing packets for TCP connections running over IPv6 is done by a function called `in6_pcblookup_hash()`. Instead of modifying all code that calls this function to handle return of an alias `inpcb`, a simple wrapper function was introduced by renaming the original function to `in6_pcblookup_hash_real()`. When this function returns an alias `inpcb`, the wrapper function simply changes the return value to the real `inpcb`.

Special care must also be taken when deleting `inpcb` structures, as performed by `in6_pcbdetach()`. We cannot allow alias `inpcb` structures to linger, pointing to freed memory. Therefore, the `pcblist` that the `inpcb` is attached to is searched for alias `inpcb` structures, and these are deleted before the original. A few other cases where code accesses invalid fields in alias `inpcb` structures has also been modified to test for NULL values, but there is a significant possibility that some such cases may remain.

Fairly obviously, the alias `inpcb` structures are added in the first place when we receive an `IP6OPT_PREFIX_ACK` in the ACK of our SYN TCP packet from the peer. The contents of the header are checked so that we do not add aliases for illegal prefixes inserted by a malicious or buggy peer. Aliases are added for all permutations of the set of local and foreign addresses, including the original addresses used for the connection. The only exception is the permutation consisting of the actual original local and foreign addresses.

The set of alias `inpcb` structures must be modified when we change addresses. `tcp_change_addr()` bypasses the `pcb` lookup wrapper function by calling the `in6_pcblookup_hash_real()` function and checking that the result is indeed an alias for the `inpcb` that we are changing addresses for. The alias is updated so that it contains the addresses used before the address change, and rehashed. After the address change in the original `inpcb`, it too is rehashed.

5.2.7 Summary

In this second part of the chapter, we have described the implementation of Peter R. Tattam's kernel based proposal for preserving TCP connections between multihomed hosts. The important issues were the state added to the TCP control block, the modifications to the TCP input and output functions and the alias `inpcb` mechanism to perform demultiplexing. Other details discussed were the construction of the prefix protocol packets and the actual low level address change operation.

5.3 A comparison of the two implementations

While the two implementations appear to solve the same problem on the surface, with the `namesocket` API additionally providing a DNS name based simplified

socket interface, there are many inherent differences. These have been categorized and will be described in turn in this section.

5.3.1 User vs kernel space

There is one huge advantage to a user space implementation over a kernel one. If your code crashes, it only takes down the application, and not the whole system. It goes without saying that any code should undergo careful scrutiny before being added to a stable kernel.

An additional advantage to the user space approach is the fact that it can be prototyped in a high level language such as Python¹, which usually offers much lower development time. Kernel solutions will probably always require a lower level language such as C or assembly.

5.3.2 Application modification

Making an existing application use the namesocket API requires substantial changes, and may not even be possible for all applications. This is a result of the fact that the API is designed to be a simplified, more abstract version of the socket API, for simple TCP applications. The kernel solution automatically benefits all applications, precisely because it resides in the kernel which all applications call upon.

The kernel solution may be undesirable for some applications, and a socket option should be implemented to disable it. This would require a very minor change to those applications to toggle the option.

5.3.3 Dealing with address changes

The two solutions differ somewhat in how they respond to address changes. The namesocket API relies on DNS to look up the addresses of the peer. Often, a DNS update will be delayed because BIND caches lookups for a configurable amount of time before going to the source again. For the kernel solution, address changes are effective as soon as `ifconfig` finishes adding the new address to the interface. There is no difference in dealing with local address changes.

Only the namesocket solution has the potential to discover new addresses for the peer in the middle of a connection, for example before attempting a resume. This is not currently implemented in the namesocket library though, it only looks up the addresses before initiating a connection and stores them for subsequent use. For security reasons, the design of the kernel solution explicitly forbids changing the set of addresses after the initial TCP SYN/ACK exchange.

¹An interpreted, interactive, object-oriented programming language, see <http://www.python.org>.

5.3.4 Overhead

With regard to this very important point, the kernel solution has the upper hand in almost every area; connection speed, memory usage, bandwidth and latency. The only disadvantage it has is that a small part of the memory overhead it adds, the extra state in the `inpcb` and `tcpcb`, which is not dynamically allocated is *always* present, regardless of whether the connection uses it or not. This also goes for code involved, some additional overhead is always present in order to check for IP packet options, alternative peer addresses, etc.

The user space solution loses out because it has to duplicate a lot of the information found in the kernel connection state. This includes the local and remote addresses, as well as the buffers allocated for buffering data for retransmission. Additionally, much of this state has to be obtained by crossing into the kernel through a system call. Examples are `ioctl()` calls to get local interfaces and addresses, and `get(sock|peer)name()` calls. System calls are very expensive in CPU time, and should be avoided as much as possible. The namesocket library could be optimized to buffer the local interface addresses for certain periods of time for example.

The overhead involved in making or resuming a connection to another host is very different between the two solutions. On initial connection, the user space solution sends a small data packet which of course requires the remote TCP to acknowledge it. When resuming a connection, a bigger initial packet is sent, and the peer answers with its own acknowledgment data packet. This will add several round trip times of latency to the ordinary TCP 3-way handshake connection establishment.

In contrast, the kernel solution just adds extra options to the packets which will be exchanged regardless, and these options are coded with a fairly high efficiency to avoid wasting header space. This additional header overhead should have no measurable impact on connection establishment speed, unless either host carries an unusually large amount of alternative addresses.

The worst overhead present in the user space solution is not the general state or the connection establishment latency though, it is the extra data copy that each end must do to resume reliably. For a fast CPU with a limited network speed (less than Gb/s class) it might not limit the bandwidth much, but it will increase CPU utilization, perhaps significantly.

5.3.5 Interoperability

The namesocket API can only be used between two namesocket peers, because of the use of initial data packets to negotiate the connection. This data would confuse any non-namesocket server and probably ruin the connection.

The kernel solution does not suffer from this problem. Interoperability with non-compliant hosts is smooth, they will simply ignore the `IP6OPT_PREFIX_SYN` in the SYN packet header, and no further actions will be taken. This is a function

5 Implementation

of the clever IPv6 option header encoding which allows a host to specify the action that the remote end should take if it does not understand the option. The choices, as explained in [23] are one of:

1. Skip the option.
2. Discard packet, do not send ICMP report.
3. Discard packet, send ICMP report even if destination address is multicast.
4. Discard packet, send ICMP report if the destination address is not multicast.

As we saw in Section 4.3.2, the prefix headers use the first choice, and hosts will simply ignore the `PREFIX_SYN` header sent.

5.3.6 Limitations imposed

The kernel solution imposes no special limitations, unfortunately the namesocket one has a few. First of all, applications which use it cannot `fork()` and access namesockets from the child process. This is a result of the global table used for locating namesocket structures from namesocket descriptors. This has the most impact on servers, which commonly `fork()` to handle new connections. Only single threaded servers using `nselect()` to multiplex, or multi-threading using true threads within the same address space can be allowed.

The namesocket approach also has a significant limitation in that only the client is allowed to change address (i.e. reconnect and resume). Working around this in user space would have required the client setting up a listening socket, which could quickly get very complicated. The kernel solution is of course truly N-to-M.

For a production system, these would be significant disadvantages. For a prototype intended for research into reconnecting and resuming connections at user level, it is not a problem.

5.3.7 Portability

Portability is one area where the user space solution finally scores a point over the kernel solution. After all, both solutions require implementation at both endpoints of a connection in order to offer TCP connections which survive link failure or address change. And the user space solution should be fairly easily portable to just about any architecture in use today, given the widespread use of the BSD socket API, even on non-BSD hosts. Some parts, especially the code to get a list of local interfaces and addresses will need modification, but this is just a small isolated part. The kernel solution requires modification to an often large and complex kernel, which may be completely different from architecture to architecture.

5.3.8 Backwards compatibility

There are no real backwards compatibility issues with the namesocket API, given that it is a completely new API. The kernel solution changes a few assumptions that could previously be made about the socket API. The most obvious one is the fact that addresses are no longer guaranteed to remain the same throughout the life of a connection. Few, if any, applications depend on this fact, but it does raise the question if there ought to be a way for the applications to be notified about an address change.

A quite common practice today is to listen only on a specific set of IPs as a security measure, to prevent connections from unauthorized hosts. This is especially common on firewalls which are also servers for the LAN that they protect. The kernel solution does not change the situation for initial connections, but if the hostid is the same for the Internet and LAN visible addresses of the host, then alias `inpcb` structures would be created such that packets from the Internet would be judged to belong to the connection, which could open up vulnerabilities through packets maliciously spoofing as part of the connection. Possibly, aliasing should only be allowed on sockets which are bound to the wildcard address.

Alternatively, addresses could be separated into groups, and sharing of addresses belonging to a different group could be disallowed. This is almost what the IPv6 *scope* does, with its hierarchy of link-local, site-local and global addresses. The implementation respects scope, in the LAN/firewall example, Internet hosts would not learn the site-local prefix of the server. The global prefix would be shared with the LAN, but that would not represent a security risk.

5.3.9 Security

Any scheme to change existing and proven protocols make alarm bells go off in the heads of Internet security experts. The noise resulting from a scheme to change the 1-to-1 address relationship of TCP connections probably bring the neighbors running over.

The two biggest fears are going to be in the areas of spoofing and hijacking of connections. Spoofing means to insert your own packets as part of a connection between two different hosts, and hijacking means to take it over and leave one party completely out of it.

The namesocket library was not intended to be secure against either, all it takes is knowledge of the addresses and ports of the two endpoints, a connection to the server of the connection, and it can be hijacked.

A serious security review of the kernel solution is outside the scope of this thesis, but given that the set of addresses is not allowed to change after the initial connection setup, we agree with the author of the proposal that the security level should be somewhere near equivalent to a 1-to-1 connection.

5.3.10 Summary

In the third and final part of the chapter, the two described implementations were compared in areas such as ease of integration, compatibility, overhead and security. In most cases, the kernel solution is shown to hold the upper hand, except when it comes to ease of integration. This is expected and acceptable, because the user level solution is only intended as a prototype.

6 Demonstration

This chapter will give a demonstration of both of the two implementations described in the preceding chapters. The implementations are described in separate sections, but the test system configuration is the same for both and therefore shown independently first.

6.1 Demo configuration

The setup illustrated in Figure 6.1 consists of two hosts connected by two separate networks. The hosts are identically configured Hewlett-Packard Kayak XU workstations, featuring dual Intel Pentium II 300 MHz processors and 256 MB RAM. The operating system used is FreeBSD 3.2 with the KAME IPv6 extensions.

The figure shows the host identifiers of the hosts inside the boxes, and the prefixes of the networks above the lines. As can be seen, the prefixes are both 64 bits long while the hostids are 32 bits. The third 32 bit quantity of the addresses is therefore zero in all addresses.

All addresses are registered in DNS.

6.2 Namesocket library

In this section we will demonstrate the namesocket library in use, by setting up a connection between the test client and server described in Section 5.1.6 and 5.1.5. The connection will then be physically broken by unplugging the network cable and letting the namesocket library reconnect and resume the connection. The demonstration will consist of an explanation of the commands executed,

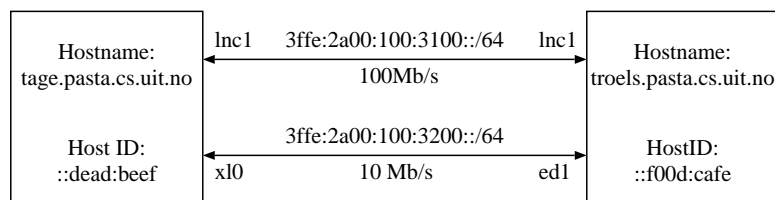


Figure 6.1: The demo host and network configuration.

6 Demonstration

and the output from the client, the server and the namesocket library. The namesocket library output consists of debug information provided by the `log()` function, as described in Section 5.1.4.

Note that the clocks on the two hosts are *not* synchronized, therefore the time stamps printed by the logging facility should not be compared between the hosts.

The server is started first, on the *tage* host.

```
[troels@tage namesocket]$ ./nsserver 1234
Debug: 646 - Thu Dec 9 01:48:25 1999 - Binding socket to port 1234
```

Then the client is launched on the *troels* host. The first action which causes debug output to be printed is the source address selection. For each of the two addresses returned for the *tage* hostname, a source address must be selected. To shorten the debug output, the address selection is only shown for one address (`3ffe:2a00:100:3100::dead:beef`) and superfluous information such as address flags has been trimmed away. The client then attempts a connection to each address in turn.

```
[troels@troels namesocket]$ nsclient tage 1234
Debug: 415 - Thu Dec 9 01:43:20 1999 - Destination address:
                                3ffe:2a00:100:3100::dead:beef
Debug: 415 - Thu Dec 9 01:43:20 1999 - Candidate source address:
                                fe80:1::260:b0ff:feb5:6643
Debug: 415 - Thu Dec 9 01:43:20 1999 - Chosen by default
Debug: 415 - Thu Dec 9 01:43:20 1999 - Candidate source address:
                                3ffe:2a00:100:3001:260:b0ff:feb5:6643
Debug: 415 - Thu Dec 9 01:43:20 1999 - Chosen by scope
Debug: 415 - Thu Dec 9 01:43:20 1999 - Candidate source address:
                                3ffe:2a00:100:3100::f00d:cafe
Debug: 415 - Thu Dec 9 01:43:20 1999 - Chosen by prefix match length
Debug: 415 - Thu Dec 9 01:43:20 1999 - Candidate source address:
                                3ffe:2a00:100:3200::f00d:cafe
Debug: 415 - Thu Dec 9 01:43:20 1999 - Candidate source address:
                                fe80:2::220:18ff:fe3a:1355
Debug: 415 - Thu Dec 9 01:43:20 1999 - Candidate source address:
                                fe80:6::1
Debug: 415 - Thu Dec 9 01:43:20 1999 - Candidate source address: ::1
Debug: 415 - Thu Dec 9 01:43:20 1999 - Attempting connection with
                                address pair 0:
Debug: 415 - Thu Dec 9 01:43:20 1999 - src: 3ffe:2a00:100:3100::f00d:cafe
Debug: 415 - Thu Dec 9 01:43:20 1999 - dst: 3ffe:2a00:100:3100::dead:beef
```

Returning to the server, it receives the connection and prepares to read an initial packet.

```

Debug: 646 - Thu Dec  9 01:48:31 1999 - handle_server_connection,
                                         nsd = 0, state = 1
Info: 646 - Thu Dec  9 01:48:31 1999 - New connection from
                                         3ffe:2a00:100:3100::f00d:cafe,
                                         port 1040
Debug: 646 - Thu Dec  9 01:48:31 1999 - New connection not ready yet
Debug: 646 - Thu Dec  9 01:48:31 1999 - handle_server_connection,
                                         nsd = 0, state = 2

```

The client sends a packet of type NEW right after making the connection.

```

Debug: 415 - Thu Dec  9 01:43:20 1999 - Sent NSPT_NEW packet

```

The namesocket layer on the server side reads the packet and informs the server application that a new connection is pending. The application calls `naccept()` and gets a new namesocket descriptor in return. This connection becomes Client 0, and this fact is printed.

```

Debug: 646 - Thu Dec  9 01:48:31 1999 - Reading request header
Debug: 646 - Thu Dec  9 01:48:31 1999 - Verifying header contents
Info: 646 - Thu Dec  9 01:48:31 1999 - Got NEW connection
Debug: 646 - Thu Dec  9 01:48:31 1999 - Signaling read ready server
                                         socket to application
Debug: 646 - Thu Dec  9 01:48:31 1999 - naccept(0) returning new nsd 1
Client 0: Connected

```

The user of the client enters some text to be sent.

```

helo

```

The server receives the text and echoes it back.

```

Debug: 646 - Thu Dec  9 01:51:21 1999 - Signaling read ready server
                                         socket to application
Debug: 646 - Thu Dec  9 01:51:21 1999 - nrecv: got 5 bytes
Client 0: Got 5 bytes

```

The client receives the echo and prints it.

```

Debug: 415 - Thu Dec  9 01:46:10 1999 - nrecv: got 5 bytes
helo

```

6 Demonstration

Up till now, we have not seen namesocket do anything that the socket API cannot handle. To provoke some more action, we unplug network cable used for the connection and enter some more text to be echoed. Remember, TCP does not know that the cable is gone, so the send call will succeed. However, the internal retransmission of TCP will eventually time out, and the timeout will unblock the waiting `recv` call with an error. This triggers namesocket's reconnect as we shall see.

```
come again
Debug: 415 - Thu Dec  9 01:47:29 1999 - nrecv: got -1 bytes
Debug: 415 - Thu Dec  9 01:47:29 1999 - resume_connection called,
                                     errno: 60 = Operation timed out
Info: 415 - Thu Dec  9 01:47:29 1999 - Attempting resume...
Debug: 415 - Thu Dec  9 01:47:29 1999 - Attempting connection with
                                     address pair 0:
Debug: 415 - Thu Dec  9 01:47:29 1999 - src: 3ffe:2a00:100:3100::f00d:cafe
Debug: 415 - Thu Dec  9 01:47:29 1999 - dst: 3ffe:2a00:100:3100::dead:beef
Error: 415 - Thu Dec  9 01:48:14 1999 - connect: No route to host
Debug: 415 - Thu Dec  9 01:48:14 1999 - Attempting connection with
                                     address pair 1:
Debug: 415 - Thu Dec  9 01:48:14 1999 - src: 3ffe:2a00:100:3200::f00d:cafe
Debug: 415 - Thu Dec  9 01:48:14 1999 - dst: 3ffe:2a00:100:3200::dead:beef
```

Notice that namesocket tries each address pair again, beginning with the first. The first pair fails after less than a minute with a “no route to host” error. The old route to the `3ffe:2a00:100:3100::dead:beef` address has been deleted as part of TCP's retry mechanism, and a new one cannot be created because the hardware address of the *tage* host cannot be found through IPv6's Neighbor Discovery mechanism [30] when the network is down.

However, the second address pair works, and the namesocket library at the server end accepts the connection (without notifying the application yet).

```
Debug: 646 - Thu Dec  9 01:53:25 1999 - handle_server_connection,
                                     nsd = 0, state = 1
Info: 646 - Thu Dec  9 01:53:25 1999 - New connection from
                                     3ffe:2a00:100:3200::f00d:cafe,
                                     port 1042
Debug: 646 - Thu Dec  9 01:53:25 1999 - New connection not ready yet
Debug: 646 - Thu Dec  9 01:53:25 1999 - handle_server_connection,
                                     nsd = 0, state = 2
```

The client namesocket library sends off a packet of type `RESUME`, containing the address and port pairs for the connection.

```
Debug: 415 - Thu Dec  9 01:48:14 1999 - Sending NSPT_RESUME
```

On receiving this packet, the server side namesocket library searches its table of namesockets, to see if it has a server namesocket with the address and port pairs in the packet. This search succeeds and it proceeds to acknowledge the resume with a RESUME_ACK packet.

```

Debug: 646 - Thu Dec  9 01:53:25 1999 - Reading request header
Debug: 646 - Thu Dec  9 01:53:25 1999 - Verifying header contents
Info: 646 - Thu Dec  9 01:53:25 1999 - Got RESUME connection
Debug: 646 - Thu Dec  9 01:53:25 1999 - New connection not ready yet
Debug: 646 - Thu Dec  9 01:53:25 1999 - handle_server_connection,
                                     nsd = 0, state = 3
Debug: 646 - Thu Dec  9 01:53:25 1999 - Reading remaining part of
                                     resume header
Debug: 646 - Thu Dec  9 01:53:25 1999 - Searching for server socket
                                     to resume
Debug: 646 - Thu Dec  9 01:53:25 1999 - Found a connected server socket...
Debug: 646 - Thu Dec  9 01:53:25 1999 - c1: 3ffe:2a00:100:3100::f00d:cafe.1040
Debug: 646 - Thu Dec  9 01:53:25 1999 - c2: 3ffe:2a00:100:3100::f00d:cafe.1040
Debug: 646 - Thu Dec  9 01:53:25 1999 - s1: 3ffe:2a00:100:3100::dead:beef.1234
Debug: 646 - Thu Dec  9 01:53:25 1999 - s2: 3ffe:2a00:100:3100::dead:beef.1234
Debug: 646 - Thu Dec  9 01:53:25 1999 - Addr's and ports match
Debug: 646 - Thu Dec  9 01:53:25 1999 - Checking lost data..
Debug: 646 - Thu Dec  9 01:53:25 1999 - Sending NSPT_RESUME_ACK

```

The client namesocket library receives the RESUME_ACK and verifies it as a valid packet.

```

Debug: 415 - Thu Dec  9 01:48:14 1999 - Reading NSPT_RESUME reply
Debug: 415 - Thu Dec  9 01:48:14 1999 - Verifying NSPT_RESUME_ACK
                                     header contents

```

At the server end, the namesocket library finishes its business by sending any lost data (none in this case) and replaces the underlying socket in the namesocket with the newly resumed one.

```

Debug: 646 - Thu Dec  9 01:53:25 1999 - Sending 0 bytes lost data
Info: 646 - Thu Dec  9 01:53:25 1999 - Successfully resumed at server
Debug: 646 - Thu Dec  9 01:53:25 1999 - New connection not ready yet

```

The client end has some lost data to retransmit, but once that is taken care of, the namesocket is back in operation.

```
Debug: 415 - Thu Dec  9 01:48:14 1999 - Lost data: 11 bytes
Debug: 415 - Thu Dec  9 01:48:14 1999 - Sending 11 bytes lost data
Info: 415 - Thu Dec  9 01:48:14 1999 - Resume successfully completed
                                     by client
```

The lost data arrives at the server over the new connection, and is promptly echoed back.

```
Debug: 646 - Thu Dec  9 01:53:25 1999 - Signaling read ready server
                                     socket to application
Debug: 646 - Thu Dec  9 01:53:25 1999 - nrecv: got 11 bytes
Client 0: Got 11 bytes
```

There are no broken cables this time, so the echo arrives safely and is printed by the client.

```
Debug: 415 - Thu Dec  9 01:48:14 1999 - nrecv: got 11 bytes
come again
```

6.2.1 Summary

This concludes the demonstration of the namesocket user space library. It has been shown to successfully accomplish its goal of resuming a TCP connection without the application taking any notice. This process can be repeated any number of times, in case of a very unstable network. Should the original network go up again, and the second go down, the connection would be migrated back to the original, and so forth.

There are two delays in the resume process that we do not control. The first is the time it takes before we get the initial timeout error which triggers the reconnect. By default, TCP retransmits data many times, using exponential backoff combined with the measured round trip time to calculate the delay between each retransmission. The second delay is the one before the connection to address pair 0 fails. This is dependent on TCP's retransmission of the initial SYN segment, which times out significantly quicker than the data retransmission.

We cannot change these delays directly, but we could maintain our own timer which expires earlier. Doing that would require making our own estimates about round trip time and the appropriate number of retransmissions, but we see no reason to override TCP in this regard.

6.3 PreserveTCP kernel modification

The same scenario presented in Section 6.2 will also be used to present the kernel solution at work. Since this applies to all socket applications, we do not need any special test applications and we therefore abuse the FTP daemon on the *tage* host as an echo server, and connect using the ordinary telnet client. The kernel solution also prints debug information when in use, although not quite as detailed as the namesocket library. The debug information from both the server and client side will be used to explain what the kernels at each end are doing.

The FTP server is spawned when needed by `inetd` which runs continuously. This means we do not have to start it manually, we can go right ahead and start the client on the *troels* host. The client will connect and read the connection banner printed by the server. Note that telnet is a well programmed TCP client which tries all destination addresses in turn. In this case the connection to the first address is successful, and no further attempts need to be made.

```
[troels@troels troels]$ telnet tage 21
Trying 3ffe:2a00:100:3100::dead:beef...
Connected to tage.
Escape character is '^]'.
220 tage.pasta.cs.uit.no FTP server (Version 6.00) ready.
```

The primary addresses for the connection are initially `3ffe:2a00:0100:3100::f00d:cafe` at the client side, and `3ffe:2a00:0100:3100::dead:beef` at the server side. Each side will send a PREFIX_SYN IPv6 options header to the other side with their alternative prefix. This prefix is used to form an alternative address, and aliases are created for each source and destination address permutation. Two addresses at each end gives four permutations, one primary and three aliases. At the client end, this results in the following output.

```
alt addr 0: 3ffe:2a00:0100:3200::dead:beef
added alias (3ffe:2a00:0100:3100::f00d:cafe.1039,
             3ffe:2a00:0100:3200::dead:beef.21)
added alias (3ffe:2a00:0100:3200::f00d:cafe.1039,
             3ffe:2a00:0100:3100::dead:beef.21)
added alias (3ffe:2a00:0100:3200::f00d:cafe.1039,
             3ffe:2a00:0100:3200::dead:beef.21)
```

Similarly, the server side kernel prints the following.

```
alt addr 0: 3ffe:2a00:0100:3200::f00d:cafe
added alias (3ffe:2a00:0100:3100::dead:beef.21,
             3ffe:2a00:0100:3200::f00d:cafe.1039)
added alias (3ffe:2a00:0100:3200::dead:beef.21,
             3ffe:2a00:0100:3100::f00d:cafe.1039)
added alias (3ffe:2a00:0100:3200::dead:beef.21,
             3ffe:2a00:0100:3200::f00d:cafe.1039)
```

6 Demonstration

We enter some text to verify that the primary connection is working, and see it echoed by the FTP server.

```
hi there
500 'HI there': command not understood.
```

This happy state of affairs is quickly ended by unplugging the network cable to simulate a network outage. Some text is entered at the client side to get some action.

```
do your thing
```

At this point, running the `netstat` command shows us the state of the connection. The addresses are cut off, but we can see that both ends are using the 100 Mb network addresses, there are 15 bytes queued to send and the state is ESTABLISHED.

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	(state)
tcp6	0	15	3ffe:2a00:100:31.1039	3ffe:2a00:100:31.21	ESTABLISHED

After a while, TCP grows weary of retransmitting to the same address, and we change to the alternative destination address. Notice that the source address stays unchanged.

```
tcp_timers: changing address in response to retransmission timeout
tcp_change_addrs: changed address(es) to
                  (3ffe:2a00:0100:3100::f00d:cafe.1039,
                   3ffe:2a00:0100:3200::dead:beef.21)
```

With the destination address, the segment of data is delivered to the server. Noticing that the client has changed address for the connection, the server quickly follows suit.

```
tcp_input: changing address in response to peer address change
tcp_change_addrs: changed address(es) to
                  (3ffe:2a00:0100:3200::dead:beef.21,
                   3ffe:2a00:0100:3100::f00d:cafe.1039)
```

Using this new address pair the server attempts to reply. Alas, the segment cannot be delivered, because the IP layer routes packets according to destination address only, and this is of course unchanged.

Again, TCP's retransmission mechanism steps in and keeps up the attempts for a while. Eventually, it gives up, and similarly to what happened at the client end, the destination address is replaced by the alternative one stored away. The time taken before TCP gives up retransmitting was previously discussed in Section 6.2.1.

```
tcp_timers: changing address in response to retransmission timeout
tcp_change_addrs: changed address(es) to
                (3ffe:2a00:0100:3200::dead:beef.21,
                 3ffe:2a00:0100:3200::f00d:cafe.1039)
```

When the segment arrives over the 10 Mb network, the client sees that the server has changed address and immediately copies this change.

```
tcp_input: changing address in response to peer address change
tcp_change_addrs: changed address(es) to
                (3ffe:2a00:0100:3200::f00d:cafe.1039,
                 3ffe:2a00:0100:3200::dead:beef.21)
```

Finally, the reply to our request is delivered to and printed by the client. The period of time which has passed since we typed in the request, i.e. the total time to migrate the connection, is approximately twice the time required before TCP gives up on a retransmission (once for each host). The implementation details surrounding this were explained in Section 5.2.5.

```
500 'DO your thing': command not understood.
```

Calling upon the `netstat` tool again, we can see that the connection has indeed fully migrated to the 10 Mb network.

```
Proto Recv-Q Send-Q Local Address          Foreign Address        (state)
tcp6      0      0 3ffe:2a00:100:32.1039 3ffe:2a00:100:32.21  ESTABLISHED
```

6.3.1 Summary

In this section we showed the time line of events for TCP communication between two hosts implementing the PreserveTCP mechanism. From the establishment of a connection over one network cable, to the migration to another, each step was narrated and illustrated with kernel debug output. We saw how a network outage in this case prompted no less than four address changes until both ends were completely migrated. The number of address changes is largely an implementation policy issue and was discussed further in Section 4.3.4.

7 Performance evaluation

Some vague estimates about the overhead introduced by the two implementations were mentioned in Section 5.3.4. In this chapter we will quantify the estimates with measured numbers and evaluate how accurate the estimates were.

7.1 Choosing measures

The first order of business is to decide which areas we wish to measure. Again referring to Section 5.3.4, we see that there are four primary areas which we speculate have been worsened by either solution.

1. Memory usage.
2. Latency.
3. Throughput.
4. CPU usage.

The first item is easier to estimate by looking at the code than by measuring. For the next two, we have chosen three tests which should show the overhead quite clearly. Ideally, we would like to measure CPU usage during all of the performance tests, to see whether that has changed as well. The three tests are:

1. Measuring send throughput by timing how fast data is transmitted through a socket.
2. Measuring `send/recv` latency by sending and receiving a single byte through a socket.
3. Measuring the overhead added to the TCP `connect` operation by connecting and closing a socket.

7.2 Making measurements

Several different existing TCP benchmark programs, including the popular and versatile Netperf¹, were evaluated for our use. Although Netperf seemed initially

¹Written primarily by Rick Jones at Hewlett-Packard, see <http://www.netperf.org>.

suitable, the desire to have complete control over the measurements won out and a custom benchmark tool, *tcpbench*, was created. The program supports just the three benchmarks described in Section 7.1, using both the socket and namesocket APIs. The source code of the program is contained entirely in a file called `tcpbench.c`, which is supplied along with the other code as described in Appendix A.

To perform a benchmark, the program is run in server mode on one machine, and in client mode on the same or another machine. Each benchmark is run a given number of iterations, and the time taken for each iteration is measured with the `gettimeofday()` system call. Using this call gives us microsecond precision, and we can calculate the number of operations per second, or alternatively the number of seconds per operation. We would have liked to measure CPU usage as well, but that is a complicated operation which requires operating system support for credible numbers. As such, it was deemed to be outside the scope of this work.

As always when doing benchmarks, we want to run each test for many iterations, since imprecision in each measurement gets averaged out over time. Measurement imprecision may occur as a result of external noise, such as process scheduling, context switches, virtual memory swapping, clock drift, other processes, or any other properties of the system which are outside our control.

Unfortunately, running a high number of iterations turned out to be difficult to do with the third benchmark, measuring the overhead in connecting and closing a socket. The problem is that each socket created does not go away immediately, the TCP connection stays around in the `TIME_WAIT` state in order to avoid old packets from being interpreted as part of a new connection [36, p.41]. The `TIME_WAIT` state lasts much longer than the benchmark, so all sockets, with associated TCP control block, etc, stays allocated during the entire run. FreeBSD has a limit to the number of such allocations, and it is not entirely trivial to increase this limit.

Another way of solving the problem is to use the socket option `SO_LINGER` to make sockets close immediately, without transmitting buffered data. This also causes TCP to skip the `TIME_WAIT` state, and deallocate socket resources immediately. This only works for the socket API testing though, namesocket may have lingering data, recall from Section 4.2.2 that the namesocket protocol sends a single packet of type `NEW` to indicate to the remote end that it is making a new connection. If we turned off lingering for namesockets which are closed rapidly after connection, we could be invalidating the whole test by preventing TCP from sending the packet.

The end result was that the third benchmark was only run for 500 iterations, while the others were run for 100000 iterations. The low number of iterations is not critical though, as long as we do not use the numbers for anything other than comparing them with each other.

7.3 Expected results

The throughput benchmark has a second parameter in addition to the number of iterations, namely the size of the buffer used for transmission. The size of this buffer makes a difference to the *absolute* performance shown, since it is obviously more efficient to transfer more data at a time. This is however not very interesting for our purposes, we are only interested in measuring *relative* performance. There is another reason why we should care about the buffer size though. Recall the circular buffer used for storing data for retransmission and described in Section 5.1.3. This buffer is optimized to only copy the tail of data sent which exceeds the size of the retransmission buffer. For data smaller than the buffer size, there is more overhead because multiple `memcpy()` operations may be needed, the data could wrap around at the end, etc.

For this test, the internal retransmission buffer of the namesocket library was set to 1024 bytes, and the size of the send buffer was varied between 512, 2048 and 8192 bytes. We would expect the performance impact of the copy into the retransmission buffer to drop off as the send buffer size increases.

When we test the throughput of a kernel with the TCP modification, we do not expect to see any drop in performance. The code added should only be executed for SYN and ACK of SYN segments, not data segments.

The latency test should also show the same performance, for the same reasons. When running through the namesocket API though, we should expect performance degradation due to the copy into the send buffer and the extra function calls involved.

In the connection speed test, we expect to see both the kernel and user space solutions show a drop in performance. Both of them add overhead to the 3-way handshake, the kernel solution with extra packet header options, namesocket by sending extra data after the handshake. As explained in Section 5.3.4, the namesocket overhead will likely be the greatest, because it involves extra packets and round trip time.

7.4 Measured results and analysis

The benchmarks were all performed on the same AMD Athlon 500 MHz workstation, with 128 MB RAM. The loopback interface was used for testing, to avoid any inaccuracy added by the network. A minimal number of background services were running during each test, and each benchmark was run three times. The results were averaged out using the arithmetic mean of the three runs.

For the address used, one single alternative 64 bit prefix was available to be exchanged both ways when the PreserveTCP kernel modification was tested. This added 16 bytes to the IPv6 header of the TCP SYN and ACK of SYN segments.

Benchmark	Normal	Namesocket	Ratio	PreserveTCP	Ratio
Throughput - 512 B	35.3 10 ⁶ B/s	32.3 10 ⁶ B/s	0.92	34.4 10 ⁶ B/s	0.97
Throughput - 2048 B	80.9 10 ⁶ B/s	72.3 10 ⁶ B/s	0.89	78.3 10 ⁶ B/s	0.97
Throughput - 8192 B	105.3 10 ⁶ B/s	103.4 10 ⁶ B/s	0.98	102.1 10 ⁶ B/s	0.97
Latency	20522 ops/s	18875 ops/s	0.92	20825 ops/s	1.01
Connection	6253 ops/s	2614 ops/s	0.42	3561 ops/s	0.57

Table 7.1: Results from performance measurements.

Looking at the throughput benchmarks of the namesocket implementation compared to the reference numbers, two things make themselves immediately apparent. The first is that the performance is lower with namesocket than without, and the second is the curious dip in performance when increasing the send and receive buffers to 2048 buffers. As explained in Section 7.3, the drop in performance was expected, but we also expected the drop to taper off as the size of the buffer increased, because of the static 1024 byte retransmission buffer in namesocket. What we see here is that the actual process of copying the bytes to the retransmission buffer is more significant than any penalty imposed by calculating offsets and copying into the middle of the buffer. In the first throughput test we copy the whole buffer of 512 bytes for each send. In the second test, we only copy half the buffer, but that is still twice as much data as in the first test, therefore it has a bigger impact on performance.

In the third test, we only copy 1/8 of the send buffer and the performance is 98% of the normal performance, a very small penalty. Note that the buffer sizes used were all powers of two, which is an advantage for the namesocket library because it leads to more efficient copying.

Looking at numbers for the kernel solution, we can see that they are all exactly 97% of the normal kernel results. This is a bit worrying, we did not expect a drop in performance for throughput. It would be easier to dismiss the numbers as noise and imprecise measurements if they were not so consistent. There are just three lines of code which are executed for every TCP segment received, regardless of whether they are part of the 3-way handshake or normal data transmission. The code is in `tcp_input()` and checks whether the peer has changed address(es) for the connection, and if so changes the local addresses as well. In the implementation tested, this test is performed regardless of whether there are in fact any alternative addresses for the current connection. It is not entirely unlikely that the code causes some performance decrease, since it is comparing the entire length of two pairs of 128 bit addresses for each TCP segment received. The code could be optimized to only compare the addresses if there are in fact any alternative addresses available. Such an optimization would not improve the results for this test though, since we do in fact have an alternative address.

Moving on to the latency test, the choice of unit, operations/s, might seem a curious choice, given that latency is usually measured in units of s/operation. This unit was chosen to deemphasize the importance of the absolute numbers,

we are only concerned with the relative performance of the three systems.

For namesocket, performance is reduced, as expected, to 92% of the reference system. Depending on the compiler implementation of the `memcpy()` call used to copy bytes into the circular buffer, this area of performance could probably be improved by using a simple inline loop to copy small amounts of data (like the one byte sent in this test).

Curiously, the modified kernel shows *increased* performance in the latency test, albeit only by 1%. This is most likely just noise since sending and receiving data should not be affected by the changes. The only change which could explain performance improvements would be the small general optimization made to the `tcp_output()` as described in Section 5.2.3. There is no logical explanation why this optimization would benefit the latency test much more than the throughput test though. Both the code described in the discussion of the throughput results above, and the optimization in `tcp_output()` affect *every* packet, no matter the number.

The final benchmark tested the connection speed. The namesocket solution reduces performance to 42% of the normal socket API. This was expected and clearly explained by the extra data sent at the beginning. The modified kernel shows 57% the performance of an unmodified kernel. This number is closer to the namesocket performance than we might have expected from the implementations. The reason is quite simple though, namesocket sends a single data packet with 4 bytes at the beginning of a connection, while the kernel solution adds 16 bytes to packets sent in both directions. When testing through the loopback interface we do not see the disadvantage that namesocket has because it adds extra round trip time to the connection establishment. The higher the latency of the network, the bigger the advantage of the kernel solution.

The biggest performance disadvantage of namesocket when testing through the loopback interface is probably the extra work it does for each IP address of the remote host, picking optimal source address, etc. The kernel solution also does some work for each address exchanged, but less than namesocket.

7.5 Summary

Using a custom developed benchmark program, we have managed to quantify the overhead added by the two implementations in several important areas. The measurements were discussed in comparison to the performance of an unmodified reference system, and in practically every case we found what we had expected. There was one deviant case where performance had unexpectedly increased, but only by a small amount, well within the margins of error.

None of the two solutions compared showed unreasonable performance, but the kernel solution was superior in all tests.

8 Conclusion

8.1 Achievements

David D. Clark, a senior scientist of the MIT Laboratory for Computer Science and chair of the IAB¹ from 1981 to 1990, is reported to have said about the work of the IETF; “We reject kings, presidents, and voting; we believe in rough consensus and running code” [22, p.23]. Through open subcommittees, such as the IPng Working Group, the IETF uses this guiding principle to develop standards for the Internet.

In this work we started by taking a look at the different multihoming problems identified by the IPng Working Group, as well as current drafts and proposals of solutions. From this, we derived a set of requirements for a useful multihoming contribution, with a primary function of migrating transport layer connections on link-failure or address change. In the design, we saw that the functional requirements could be met in two different ways, and we created complete designs for both. The designs were based on preliminary drafts and proposals, and therefore required varying degrees of additional work to be completely specified.

The designs were then implemented in an existing, open operating system and demonstrated to fulfill the functional requirements. Advantages and disadvantages the solutions were pointed out, often by comparing one to the other. Finally the two implementations were put through a performance evaluation, which showed that both were performance wise sound, but also that the kernel solution held the upper hand in all tests.

We conclude that the design and implementation of the kernel solution is very promising, and worthy of further development. Returning to the Internet standards process, the next natural step would be to create an Internet Draft, and present it to the Internet community. If the draft is found acceptable and other implementations are created, it would eventually be published as an Internet standard in the RFC format.

8.2 Future work

During the development of both of the presented solutions, many ideas and avenues of experimentation came up. The namesocket user space library may

¹The Internet Architecture Board, an organization dedicated to long term studies of the Internet.

be useful to others as a prototype platform for experimentation and to give ideas for an official new API to improve multihoming, but the most potential lies with the kernel solution which proved itself to surpass the user space implementation in practically all areas.

This does not mean that the user space library is totally redundant when the kernel solution is in place. The kernel solution depends on making an initial connection in order to exchange the alternative addresses. This in turn means that the application, or an API like namesocket, acting on behalf of the application, must try all addresses for the remote host at connection time.

We have many ideas for future work with the kernel solution. An important step would be to port the implementation to current versions of the FreeBSD kernel and indeed to all platforms supported by the KAME distribution. This would allow it to be evaluated more thoroughly, both in terms of other architectures and interoperability.

Some implementation details, like the alignment of data in the options headers and the use of alias `inpcb` structures deserve further analysis, for example in terms of performance and impact on other parts of the kernel.

As mentioned in Section 5.3.8, some sites or applications may wish to disable the solution. Kernel and socket level options should be provided in the implementation to facilitate this.

The idea of giving prefixes exchanged individual priorities has come up in IPng Working Group discussions. This is another aspect of address change policy, an area with considerable room for experimentation. For more information about this issue, see Section 4.3.4.

Another issue raised in IPng Working Group discussions is the fact that the TCP checksum only covers an IPv6 pseudo-header which does not include header options. This means that any corruption to the prefixes sent in the `PREFIX_SYN` header is undetectable by the receiver. This problem could be solved either through extending the TCP checksum, or by including a separate checksum in the `PREFIX_SYN` header.

Further work in security is also appropriate. Is the mechanism secure enough, and how secure does it need to be? This should be explored both in terms of hijacking and spoofing attacks, as well as Denial of Service. To prevent the latter, it might be prudent to explicitly limit the number of prefixes in the headers.

Possibly, different levels of security could be implemented, in a user configurable manner. This could for example be used to allow changes to the set of addresses; a risky operation in terms of security, but also a very useful one to deal with for example network renumbering.

The mechanism might be useful outside the area of fault tolerancy as well. It could for example be used to migrate traffic to an alternative provider due to some external policy like time of day or network load.

As mentioned in Section 2.7.3, some of the Mobile IPv6 mechanisms, like the home address destination options header, could also be used within the context of this solution. Employing this mechanism could be used to tip the scales in the weighing between added state or added network load in the implementation.

The many potential ways to continue this work is perhaps the most significant result. Rather than closing off all research in an area, we have carved out an initial path and shown a number of exciting opportunities for future work.

References

- [1] ALBITZ, P., AND LIU, C. *DNS and BIND*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1998.
- [2] BATES, T., AND REKHTER, Y. RFC 2260: Scalable Support for Multihomed Multi-provider Connectivity, Jan. 1998.
- [3] CERF, V., AND KAHN, R. A Protocol for Packet Network Interconnection. *IEEE Trans. on Commun. COM-22* (May 1974), 637–648.
- [4] CONTA, A., AND DEERING, S. RFC 2463: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification, Dec. 1998.
- [5] CRAWFORD, M. RFC 2672: Non-Terminal DNS Name Redirection, Aug. 1999.
- [6] CRAWFORD, M. RFC 2673: Binary Labels in the Domain Name System, Aug. 1999.
- [7] CRAWFORD, M., HUITEMA, C., AND THOMSON, S. DNS Extensions to Support IPv6 Address Aggregation and Renumbering, Nov. 1999. Work in progress.
- [8] DAY, J., AND ZIMMERMAN, H. The OSI Reference Model. In *Proceedings of the IEEE* (Dec. 1983), pp. 1334–1340.
- [9] DEERING, S., AND HINDEN, R. RFC 2460: Internet Protocol, Version 6 (IPv6) specification, Dec. 1998.
- [10] DRAVES, R. Default Address Selection for IPv6, Oct. 1999. Work in progress.
- [11] DRAVES, R. Simple Source Address Selection for IPv6, June 1999. Work in progress.
- [12] DROMS, R. RFC 2131: Dynamic Host Configuration Protocol, Mar. 1997.
- [13] DUPONT, F. Multihomed routing domain issues for IPv6 aggregatable scheme, Sept. 1999. Work in progress.
- [14] EGEVANG, K., AND FRANCIS, P. RFC 1631: The IP Network Address Translator (NAT), May 1994.

References

- [15] FERGUSON, P., AND SENIE, D. RFC 2267: Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing, Jan. 1998.
- [16] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACHAND, P., AND BERNERS-LEE, T. RFC 2616: Hypertext Transfer Protocol — HTTP/1.1, June 1999.
- [17] FULLER, V., LI, T., YU, J., AND VARADHAN, K. RFC 1519: Classless inter-domain routing (CIDR): an address assignment and aggregation strategy, Sept. 1993.
- [18] HINDEN, R., AND DEERING, S. RFC 2373: IP version 6 addressing architecture, July 1998.
- [19] HINDEN, R., AND DEERING, S. Minutes of IPng Working Group Meeting, Tokyo, Sept. 1999.
- [20] HINDEN, R., AND MANKIN, A. Minutes of IPng Working Group Meeting, Washington DC, Nov. 1999.
- [21] HINDEN, R., O'DELL, M., AND DEERING, S. RFC 2374: An IPv6 aggregatable global unicast address format, July 1998.
- [22] HUITEMA, C. *Routing in the Internet*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1995.
- [23] HUITEMA, C. *IPv6: The New Internet Protocol*. Prentice Hall, 1996.
- [24] ICHIRO HAGINO, J. IPv6 multihoming support at site exit routers, July 1999. Work in progress.
- [25] INTERNET ASSIGNED NUMBER AUTHORITY. IPv6 Parameters, Sept. 1999.
- [26] JOHNSON, D. B., AND PERKINS, C. Mobility Support in IPv6, Oct. 1999. Work in progress.
- [27] KIRKPATRICK, S., STAHL, M. K., AND RECKER, M. RFC 1166: Internet numbers, July 1990.
- [28] MANKIN, A. Minutes of IPng Working Group Meeting, Oslo, July 1999.
- [29] MOCKAPETRIS, P. V. RFC 1035: Domain names — implementation and specification, Nov. 1987.
- [30] NARTEN, T., NORDMARK, E., AND SIMPSON, W. RFC 2461: Neighbor discovery for IP Version 6 (IPv6), Dec. 1998.
- [31] POSTEL, J. RFC 761: DoD standard Transmission Control Protocol, Jan. 1980.
- [32] POSTEL, J. RFC 791: Internet Protocol, Sept. 1981.

- [33] POSTEL, J., AND REYNOLDS, J. K. RFC 959: File transfer protocol, Oct. 1985.
- [34] REKHTER, Y., AND LI, T. RFC 1518: An architecture for IP address allocation with CIDR, Sept. 1993.
- [35] SOMMERVILLE, I. *Software Engineering*, fifth ed. Addison-Wesley, 1996.
- [36] STEVENS, W. R. *UNIX network programming: Networking APIs: sockets and XTI*, second ed., vol. 1. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, 1998.
- [37] TANENBAUM, A. S. *Computer Networks*. International Edition. Prentice-Hall International, 1996.
- [38] TATTAM, P. R. Preserving Active TCP sessions on Multihomed IPv6 Networks, Sept. 1999. Work in progress.
- [39] THOMSON, S., AND HUITEMA, C. RFC 1886: DNS extensions to support IP version 6, Dec. 1995.
- [40] VIXIE, EDITOR, P., THOMSON, S., REKHTER, Y., AND BOUND, J. RFC 2136: Dynamic Updates in the Domain Name System (DNS UPDATE), Apr. 1997.
- [41] WRIGHT, G., AND STEVENS, W. R. *TCP/IP Illustrated: The Implementation*. Addison-Wesley, Reading, MA, USA, 1995.
- [42] YU, J. J. IPv6 Multihoming with Route Aggregation, Nov. 1999. Work in progress.

References

A The CDROM

The attached CDROM contains all material related to the work. This includes:

- The thesis itself, in PostScript and PDF format.
- All IPng Working Group drafts referenced in the thesis as well as the HTML file with the proposal from Peter R. Tattam.
- The source code to the namesocket library.
- The source code to the PreserveTCP kernel implementation. This is available in three different formats:
 1. The complete FreeBSD 3.2 kernel tree, with the KAME IPv6 patch from October 12th, 1999 and the PreserveTCP patch applied. All files modified by the PreserveTCP patch are present in the original form, with the extension `.orig`.
 2. A unified diff containing all changes.
 3. Individual colored diffs for each file modified. Reading these colored diffs is the recommended way of browsing the source code.
- The source code to the tcpbench benchmark program used for performance measurements.

The root directory of the CDROM contains a file called `index.html` with hyperlinks to the contents. All source code produced by the author is also available in HTML, with syntax highlighting and coloring for browsing.