An Architecture for Reliable Encapsulation Endpoints using Commodity Hardware

by

Robert M. Robinson

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical & Computer Engineering

Waterloo, Ontario, Canada, 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Customized hardware is expensive and making software reliable is difficult to achieve as complexity increases. Recent trends towards computing in the cloud have highlighted the importance of being able to operate continuously in the presence of unreliable hardware and, as services continue to grow in complexity, it is necessary to build systems that are able to operate not only in the presence of unreliable hardware but also failure-vulnerable software. This thesis describes a newly developed approach for building networking software that exposes a reliable encapsulation service to clients and runs on unreliable, commodity hardware without substantially increasing the implementation complexity. The proposal was implemented in an existing encapsulation system, and experimental analysis has shown that packets are lost for between 200 ms and 1 second during a failover, and that a failover adds less than 5 seconds to the total download time of several sizes of files. The approach described in this thesis demonstrates the viability of building high availability systems using commodity components and failure-vulnerable server software.

Acknowledgments

I would like to acknowledge my thesis supervisor, Dr. Paul A. S. Ward of the University of Waterloo, for his guidance and contributions to this thesis. I would also like to thank Dr. Michael J. Robinson of the University of Calgary for his assistance with structuring the statistical analysis of my experimental results. I wish to express my gratitude to Pravala, Inc. for allowing me to utilize the *Accoriem* platform to implement and test my proposal, as well as providing me with the time and resources to explore this problem space.

I wish to thank the editors of this document, Dr. David Taylor, Nicholas Armstrong, and Todd Kemp, for taking the time to ensure that my thesis was properly structured and readable. I also wish to thank Dr. Rudolph Seviora and Dr. Sebastian Fischmeister, both of the University of Waterloo, for acting as the readers of this thesis.

Dedication

This is dedicated to my parents, Cathy and Mike.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Encapsulation systems are commonly used to allow unsupported network protocols to run over an existing network. This is accomplished by wrapping a fully-formed packet from an unsupported protocol inside another, supported protocol; allowing the network to treat traffic generated by the unsupported protocol as traffic it already handles. Building a reliable encapsulation system has traditionally required either complex server-side failover solutions or weak reliability guarantees, neither of which are particularly appealing approaches for highly available network systems. This thesis proposes a third approach for building high reliability encapsulation systems — redundant server-side components with loose coupling and an intelligent client component that actively maintains connections with multiple servers and selects the most appropriate server to pass traffic through. This approach is believed to be easier to implement than existing solutions, provides minimal packet loss during a server-side failure, works on commodity hardware, and can be adopted by a variety of encapsulation systems.

Encapsulation is commonly used as an approach to add additional features to the network without having to change the underlying network protocol — Internet Protocol Security (IPsec) [14], Teredo [11], Mobile Internet Protocol (IP) [18], and IP-in-IP [21] are all examples of this paradigm. It is common to deploy many of these solutions in a client/server architecture, with many client devices running the encapsulation protocol connecting to a single server responsible for decapsulating traffic and passing it on to the

destination network. In these systems, failures in either the client or server component will cause network traffic to stop flowing; in the case of a client failure only the local device will lose access, but in the case of a server failure all of the clients will lose access. Current systems approach this problem from one of two perspectives: a complex server-side failover solution that enables clients to maintain connectivity, or a simple approach that does not maintain connectivity but simply reconnects to a new server.

One of the more complex aspects of handling the failure of a server in the encapsulation environment is ensuring that traffic is still routed to the client after the failure has occurred. Encapsulation systems are typically exposed to the client as a virtual network interface that receives a subset of all IP traffic generated by the client, with each interface being assigned an IP address that acts as the source IP for all traffic entering the encapsulation system. The address that is assigned to the client typically comes from a subnet that has been assigned to the corresponding server, with different servers assigning addresses from different subnets. Even if all servers assign addresses from the same subnet, they will have to perform Network Address Translation (NAT) on the addresses before they leave the server, because it is not possible for two clients to have the same transport address visible to the wider network. Consequently, if a client disconnects from a server and connects to another one, it will not keep its transport address, which causes all flows traversing the encapsulation system to break. This is undesirable because many applications do not currently handle address changes well; further, it requires handling server failure on a per-application basis. It be would preferable if migration from one server to another could be accomplished without changing the address assigned to the virtual interface; that is, performed in a 'seamless' (from the client perspective) manner.

The primary goal of this thesis is to develop a mechanism for software-based encapsulation systems to provide their clients with continuous, uninterrupted service in the presence of server failures. The most important measure of whether a solution provides continuous service in the presence of server failures is whether a client application that is passing traffic through the encapsulation system can continue operating without having the connection drop, time out, or otherwise close. More specific measures of the impact of a failure include how many packets are lost during a failover and the impact of a failover on file download

times. The complexity of the solution is also considered — solutions that require few modifications to the system and solutions that have components that can be utilized by other encapsulations systems attempting to achieve similar goals are preferred as they provide more value to developers of similar systems.

The major contributions of this thesis are the proposal of a new architecture for failure-tolerant encapsulation systems, the development of an analytical framework to measure the impact of a failure, and the experimental validation of said architecture in an actual encapsulation system. The solution detailed in this thesis is believed to be the first example of a failure-tolerant encapsulation architecture that neither requires a complex server implementation, nor passive failover devices. It is believed that this work is applicable not only in the discussed example but also in other commonly used encapsulation systems, including Virtual Private Networks (VPNs), Internet Protocol version 6 (IPv6) transition technologies, and Mobile IP-capable solutions. Wide adoption of the principles discussed in this thesis will make it possible to develop and deploy highly available encapsulation systems in the future, making it easier for individuals or organizations to continue adding new features to existing network layer protocols without having to develop new approaches for providing fault-tolerance and high-availability capabilities.

This thesis is divided into six chapters. Chapter 2 includes a more detailed description of the problem space, as well as a survey of related approaches. This includes both research proposals and implemented systems. Next, Chapter 3 contains the design of the encapsulation reliability solution, with a focus on the overall design of the system and its impact on system behaviour. Chapter 4 details the implementation of the encapsulation reliability design in the context of an existing system. In Chapter 5 the implementation is validated using several micro-benchmarks. Conclusions and future work are presented in Chapter 6.

Chapter 2

Background and Related Work

A typical encapsulation system incorporates elements from a variety of computer-science fields; the underlying transport mechanism is built much like a typical Layer 2 (L2) point-to-point transport protocol, the packet forwarding logic uses similar algorithms to those used in traditional Layer 3 (L3) routers, and the reliability and failover behaviour has been extensively studied in both the high-availability routing literature and the high-availability distributed systems literature.

The use of encapsulation to support additional functionality on top of the base Internet Protocol version 4 (IPv4) protocol has a long history of use on the Internet. It is commonly used in two separate but related roles, legacy support and network isolation. Encapsulation has been used to enable non-IPv4 protocols to be usable over the Internet; there are a variety of standards describing this mode of operation including Generic Routing Encapsulation (GRE), 6to4, and Teredo. A more common deployment architecture, however, involves using encapsulation to provide network isolation for IPv4 traffic that needs to traverse the Internet to reach its destination. One common example of encapsulation providing network isolation can be seen in a VPN scenario. The network-isolation approach has also been codified in a number of different standards, including both IP-in-IP and IPsec, as well as being used by application designers (*OpenVPN*) to provide similar isolation capabilities.

2.1 Traditional Encapsulation Systems

A typical encapsulation architecture has two components, as shown in Figure 2.1. These components are the Encapsulation Node (EN) and the Decapsulation Node (DN)[1]. The EN typically runs on a client device, such as a smartphone or a laptop, and initiates a connection to the DN. The DN typically runs somewhere in the network that is directly connected to the next hop destination of the encapsulated traffic. This can include a private corporate network, the IPv6 Internet, or the IPv4 Internet, depending on the deployment requirements. The EN and DN may participate in one or more authentication sessions, encryption-algorithm negotiations, or other configuration steps before both endpoints are ready to transfer data. Completion of the initialization process will result in any packets that the EN receives on its virtual interface being sent to the DN. Each DN assigns a client an IP address from a pre-configured subnet; each DN is usually assigned a unique subnet to prevent duplicate addresses. Once the EN has established a session with the DN, it cannot migrate to another DN without tearing down its session with the first. This coupling is necessitated both by the assignment of an IP address to the EN (routing entries to the EN are, by definition, set to have the DN as the next hop) and potentially by the transport protocol employed by the EN and DN themselves. The use of Transmission Control Protocol (TCP) as a transport protocol prevents seamless migration because sequencing and other state issues prevent the connection from continuing properly on a new host. Typically, the DN is implemented on specialized or dedicated hardware, to ensure that the encapsulation system offers high availability to the ENs.

There are two distinct approaches to performing encapsulation — wrapping the data to be encapsulated directly in IP, or utilizing a transport protocol running over IP to carry the encapsulated data. There are many standardized examples of both approaches; GRE, 6to4, IP-in-IP, and IPsec all encapsulate their data in raw IPv4 packets, and as a result have been assigned different IP protocol numbers to allow routers to treat these forms of traffic differently from TCP or User Datagram Protocol (UDP). Newer protocols have

[1]While the terms EN and DN are used to differentiate the two devices, both endpoints perform encapsulation and decapsulation operations — for simplicity a single direction of traffic flow is assumed in the rest of this thesis even though it holds for the reverse as well.

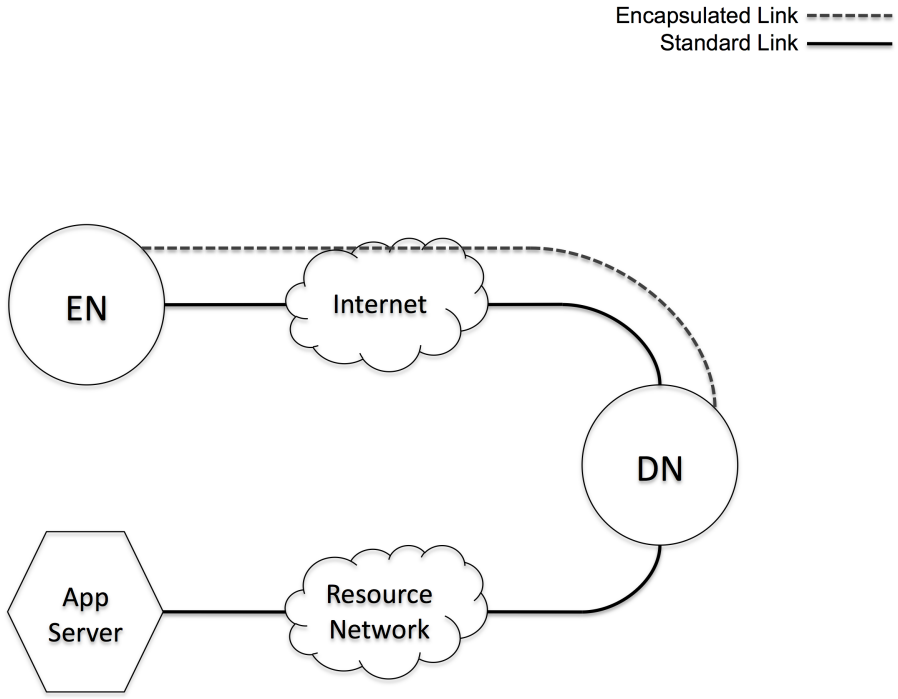

Figure 2.1: Typical Encapsulation Architecture

adopted the latter approach, with protocols such as Teredo and OpenVPN encapsulating their data in either TCP or UDP. The reasons for this are discussed below.

Protocols that directly encapsulate traffic in IPv4 are considered to be the preferable approach because they minimize the amount of overhead added to every packet. While preferable from an overhead perspective, they encounter major deployment issues on the Internet today partly as a result of the prevalence of NAT in access networks and partly due to restrictive firewall configurations. This has caused problems not only for encapsulation systems (see IPsec passthrough mode on some routers) but for newer transport protocols, such as Stream Control Transfer Protocol (SCTP) as well. To work around the deployment issues that NAT introduces, several encapsulation protocols that directly use IPv4, for example IPsec, have been updated to utilize transport-level encapsulation. RFC 3948

[12] was developed to allow IPsec to function in the presence of NAT by wrapping the Encapsulating Security Protocol (ESP) packets in UDP.

2.2 Traditional Encapsulation Failover

The underlying protocol used to encapsulate data has a direct impact on the complexity of enabling seamless server failover. Approaches that utilize a stateless transport layer, either directly using IP or wrapping a payload in UDP, can potentially utilize existing IP-redundancy protocols such as Hot Standby Router Protocol (HSRP) or Virtual Routing Redundacy Protocol (VRRP). In such an environment, both servers will be configured to share a single IP address, with the primary server receiving all traffic and the standby configured to take over should the primary fail. If the IP-redundancy protocol is used both on the inbound interface (to allow the client traffic to continue arriving) and on the outbound interface (to ensure traffic returning from the encapsulation network can be routed back to the client properly) then the failure of a single server will go unnoticed by the client. This approach does not, however, work with systems that utilize a stateful transport layer, because while IP redundancy protocols allow for an IP address to migrate between servers, they do not do the same for transport connections. There is no widely supported mechanism to migrate the state of a TCP connection between operating-system instances, requiring that the client handle failure cases through a reconnection approach.

To allow TCP connections to be migrated between independent operating system instances, researchers have developed a system that integrates with the server-side TCP stack. This solution, named Fault-Tolerant Transmission Control Protocol (FT-TCP) [1] [31], combines traditional IP failover protocols with a new component that sits both above and below the existing TCP stack to intercept all packets being send and received using TCP. It logs all packets to a data store that is shared amongst all the hosts in the cluster. When a host fails, another takes over and 'picks up' from where the last host left off based upon the state committed to the shared data store. The new host continues the connection by rewriting all subsequent incoming and outgoing TCP packets to reflect the proper state of the connection. This enables FT-TCP to overcome the state and se-

quencing issues which traditionally prevent TCP connections from being able to migrate between hosts. While FT-TCP enables a TCP connection to be migrated between hosts, it has several disadvantages - supporting this protocol requires tight coupling with a host operating system, as well as requiring tight coupling between all the hosts in the cluster. As well, it requires an active-passive deployment architecture, which tends not to scale in an efficient manner. One final limitation is that FT-TCP is limited to supporting TCP failover between different hosts - any application that uses UDP will require a different solution.

The IP-redundancy protocols have a limitation as well; it is only possible for a single machine to be the primary server for a specific IP address. Load balancing across HSRP or VRRP requires multiple IP addresses configured on the cluster with each machine assigned as primary to one of the addresses [24]. This complicates active-active configurations with more than a single server because it is now necessary to load balance clients across the IP addresses exposed by the system.

Utilizing existing IP-redundancy protocols directly has the advantage of simplifying the client; it simply connects to a single IP address and passes traffic to the server. The tradeoff to this simplicity is the complexity of the server implementation — it becomes necessary to synchronize all state used in the decapsulation process between all of the failover servers in real-time, otherwise a failure of the primary server will result in interruption of service to the client. This state is not always static — IPsec requires that session keys for each client be shared between all servers that may process a client's traffic, and any service with restricted access requires that authentication information be shared between all servers. The complexity of this state synchronization only increases as the encapsulation system is expanded to support a larger number of servers that can be simultaneously used, especially in load-balancing architectures. It is relatively straightforward to synchronize two servers that are sharing an IP address in an active/passive architecture, but synchronizing many servers in an active/active architecture is a much more difficult problem.

The approach proposed here sidesteps many of these concerns by pushing the failover logic into the client — a small amount of extra logic enables clients to connect to an arbitrary number of servers that do not need to be directly synchronized with the other

servers in the failover cluster. It enables the solution to scale to a large number of servers in an active/active architecture without requiring complex server-side logic or configuration parameters to support, and does not require scaling IP-redundancy protocols.

2.3 The Complexity of IP Continuity

In order to provide the client a continuous, uninterrupted connection to the Internet, it is necessary to provide more than just a redundant connection to the decapsulator, it is also necessary to ensure that the IP address assigned to its virtual interface does not change as the client moves between decapsulators. The IP address assigned to the EN is able to migrate between interfaces as the EN enters and leaves different networks as long as it is communicating with the same DN; the DN advertises the subnet from which the EN addresses are assigned and other routers use this information to send the EN's traffic to the proper DN. In most environments this is statically configured because the information is not dynamic — if a DN goes down then all of the ENs connected to it are no longer reachable. This mode of operation is not acceptable in a fault-tolerant system, however, and as a result it is necessary to ensure that an EN's IP address will migrate between DNs in the relevant failure scenarios.

Providing IP continuity to clients in an encapsulation system has not been the focus of a lot of literature to this point, router-reliability protocols such as HSRP or VRRP excepted. There are two related fields that have received a lot of research recently, however — supporting address reassignments (for a variety of reasons) and providing IP continuity in virtualized datacentres. A summary of the relevant research of each field is presented below.

2.3.1 Address Translation

Address translation is a somewhat overloaded term; here the intent is not to refer to the one-to-many NAT commonly used to provide connectivity to many clients behind a single IP address, instead the term is used to refer to a one-to-one NAT whereby each client has

a unique address on both sides of the NAT device, these addresses just happen to exist in different address spaces. This approach is used not only to help reduce the size of global routing tables, but also to simplify the transition to IPv6.

The paper *Towards a new Internet routing architecture* [30] observes that one of the biggest contributions to the growth of the global BGP routing table is the number of Provider Independent (PI) address subnets that are being assigned, and examines several solutions to reduce the impact that PI networks have on the global routing table while still allowing for multihoming across independent Internet Service Providers (ISPs). PI addressing space is named as such because it is not allocated from address blocks that carriers have been assigned (hence, 'provider independent'), both to simplify renumbering when changing providers and to enable multihoming. The use of PI addressing has increased in recent years for both of the previously mentioned reasons; unfortunately it is essentially impossible to aggregate PI addresses because they are inherently assigned to disjoint organizations. This dramatically increases the size of the global routing table, since all providers now need to know how to reach each PI block independently. To support PI addressing while still enabling address aggregation, *Towards a new Internet routing architecture* [30] proposes a two-layer addressing scheme whereby PI space is mapped into Provider Assigned (PA) addresses, which enables clients to be assigned a single address that is persistent across ISP failure, and in effect enables IP continuity in a failure environment. Essentially, the border routers between the PI and PA addressing spaces perform a version of one-to-one NAT, except that multiple 'public' addresses map to a single 'private' address.

The author of *Six/One* [29] proposes a solution for enabling PI addressing in IPv6 deployments by deploying what are essentially one-to-one NAT devices on both sides of the transit network. This architecture is similar to *Towards a new Internet routing architecture* [30] because it relies on a transparent, in-network mechanism for mapping client addresses to globally reachable addresses in such a manner that an assigned globally reachable address can change without impacting the client address or its connections. This does not completely work with IPv4, however, because it relies upon an IPv6 extension header to carry the original, untranslated address to the end host, making it possible for other

hosts to associate connections coming from two different addresses as actually coming from the same host (if the PI-to-PA mapping changes partway through a connection). Clearly, this is a major limitation, because upgrading all end hosts on the Internet to understand multiple sets of addresses in the IP header is not a trivial or rapidly accomplished task.

Both *Towards a new Internet routing architecture* [30] and *Six/One* [29] are built with the assumption that the PI space is managed independently from the PA address space, allowing the organization to manage the operation of the PI space as it wishes. This is an important aspect of both approaches; organizations have the flexibility to move hosts throughout their organization without interrupting existing connections. Adapting this to an encapsulation system requires an extra layer of indirection be added to the system, yet if the routes can be properly adjusted within the organization during a failure it is possible to hide this failure from existing connections.

2.3.2 Datacentre Networking

The adoption of virtualization technology in the datacentre has provided a number of advantages to operators, including improved utilization, increased flexibility, and higher uptime. Virtual Machines (VMs) are able to be migrated between physical systems, making it possible for physical hosts to be taken down for maintenance without requiring any observable downtime. Virtual-machine migration is a substantial advantage provided by a virtual environment; however, enabling this migration greatly complicates the underlying network infrastructure. Different VMs are expected to run on different networks, and moving these machines between physical hosts requires that every possible network a VM is connected to be available to all of the physical hosts on which said VM could possibly execute. In effect, every physical host needs to have connectivity to every network any of the VMs could use. Furthermore, it is necessary for the underlying network to be able to switch packets destined to a given VM to the physical host on which this VM is currently executing, something that will dynamically change during a VM migration. Addressing some of these obstacles to VM migration in a datacentre network has been the focus of several recent papers, discussed below.

PortLand [17] presents a network architecture that is designed to support host mobility in a datacentre. The major focus of the work in *PortLand* is modifying the underlying L2 protocol (Ethernet) to enable the required L3 mobility features in a virtualized datacentre. They accomplish this by replacing the broadcast behaviour of Ethernet with a centralized directory of Media Access Control (MAC) addresses-to-switchport mappings that are transparently updated by inspecting outgoing packets from each switchport. A side benefit of eliminating Ethernet broadcasting is that it reduces the amount of traffic that hosts connected to the network need to process. The modifications to Ethernet in *PortLand* are not just constrained to supporting IP mobility; the centralized directory is responsible not only for maintaining a MAC-to-port mapping but also detecting failures in the network fabric and switching the selected path to prevent a loss of connectivity.

Floodless in SEATTLE [15] is another paper that explores the limitations of the Ethernet protocol in the context of supporting IP mobility for VMs. Much like *PortLand*, *SEATTLE* removes the broadcast-address lookup component with something considerably more scalable; however, in contrast to *PortLand*, it proposes collapsing all the L3 subnets into a single broadcast domain which is run across all switches in the network. While a single broadcast domain would be infeasible if broadcasting was used, the switches are modified to use a distributed hash table (DHT) to perform L2 address resolutions *in lieu* of the traditional broadcast-based solutions, removing one of the major impediments to large subnets in a datacentre. With all of the VM IPs now on the same subnet, it is possible to relocate a VM anywhere in the datacentre without worrying about how the switching infrastructure will interact with the routing infrastructure — an L3 device can sit on the edge of the datacentre and let the switching infrastructure take care of the rest.

Supporting IP continuity in a datacentre network requires modifying the underlying transport protocol, and while different approaches are taken they all remove the broadcasting component from Ethernet and replace it with a centralized data store that can be updated dynamically when the structure of the network changes. This is unfortunately something which is not feasible to replicate directly in an encapsulation system because most existing encapsulation systems are built under the assumption that the DN performs both L2 and L3 functions, making it infeasible to have multiple DNs share a logical L2

address space. However, the approach can be moved one layer up the stack, and instead of having a centralized directory of MAC-to-switchport mappings each DN could write to a centralized directory of IP-to-DN mappings, and allow the DN to use its existing mechanism to map IPs to encapsulation links. As long as the directory is updated when IPs move DNs, IP continuity will exist for the ENs.

2.4 Software Routing

Traditional IP routers are built using custom silicon that embeds forwarding logic into hardware; while capable of forwarding at very high rates (over 10 Gbps) they are expensive both to build and deploy, and are unable to support protocol changes easily. There have been various attempts to address these limitations by using a commodity machine running a commodity operating system (typically Linux), yet the performance of a commodity platform has never been able to match that of the custom silicon found in dedicated routers. The performance issue has resulted in software routers being relegated to use in branch offices and development labs, but not in major datacentres. Indeed, most encapsulation systems today that include routing functionality are built into dedicated routers, and are not typically deployed on general-purpose computers [22].

Recently, the field of software routing has experienced an increase in interest due to several different but related factors. First, advancements in both processor architectures and network-interface card designs have dramatically increased the rate at which general-purpose machines are able to process network traffic, addressing one of the largest barriers that software routers traditionally faced. The advancements in processor architectures has not just increased the speed and number of cores that can exist on a chip; operating-system virtualization has become a popular way to increase datacentre utilization while also increasing the uptime and flexibility of the overall system. Features like virtual-machine migration and hot failover of virtual machines have introduced changes to the way datacentre networks are being designed and operated, yet the underlying network technology has not seen a similar rate of change. It is difficult to determine if this is because such changes are not necessary, or if the cost of the existing infrastructure is preventing

innovation from extending to the datacentre networking field. The difficulty of adding features to dedicated routers has been an impediment to deploying new capabilities to virtualized datacentres; motivating further interest in adopting software routing technology due to the simplicity of updating software routers.

The authors of *Towards high performance virtual routers on commodity hardware* [8] explored the limits of running a software router on a commodity server, and discovered that the memory subsystem is the source of bottlenecks in a typical mid-range multicore server system performing software routing. They were able to achieve forwarding rates of approximately 7-million packets per second for 40-byte packets, after carefully mapping routing processes to cores. They also explored the impact of running software routers in virtual machines; virtualization technology was discovered to have a significant impact on the obtainable forwarding rates, with a virtualized instance achieving between 1 and 2 million 40-byte packets per second in the best case. Given that decapsulation nodes are, fundamentally, software routers with additional logic, this provides an important practical upper limitation on the performance which each DN can be expected to achieve. This paper also explores the impact of different approaches to sharing physical network interfaces with virtualized routers; when the network card is able to perform packet de-multiplexing and assign each virtualized router an independent packet queue on the NIC it is possible to achieve much better isolation and fairness metrics as compared to having the hypervisor handle all packet processing itself. This is unsurprising but an important factor to consider when building virtualized routers — the capabilities of the underlying hardware can have a major impact on the performance that each routing instance is capable of achieving.

The issue of scaling software routers running on commodity hardware to multi-gigabit rates is examined in *Can software routers scale?* [2], where the authors examine the impact that the architecture of commodity hardware has on the potential performance achievable by a software router. They propose an architecture not dissimilar to the one proposed in this thesis whereby a collection of routers are clustered together to achieve higher throughput rates than would be otherwise achieved by a single machine; however, they focus on building a single logical router from multiple machines which are interconnected using a Valient load-balancing mesh. The key observation underlying their paper is that one of the

most basic limitations of existing software routers is that they focus on the 'single server as router' architecture, yet this is fundamentally unable to match the performance of custom hardware; therefore, future software routers will need to be built using a clustered architecture to obtain speeds comparable to custom hardware using a software router. The programmability of a software router is also highlighted as a key advantage of this approach; it becomes easier and less expensive to deploy new capabilities to existing routers.

ViAggre [3] presents an architecture by which the growth of the Default Free Zone (DFZ) is addressed by splitting the global routing table onto a collection of routers through the use of virtual network prefixes. This is, in effect, attempting to solve the same problem as *Towards a new Internet Routing Architecture* [30], large routing tables due to increases in PI addressing, yet instead of attempting to solve the source of the problem it instead proposes a way to hide the impact of the issue. The use of virtual network prefixes makes it possible to have large swaths of relatively unused address space handled by a single router, while allowing the more popular subnets to be handled by dedicated machines, and dynamically shifting the assignment of prefixes to routers can allow for dynamic load balancing across the routing cluster. This is similar to the approach proposed in *Can software routers scale?* [2] whereby less capable devices are shown to have a higher aggregate throughput by splitting the load across a cluster of machines. Splitting the routing table into components that are loaded onto different routers is similar to the approach presented in this thesis takes to separate load across the different decapsulators in the system, where the ingress router knows how to route specific packets to the responsible decapsulator.

The work in *Can software routers scale?* [2] is extended in *RouteBricks: Exploiting parallelism to scale software routers* [7], where the authors implement the proposed software-router architecture and validate its performance using several real-world benchmarks. The prototype is shown to be capable of forwarding at a rate of 35 Gbps using a 4-server cluster; an impressive demonstration that modern commodity machines are capable of forwarding at rates that used to be exclusively obtainable with customized hardware. The implementation resulted in several important results; first, the Central Processing Unit (CPU) architecture plays an important role in achieving high throughputs — the parallel memory bus present in the Intel Nehalem architecture was required to achieve the required forward-

ing performance, and shared-memory-bus CPUs were incapable of obtaining a comparable level of performance. Second, it was necessary to use multi-packet queue network cards to avoid queue-CPU core contention. Finally, it was necessary to pin each CPU core to a specific network-card packet queue to avoid cross-core queue locking.

The encapsulation system presented in this thesis is not expected to be deployed in environments that require extremely high packet-forwarding performance, because the clients are expected to be connecting to the server over the Internet, and most datacentres are connected to the Internet using links that have a capacity in the small single digits of Gigabits per second. Given that others have demonstrated systems that are capable of pushing much higher rates than the system proposed in this thesis is expected to handle using a small cluster of commodity machines, it validates the feasibility of the system running as a software router on commodity hardware.

2.5 Handling Decapsulator Failure

The software reliability field has two perspectives on the development of fault-tolerant software — perform extensive testing on all possible failure cases to ensure that the software never crashes, or build in the assumption that software will fail and ensure that failure handling does not break the system. The second approach is much easier to implement and validate in complex software, especially when safety is not a concern. There are many variations on this general approach to building software, but two major approaches are to detect and try to handle any errors that occur or to crash whenever an error is detected. Software that implements the second approach is called 'crash only' software, and this is the approach that more and more highly reliable, highly parallel solutions are adopting [5] [13] [6] [10].

Traditionally the networking field has not adopted the 'crash only' approach because it is not terribly difficult to build mostly reliable software on reliable hardware, and with custom hardware forming the basis for most networking gear it is expected that hardware failures will be relatively rare. Indeed, for the few cases when failure is not tolerated, specialized failover approaches have been developed, yet these are often complex and in-

flexible. As the networking field begins to utilize software routers running on commodity hardware, it is important to recognize the intrinsic differences between the two approaches and the additional work required for systems running on commodity hardware to offer similar reliability guarantees.

Integrating a 'crash only' approach to an existing encapsulation system is greatly simplified when IP continuity is provided to ENs during a failure — a small process can monitor the state of each DN and forcibly kill it if required. Clients that are written assuming a DN can go down at any time will simply switch to an alternate DN when the primary stops, either due to a failure of the software or the hardware. Minus some small packet loss, the EN does not experience any problems because the IP address migrates with the client to the new DN.

Chapter 3

Design

A reliable encapsulation endpoint architecture does not exist in a vacuum; it is built on top of existing encapsulation systems, and therefore the design will be implemented differently depending on the specifics of the particular encapsulation system in question. The high-level architecture of the encapsulation endpoint reliability architecture is discussed here in relation to the previously-discussed standard encapsulation architecture, and how IP continuity is provided for that architecture. A specific implementation is described in Chapter 4.

The encapsulation endpoint reliability architecture proposed in this thesis, in contrast to a traditional encapsulation architecture, has four components; the encapsulation node (EN), the decapsulation node (DN), the decapsulation node failure detector (DN-FD), and the edge service router (ESR). In this architecture the EN and DN occupy the same roles that they do in the traditional encapsulation architecture, with some minor differences. The ESR acts to ensure that the IP address assigned to an EN can be migrated between DNs when a failure occurs by maintaining a system-wide EN-to-DN map. The DN-FD acts to detect failures in the DN, and kills nodes acting in an erroneous manner.

One difference between our architecture and a typical encapsulation architecture is that the failover DNs are not run on specialized hardware; instead commodity hardware is used. While typical encapsulation systems may deploy DNs on commodity hardware, failover-capable versions of these DNs tend to be deployed on custom hardware. Another

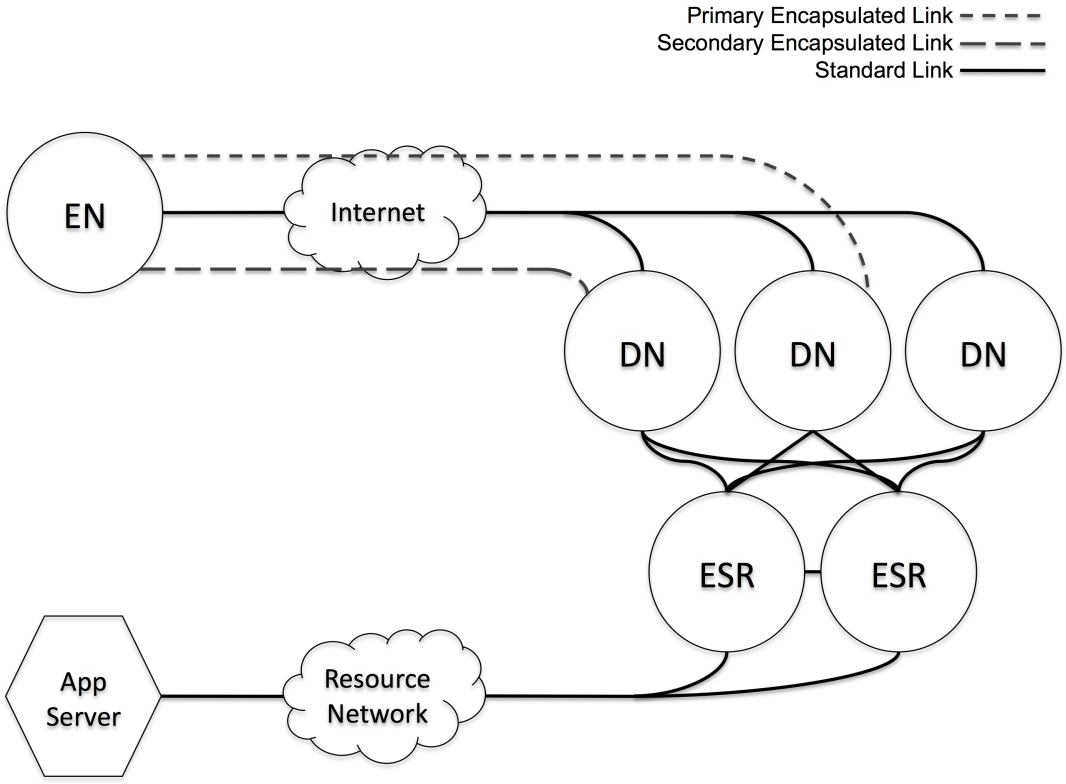

Figure 3.1: Failover Encapsulation Architecture

difference is that instead of having the EN connect only to a single DN, the EN connects to multiple (n, n >1) DNs, where n is one more than the greatest number of concurrent DN failures that are to be supported. The first DN to which the EN connects is responsible for sending configuration settings, including IP address, to the EN — this DN also becomes the initial destination for all client traffic. During the connection setup process with the additional n-1 DNs, the EN supplies its configuration (including assigned IP address) and marks the DN as 'secondary'. Being marked as 'secondary' does not cause the DN to treat the EN any differently than it would if it were primary; it simply maintains the session in preparation for a failover situation.

The architecture providing failover capabilities to an encapsulation system is shown in Figure 3.1. Each of the DNs, in addition to being connected to the Internet (to terminate

EN connections), is connected to the Edge Service Router (ESR) using whatever version of IP the encapsulation system supports (IPv4 or IPv6). The network connected to the ESR is whatever the intended destination for this encapsulation system is; possibilities include a private IP network, the public IPv6 Internet, or the public IPv4 Internet. The version of IP running on this network matches the protocol type used to determine the IP addresses assigned to the EN. When the primary DN assigns an IP address to the EN, it does so out of its own address pool, and the ESR is configured with the address ranges assigned to each DN. When the DN receives a packet from the EN, it decapsulates the packet and injects it into the network where it will be routed through the ESR before traveling to its final destination. Packets returning from an end host pass through the ESR, where it routes the packet to the proper DN, at which point it follows the standard encapsulation protocol for returning to the EN.

Every DN is configured with two classes of subnets, primary and secondary, instead of being configured with a single subnet. A primary subnet is used by a primary DN to assign addresses to clients that are not yet connected to a DN. A secondary subnet is one that contains part of a primary subnet on a different DN, and reflects which ENs will fail-over to the DN should its primary fail — this is used to reduce the potential impact of churn on the routing table during DN failures. Additional details on primary and secondary subnets can be found in Section 4.4.

Since ENs will migrate between DNs as a result of DN failure, it is necessary for the ESR to have an up-to-date view of which EN is connected to which DN. Establishing such a mapping of ENs to DNs is analogous to creating and maintaining a routing table; each EN acts as an end host and each DN acts as a packet router. This association table is referred to as the EN-DN Mapping Table; further, it must act as a reliable data store to ensure that the failure of any DN or ESR does not cause an interruption in service. Given the similarity in functionality to a routing table, the EN-DN Mapping Table could be implemented using one of several common dynamic routing protocols, including Open Shortest Path First (OSPF) or Routing Information Protocol (RIP). An alternative approach would be to utilize an existing reliable shared data store, such as the Google Chubby Lock Service [4], to persist the mapping table. The use of Chubby to store small pieces of identification information

(such as host name to IP mappings) has been demonstrated by Google; such a service could feasibly store the EN-DN Mapping Table reliably as well. Both the DN and ESR read and write to the EN-DN Mapping Table; the exact method that they use to access the table depends upon the specific mapping table implementation; however, all implementations will require a component running on each DN and each ESR. When choosing how to store the EN-DN Mapping Table it is necessary to consider how both the DN and ESR will interact with the table; using an existing routing protocol would allow existing dedicated hardware to fulfill the ESR role; using a custom software solution like Chubby would preclude this approach.

The DN-FD is an independently developed piece of software that runs on each DN and monitors the state of the system to detect component failures (both hardware and software). This piece of software is independent of the specific encapsulation system; by separating the encapsulation system from the failure detector it is possible to build a small, well-tested, and robust failure detector that can be utilized across different encapsulation implementations. The design and implementation of the DN-FD can leverage much of the research that has already been done in this field; indeed it could be feasible to allow the DN to signal error conditions directly to the DN-FD rather than just relying on the DN-FD to detect all possible failures. Upon detecting a failure or inferring the presence of a failure, the DN-FD forcibly terminates the DN software and restarts it after a configurable period of time. The amount of time that will elapse before restarting the DN will depend on the number of previous failures, the load on the system, and other detected hardware issues. As long as one of the secondary DNs is still working, the system will continue functioning as before (ignoring the packet loss that will occur before the client realizes that the DN has failed), ensuring reliability as long as one secondary DN is still available.

3.1 DN Failover

When the primary DN fails, the EN selects one of its secondary DNs to take over, and begins sending all of its traffic to this DN. Given that the DN had already been initialized, there is no delay in establishing a handshake with the DN, performing authentication, or

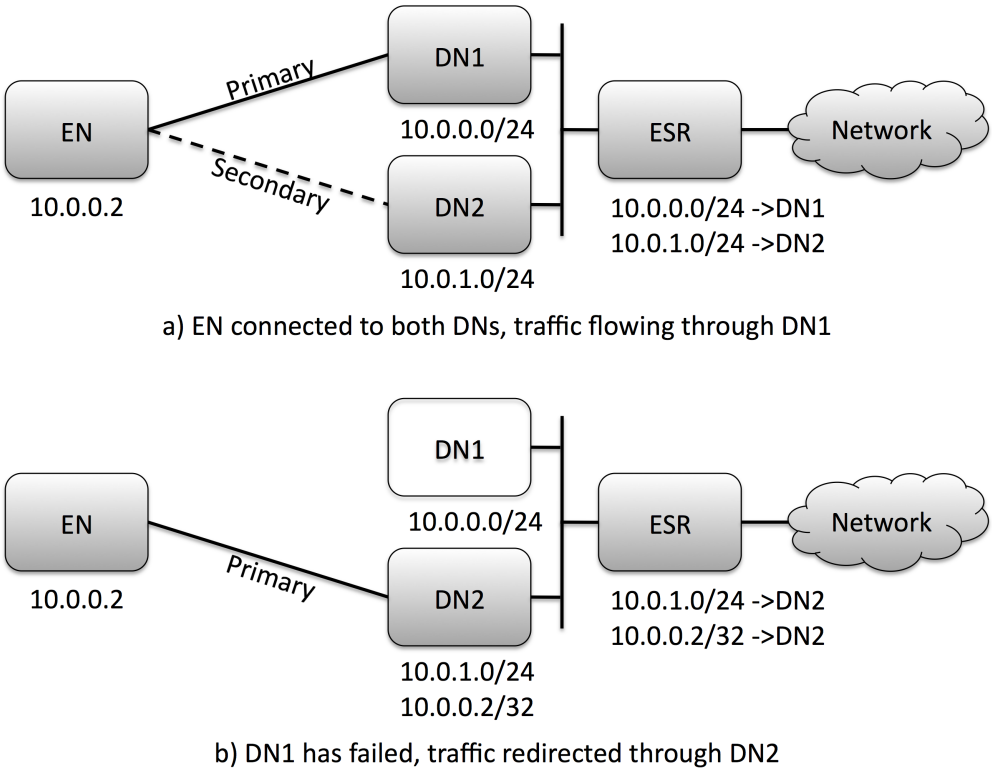

Figure 3.2: System Prior to (a) and Following (b) Failure of a DN

any other initial connection tasks specific to a particular encapsulation implementation. When a secondary DN begins receiving traffic from the EN, it marks itself as the primary for this EN, and updates the EN-DN Mapping Table to inform all other DNs and ESRs that it is now responsible for packets destined to the specific EN. Figure 3.2 shows a simple example of a single EN and two DNs that experience a DN failure. Here, the system is functioning normally in Figure 3.2 (a), and once DN_1 has failed Figure 3.2 (b) shows the result of the EN failing over, with DN_2 adding the EN host address into the EN-DN Mapping Table.

At this point, all traffic from the client is properly redirected back to the EN over one of the secondary DNs without a substantial interruption of service. While some packets are lost, including all outstanding packets in transit to the primary DN along with any

packets returning to the DN from the destination server before the routing rule is applied, this likely constitutes a small fraction of all the packets that are transmitted during the life of a typical encapsulation session. This can be partially remedied by performing buffering on the EN if packet loss is found to be a substantial issue; this possibility has not been explored further here as transport-level retransmissions were considered sufficient.

Handling DN recovery has two elements; how the restarted DN handles new clients, and how ENs that previously used the DN are migrated back. When a DN first starts, it reads the EN-DN Mapping Table and notes any addresses from its address range that are currently in use by other DNs. It then ensures that it does not assign any of these addresses to newly connecting clients. When a DN handling clients that are not part of its primary subnet notices that the DN for this subnet is alive, it informs the EN that the primary DN is available again. The EN uses this information to establish a new session with the DN and once it has completed the initialization routine it marks the original DN as primary again and informs the secondary DN. The secondary DN, upon receiving this message, removes any information it had in the EN-DN Mapping Table, allowing the default rule to again take effect and have the client resume communication through the primary DN. Cleaning up the mapping table is important; if default rules are not relied upon it is possible for the table to grow in an unwieldy manner as DNs hosting secondary subnets fail and need to be further split.

3.2 Detecting Failures

The types of failures that the DN-FD needs to be capable of handling depends on the expected failure model of the system; the interaction between the DN-FD and the system is shown in Figure 3.3. Here it is assumed that the failures experienced by both the DN and the hardware that the DN is executing on are classified as fail-stop; unexpected states cause the DN software to crash, while hardware issues cause the entire system to go down. Should the DN-FD detect packet corruption, memory corruption, or any other 'soft' failures then it will kill the system, enabling a fail-stop model to be used to describe all classes of failures that the system experiences. The DN-FD should be liberal in what it

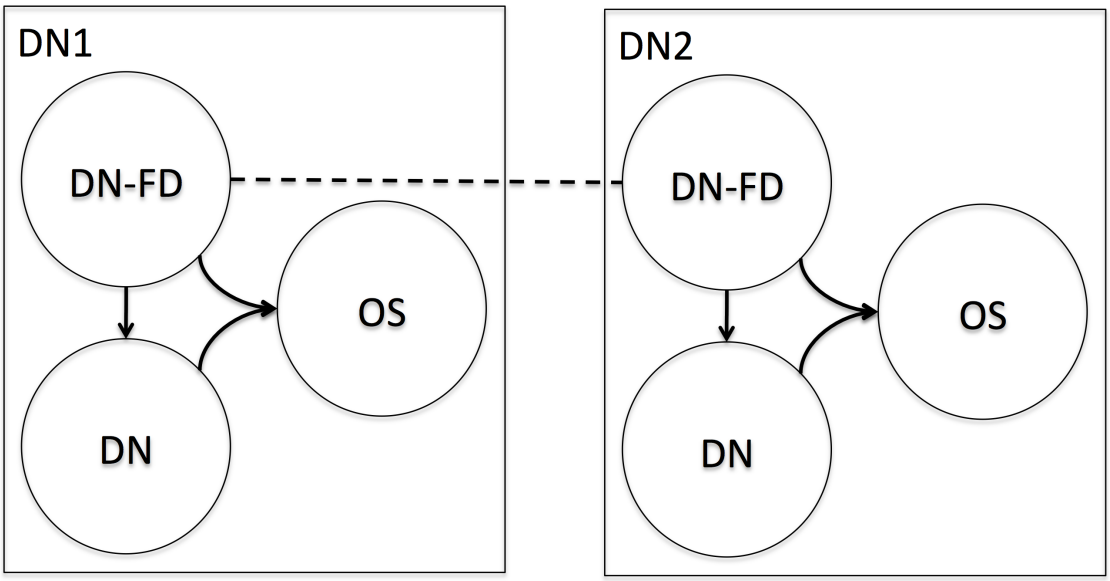

Figure 3.3: DN-FD Architecture

determines to be a failure state — it is desirable to have a failure detector that is capable of identifying every error condition even if it has a non-zero false positive detection rate. As long as the false negative rate is zero the system will operate without issue; bringing the false positive rate down is merely an efficiency issue. This is preferable to the system experiencing a correctness issue by having a non-zero false negative rate.

The EN is able to detect a DN failure using one of two approaches: detecting the DN failure itself, or having the secondary DN notify it that the primary has failed. The former approach can signal an error relatively quickly when failures are isolated to the DN software; the DN host returns a Port Unreachable message to the EN as soon as data is sent. The latter approach requires additional complexity in the DN cluster, however it results in a faster failover when the DN host fails because the EN is not stuck waiting for a host that is no longer available to respond. A basic monitoring strategy would be for every secondary DN to monitor all of the DNs that its connected ENs are using as the primary. If an error occurs, all of the secondary DNs would then notify the EN of the failure. This approach has room for optimization, both from the perspective of how the EN is notified of a failure and the amount of processing that each DN is required to perform to ensure

the EN is informed of a failure quickly.

The distributed nature of the described encapsulation system adds many different sources of failures to the system. There are three major classes of failures that can be encountered: network failures, hardware failures, and software failures. Network failures are considered last; first the impact that hardware and software failures have on the behaviour of the system is explored.

The use of commodity hardware as the underlying platform for the DNs exposes the system to many hardware failures; cost is often the primary concern when building commodity systems and reliability is often sacrificed to meet this goal (error-correcting RAM is rarely used, as one small example). Hardware failures on a given DN can be detected by the DN-FDs running on the other DNs in the cluster; given that all the DNs are well-connected, there is minimal overhead in having a sub-second heartbeat shared by each DN-FD. It is difficult for ENs to rapidly detect a DN failure caused by hardware failure; connections do not immediately time out and there is also the path latency between EN and DN slowing things down. Having each DN monitor the status of all other DNs enables a secondary DN to inform all of its ENs that their primary has failed and they should initiate a failover.

The implementation complexity of typical encapsulation software virtually guarantees that there will be failures caused by the DN software; it is the primary responsibility of the DN-FD to catch these and terminate the DN. It is possible for the DN-FD to monitor not just the DN software but also operating-system behaviour to detect anomalous conditions. As mentioned earlier, these are solved by immediately killing the DN; there is little cost in doing this because of the seamless-failover mechanism.

Rapidly detecting failure of the network is, in contrast, difficult to accomplish in a data-efficient manner due to the difficulty in distinguishing arbitrary packet loss from a failure case. The proposed failover architecture does not provide IP continuity in most network-failure cases, because the DNs and ESR are located in the same logical network location. Loss of connectivity to one DN as a result of a network failure, then, implies that connectivity has been lost to all DNs. Handling DN failure across independent network locations is an area of future work; it is clearly difficult to ensure IP continuity for the

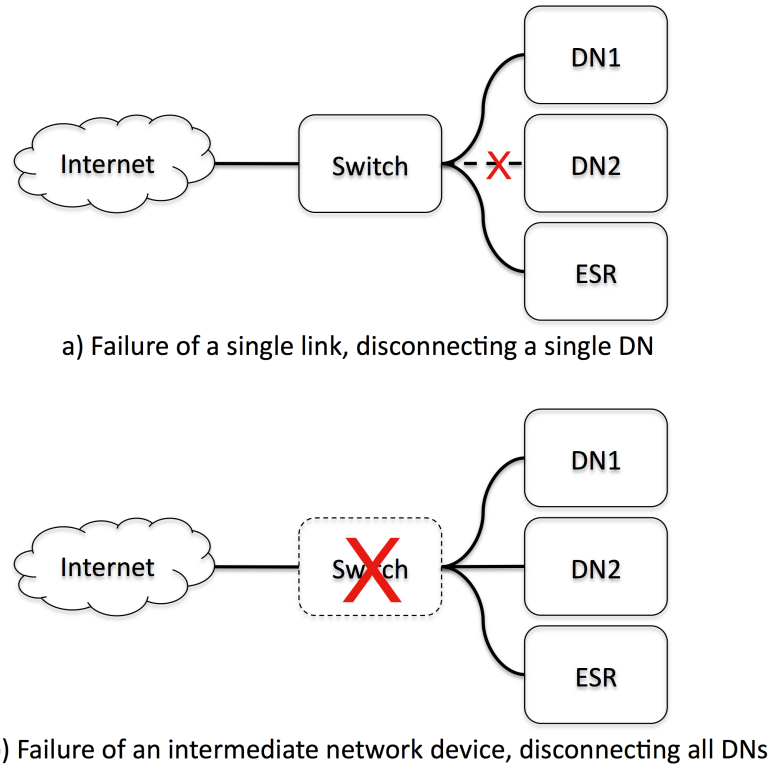

Figure 3.4: Network Failure Scenarios

client across independent network locations. It is assumed that the DN cluster is deployed in a hub-and-spoke fashion, with each DN, ESR and border gateway connected to the subnet through a single L2 link. Thus, network failure either occurs by partitioning a single DN from the network (link failure) as shown in Figure 3.4 (a), or the hub dies and removes connectivity to all DNs, shown in Figure 3.4 (b). It is not expected that multi-node partition scenarios will be encountered. If a link failure occurs within the DN cluster it will be detected in the same manner as DN host failure; the DN-FD running on other DNs will be unable to reach the DN and will mark it as dead.

3.3 ESR Failover

Introducing the ESR component into the architecture has the potential to add another failure point into the system; however, the simplicity of the ESR role makes it easy to support failover using existing protocols. Fundamentally, the role of the ESR is to determine which DN is able to reach a specific network address. It accomplishes this by participating in a routing algorithm in which all of the DNs also participate. It advertises itself as the next hop for all DN-assigned addresses and, as a result, it functions in a similar manner to a peering router in a standard routing architecture.

Accomplishing the above tasks in a failover-capable manner is something that every common enterprise router needs to accomplish; as a result, many standards have been developed to enable this functionality. Running a pair of commodity routers, either hardware or software, that share a virtual IP address using any existing protocol (HSRP [23], VRRP [16], or Common Address Redundancy Protocol (CARP) [25]) will enable a cluster of ESRs to provide fully transparent failover to the DNs without requiring any DN-specific logic. In fact, existing hardware can be re-purposed to accomplish this.

3.4 Analytical Analysis of a Failover

In order to characterize the behaviour of the system during a failure a brief analysis of the operation of the system during a failure has been performed. This analysis will help to provide context to the results in Chapter 5 and provide a brief sanity check to ensure that the measured results are logical. Recovering from a node failure in the system is characterized by three key events which all impact the speed at which a failover will occur. The system is shown in Figure 3.5, where a failure occurs at time T_0. Prior to T_0 the system is behaving properly; traffic is being encapsulated at the EN and is flowing through DN_1. At T_0, the DN software running on DN_1 experiences a failure and is killed. This stops all traffic flowing between the client application and the server.

At time T_1, the EN detects that DN_1 has failed. The exact nature by which the client determines the existence of a failure is not relevant here, as the minimum bound can be

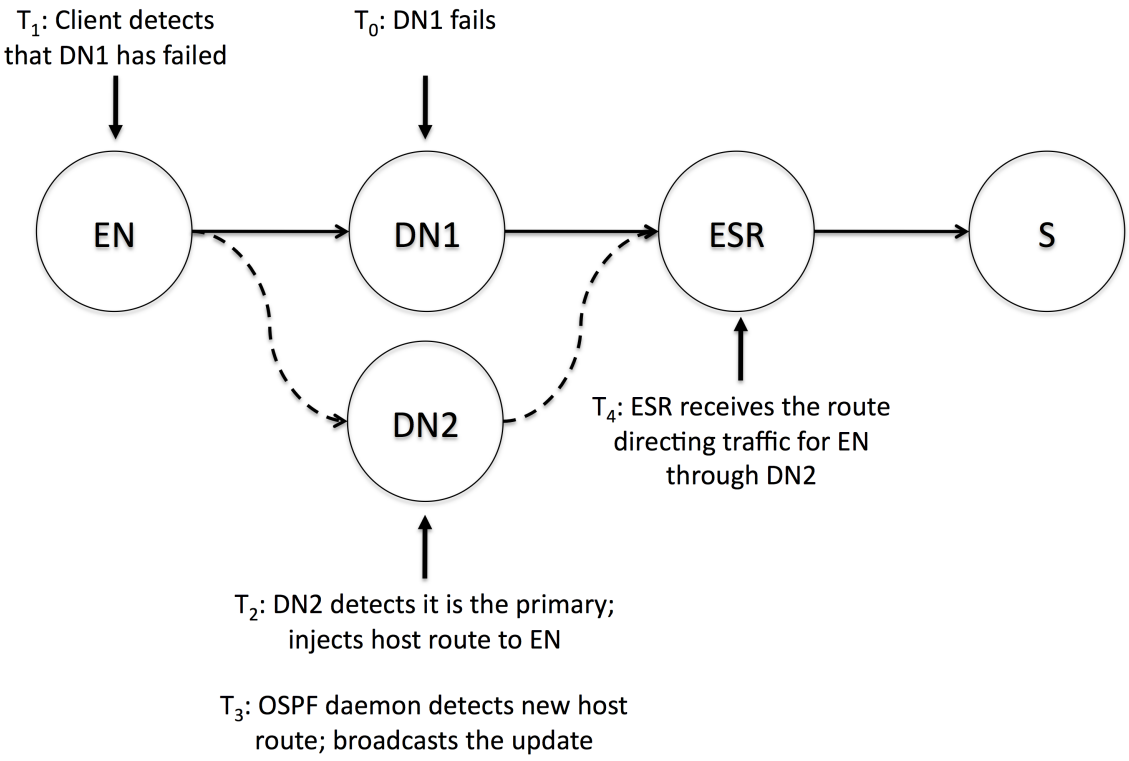

Figure 3.5: Timeline Of System States During Failover

calculated independently of failure detection mechanism. The minimum value of T_1 is:

$$T_1 >= T_0 + \delta + \frac{RTT}{2} \tag{3.1}$$

It will take at least half of a round trip for the notification to travel from the DN to the EN; it will also take δ time for the DN host to to detect that the DN process has died and close the connections previously owned by this process. At time T_1 the EN changes its behaviour and begins directing all of the client-application traffic to DN_2. DN_2 first determines that a failure has occurred at T_2, when it receives a message from the EN. This will occur approximately $\frac{RTT}{2}$ seconds after T_1, assuming that the message is not delayed or lost due to unfavourable network conditions. At time T_2 DN_2 marks itself as

the primary DN for the EN, and injects a route into its local routing table. This means that at T_2 the client application is able to send traffic to the server properly; unfortunately, the server is not able to pass data to the client application, because the reverse route from the server to client has not yet propagated to the ESR.

Once the DN daemon running on DN_2 has injected the EN route update, a variable amount of time will pass before the routing daemon, also running on DN_2, will detect the routing-table change. T_3 is defined as the time when the routing daemon has detected the addition to the routing table, at which point the routing daemon broadcasts this information to the other DNs and the ESR in an update message. The ESR, connected on the same subnet to both DNs, will receive this update message some time after T_3. Once the routing daemon running on the ESR has processed the packet and has injected the route into its routing table, the system will have reached time T_4. T_4 is defined as the time when the system is again functioning properly and traffic can flow bidirectionally between the client application and the server.

One possible, but somewhat complex, optimization that can be made to reduce the overall time to complete a failover involves DN_2 detecting the failure of DN_1 itself. This would allow the time between T_3 and T_4 to proceed in parallel with the time between T_1 and T_2, reducing the failover time to the maximum of these two time intervals. Given that the typical latency between a client and server running over the Internet is in the range of tens or hundreds of milliseconds, having DN_2 mark itself as primary before the EN has detected the failure would result in a decrease in the failover time observed by the EN. This optimization is safe even if DN_1 has not died — traffic originating from the EN will continue to pass through DN_1, while traffic returning to the EN will pass through DN_2. This asynchronous routing situation is not ideal; however, traffic is still able to flow in both directions because both links are available.

In addition to understanding the sources of delay, it is important to understand the characteristics of each delay source to help improve the understanding of why certain results are observed in the experimental results. The differences between T_0 and T_1 and T_1 and T_2 are likely to be somewhat static regardless of the test, as the RTT in a typical environment is likely to be relatively stable. The difference between T_2 and T_3 is much more variable,

however, as the routing daemon polls the routing table on a fixed interval, yet the EN route can be injected at any point in this interval. The scanning interval configured on our system was 1 second, the smallest value supported by the *bird* routing daemon. Finally, the time between T_3 and T_4 is going to depend upon the behaviour of the DN/ESR network and the current packet loss rate, as the multicast delivery mechanism does not guarantee that a routing update be received by all the other nodes. With a presumed low loss rate, and a high-speed DN/ESR network, this should be quite low, on the order of single-digit milliseconds.

Chapter 4

Implementation

The proposed endpoint reliability architecture has been implemented within the context of an existing encapsulation system named *Accoriem*. This encapsulation system was designed to provide mobility and aggregation services to multi-interface mobile devices and uses a combination of a TCP-based control channel and a UDP-based data channel that runs on each available physical interface, enabling potential bandwidth aggregation, seamless handoffs between networks, and remote manageability of the device. This system is believed to be a reasonable example of a real-world encapsulation system — it maintains multiple independent connections between the client and the server and requires both transport and application-level state to be maintained between endpoints. It was not developed simply for this thesis — it is a system that existed prior to this thesis and has seen real-world adoption. Implementing the encapsulation endpoint reliability architecture within the *Accoriem* system allows for an EN to function seamlessly in the presence of DN failures.

4.1 Accoriem

To provide some context to the discussion of the failover implementation, a brief description of the standard *Accoriem* architecture is provided. Figure 4.1 shows the primary components of the system involved in establishing a connection from the EN to the DN and then

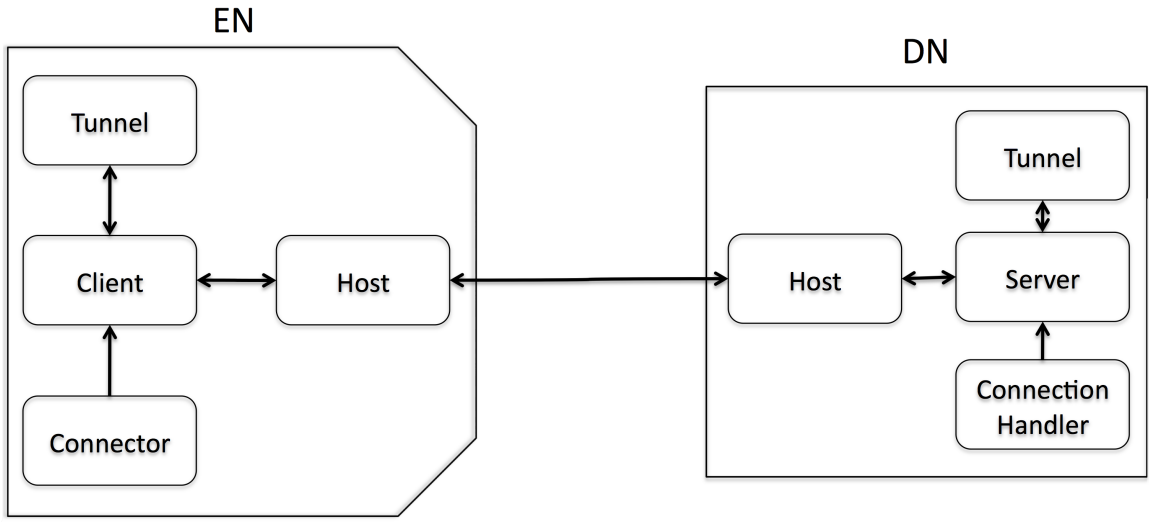

Figure 4.1: Accoriem System Components

passing encapsulating application traffic between the endpoints. A detailed description of the *Accoriem* system can be found in *Practical Multi-Interface Network Access for Mobile Devices* [20]; however, it provides substantially more detail than is required here.

The EN is composed of a series of modules that are responsible for encapsulating and decapsulating traffic and maintaining connection with the DN. As shown in Figure 4.1, there are four core modules in the EN. The `tunnel` module is responsible for handling the virtual-tunnel interface, including reading and writing IP packets entering and leaving the system and assigning the virtual IP address to the interface. The `connector` module is responsible for detecting physical-interface state changes and connecting to the DN when a new interface is available. This module receives configuration state from the DN, including the virtual IP address. Once the connection has been established the link is passed to the `host` module. The `host` module is responsible for maintaining the links with the DN, sending and receiving packets over the appropriate interface, detecting DN loss, and other similar tasks. The `client` module acts as a go-between for the other modules; it does not have a major role in the base *Accoriem* system.

The DN is built in a similar manner, both the `tunnel` and `host` modules are the same as in the EN because they perform the same roles regardless of which side of the link they

are on. The `connection handler` module handles incoming connection requests from new clients, and is responsible for getting the client into a configured state before the EN is ready to pass traffic; once the host has been configured, the connection is passed to the `server` module. The `server` module is responsible for maintaining the collection of `host` objects; one for each connected EN. The `server` is more complex than the equivalent `client` module, however, as packets received by the `tunnel` module are not all destined for the same EN. In the EN case, all received packets are destined for the DN, so they are all passed to the same `host`. The `server` module maintains a lookup table of `host` objects indexed by their virtual IP address, allowing packets received by the `tunnel` to be passed quickly to the appropriate EN. This lookup mechanism is also used to ensure that new connections are passed from the `connection handler` to the correct `host`.

4.2 EN and DN Modifications

A high-level architecture of both the failure-tolerant EN and DN can be found in Figure 4.2; the modules with a gray background required modifications to support DN failover. It is shown that changes were required on both the EN and DN; however, it is notable how isolated the required changes were. The existing data path was essentially untouched, suggesting that the addition of DN failover capabilities should not change the overhead or delay introduced by the system during normal operation.

Changes to the EN involved two major components: the connection-initiation and connection-management modules. The connection-initiation module required two additions: detecting the existence of the backup DNs and sending additional configuration parameters from the EN to the backup DNs. Given that each set of connections to a DN is independent from all other DNs, it was possible simply to introduce a collection of DNs to connect to, and duplicate the existing connection initiator so that each one properly connected to its respective DN. Initially choosing the primary DN is simple — the first one to successfully connect is the primary. This not only ensures that the client is connected as quickly as possible, it also simplifies the connection process by having a generic connection approach (rather than different behaviours for the primary and secondary links).

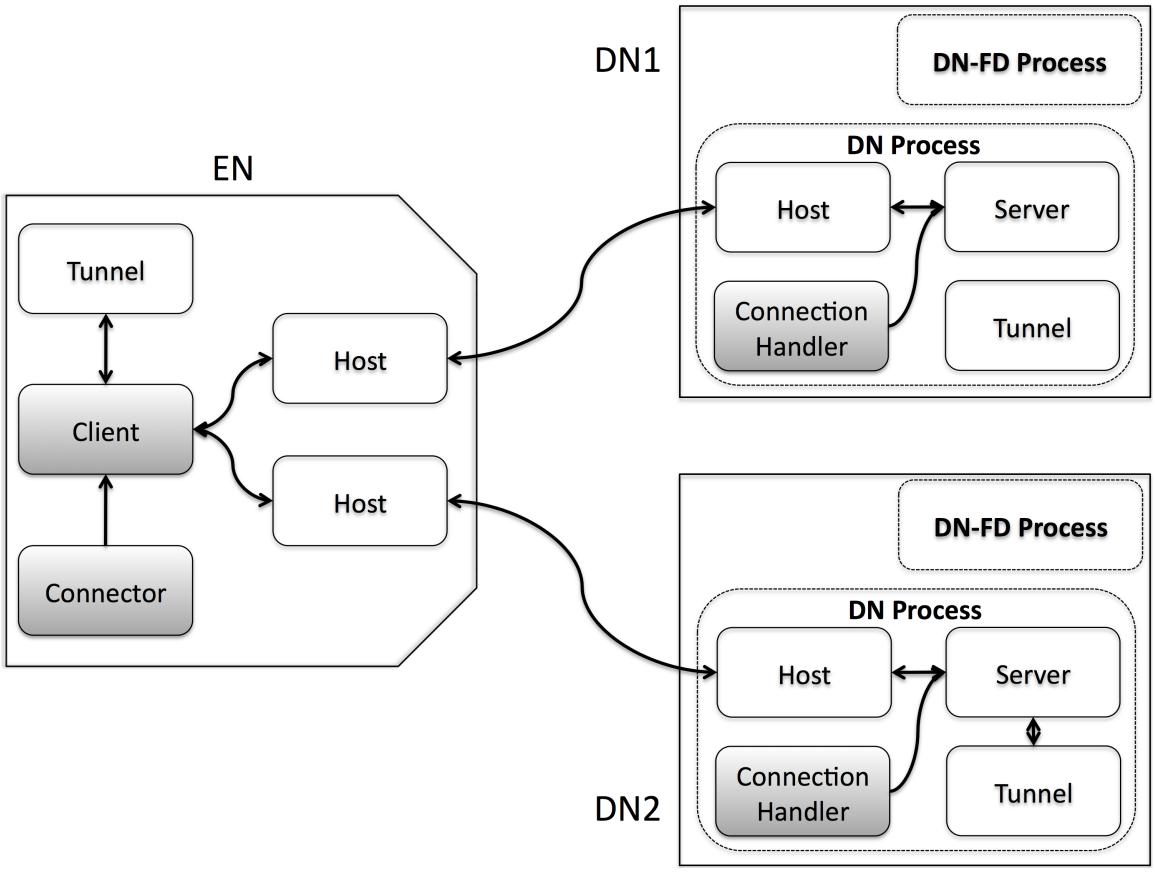

Figure 4.2: Accoriem Components: Modified Modules shown in Gray

Sending additional configuration parameters just extended the existing post-connection configuration logic to inform the backup DNs of the encapsulated IP address of the given host.

The EN-side changes to the connection-management module primarily involved replicating the existing connection management and DN control logic to support multiple DNs. The `host` module was initially responsible for transferring commands between the EN and DN, determining how to split traffic between the existing interfaces, and managing link information (latency, packet-loss rate, etc.). Rather than complicate the existing link-management code an additional layer of abstraction was added in the form of the `client` module, which receives packets to be encapsulated from the `tunnel` and passes them to the `host` for transmission to the DN. This was also done to avoid complicating the design

by introducing additional failure-tolerant code into the `host` module. Instead of handling a single `host`, the `client` maintains a collection of `host` objects; each representing an active DN. When the primary `host` informs the `client` it has disconnected, the `client` chooses another `host` from its collection and begins passing all traffic over this `host`. The DN failure-detection code already existed, and changes to the `client` were constrained to how the `client` received newly created `host` objects from the `connector` and which `host` would be passed traffic from the `tunnel` module.

The DN required changes in two modules: the initial client-connection handler and the packet-processing module. It also required the creation of a new module to handle route injection. The initial client-connection handler was modified to take encapsulator address information from the client on backup DNs, a trivial extension of the existing configuration mechanism. The creation of a module to inject routes was similarly straightforward: by hooking into the Linux kernel *netlink* interface, the process of adding routes took little additional code. The packet-processing module changes were solely to detect that a DN has changed from backup to primary via the arrival of encapsulated packets from the client, in order to trigger an injection of the EN IP address into the DN routing table. Clearly, this adds a bit of additional overhead into the data path (a single additional conditional check is executed for every packet), but it was decided this was preferable to relying on the EN explicitly notifying the DN it has become master. A data packet almost always arrives more quickly at the DN than a control message, making the control message redundant data. The use of control messages was thus deemed to be the suboptimal approach.

As described above, the reliable EN implementation has additional complexity as compared to the unreliable implementation, increasing the probability that the EN itself will experience a failure due to a bug in the code. This is unavoidable given the architecture of the EN; in order to establish a connection properly with multiple DNs and fail over between these connections, it is necessary to modify how the EN maintains this information internally. While the EN is more complex, it is not substantially more complex (compared to the overall complexity of the entire system). Further, the additional logic is exercised whenever a client changes DNs, which is expected to be a relatively frequent operation. This increases the probability of any new bugs being exposed, since the majority of the

changes reside in a code path that is exercised regularly.

Like the EN, the DN implementation that supports the reliable architecture is more complex than the unmodified DN; unlike the EN, however, this complexity does not increase the chances of a failure. This is because the system is designed to detect and handle DN failures; if the modified DN encounters an error then either the DN will kill itself or the DN-FD will detect the failure and kill the DN. In both cases the EN will fail over to a secondary DN and continue passing traffic without substantial interruption.

4.3 EN-DN Mapping Table

One of the new components introduced by the reliability architecture is the EN-DN Mapping Table; a distributed data store that is read by the ESR and both written to and read from by the DNs to ensure that each node in the system knows how to reach every EN. This mapping table is structured much like a routing table, and it was decided to take advantage of this similarity by choosing a dynamic routing protocol to implement the EN-DN Mapping Table in *Accoriem*, specifically OSPF. Utilizing an existing distributed data store reduces the chances of introducing additional software failures, since OSPF implementations are relatively common and used by a large number of other systems. OSPF was chosen over alternatives (such as RIP or IS-IS) due to its straightforward configuration, well-understood behaviour, widespread deployment, and fast re-convergence time. The OSPF protocol maintains connectivity information about every other peer in its area; this information can be used to detect hardware failures in other DNs without requiring additional implementation work in the DN-FD.

Once a primary DN has failed, all of the clients assigned to that DN will migrate to other, functioning DNs, continuing operation with addresses contained within the subnet allocated to the first DN. Upon recovering from the failure, the DN needs to ensure that it does not assign any addresses that are already in use to new clients, otherwise they will never receive any traffic. This is because the existing host routes will override the network route and cause the initial client assigned this address to receive both its normal traffic and the traffic from the newly connected host. The DN, however, is able to determine which

addresses from its subnet are in use by other clients via its copy of the routing table. The routing table on each DN should be synchronized, within the routing propagation time, simplifying this process and making it DN-agnostic.

4.4 Routing-Table Growth

One concern that arose during implementation involved the size of the routing table that would eventually result if many DNs ended up failing and if ENs were not migrated promptly back to their primary DN upon its return to service. There are several different ways to solve this problem — the cleanest but most complex involves re-addressing ENs that have failed over to a secondary DN if it is detected that no connections are currently in use. By readdressing the EN to an address from the secondary DN pool it is possible to remove the lingering route, ensuring that the routing table remains as small as possible. There are two significant issues with this solution — first, detecting that an EN has no running connections; second, there is no guarantee that an EN will ever become ready to be readdressed (that is, an EN may continually send traffic).

An alternative approach that is both simpler and easier to manage is to inject subnet routes instead of host routes by intelligently assigning addresses to clients. Essentially, on startup each DN performs subnetting on the address range it is assigned, based upon the number of other DNs that are available for an EN to fail over to. Then, as clients connect, they are assigned an address from the range that matches with the secondary DN to which they are connected. Thus, when the primary DN fails, each secondary DN only has to specify that all clients in the subnetted address range are reachable though it, instead of each host individually. One benefit that this scheme provides is that as soon as one client detects a failure, the DN can inject the subnet route and ensure that traffic for all ENs that were previously attached to the failed DN begin receiving the traffic they expect (potentially before an EN even notices that the DN has failed). This reduces the visibility of a failure even more than the base system allows for. Secondary DNs can either be configured with the size of the subnet to assign or dynamically calculate the subnet size based upon the number of DNs in the cluster and the addresses that the connected ENs

have been assigned.

4.5 Routing in the DN Cluster

If the ESR is a logically separate device from the default gateway on the DN network, then by default all decapsulated traffic from the clients will be sent to the wrong device (the default gateway instead of the ESR). Preventing this from occurring requires the use of source-based routing rules applied to each DN. The source-based routing rule directs all traffic from the client subnets to the ESR instead of the default gateway, allowing the ESR to process the decapsulated traffic as required. This source-based rule can be statically applied at startup and does not need to change as a result of client failovers (as long as the specified client subnet is large enough to encompass all possible client addresses). This simplifies deployment by allowing the existing default gateway to be utilized. This will not be necessary in situations where different network-layer protocols are used. For example, if the EN connects to the DN over IPv4 but passes IPv6 traffic, then it is only necessary to configure each DN to utilize the ESR as the default gateway for all IPv6 traffic (bypassing source-based routing entirely).

Chapter 5

Experimental Validation

Using the solution implemented above, several micro-benchmarks have been run to validate the proposal in the context of an actual implementation. The micro-benchmarks were run in a setup consisting of five machines as illustrated in Figure 5.1: an EN running the client software, two DNs running the server software, an ESR running standard routing software, and a standard router acting as the Internet. The EN has two interfaces configured on different subnets, the DNs and ESR are all located on a third subnet. All traffic uses IPv4; the ESR performs NAT before forwarding traffic onto the Internet. The EN and DNs are separated by a sub-millisecond-latency link with an available bandwidth of 1 Gbps dedicated to the EN.

The performance of the system was evaluated in three different environments: one built entirely using virtual machines, one built entirely using low-powered commodity boxes, and one with a low-powered commodity client and a server cluster built from virtual machines. The first environment was designed to reflect the possible 'best case' deployment scenario — powerful machines all interconnected using a high-speed backbone. The second environment was designed to show the behaviour on hardware that is truly commodity — this hardware is obtainable by any individual for a lower cost than a typical desktop. The final scenario was designed to represent a reasonably realistic deployment scenario — low-powered clients such as mobile smartphones connected to a typical virtualized data centre.

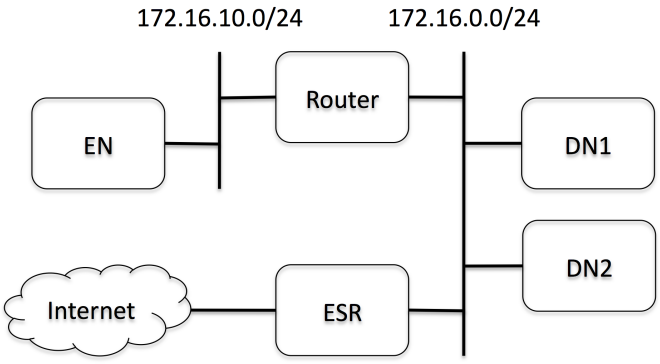

Figure 5.1: Experimental Testbed Setup

It is important to observe that the characteristics of the test network are somewhat different from what would be experienced on the Internet, both in terms of the latency and bandwidth available between the EN and DN. The latency between the EN and DN is sub-millisecond for all three testbeds, yet the latency separating the EN and the DN on the Internet is likely to have be much higher. Conversely, while the bandwidth of the link separating the EN and DN is one Gigabit per second, it is much more likely to be on the order of several Megabits per second on the Internet. The exact latency and bandwidth that exists between the EN and DN is going to depend heavily on the specific deployment scenario; rather than attempt to characterize the performance of the solution for all potential network configurations this thesis presents results from the best case scenario and provides a framework to determine the impact that varying the latency will have on the solution. The major factor that will impact this solution on the Internet is the latency separating the EN and the DN; the minimal amount of additional information transmitted during a failover prevents bandwidth from impacting the speed of a failover.

The OSPF routing protocol is used to synchronize the address state between the DNs and the ESR; on the DN the chosen OSPF implementation was *bird* [9], an open-source routing daemon maintained by CZ.NIC. The choice of *bird* over the better-known *quagga* [27] was made for two reasons: *quagga* has some known stability issues and *bird* is simpler to configure. On the ESR, the OSPF implementation selected was *OpenOSPFd* [26], an

open-source implementation of OSPF developed by the OpenBSD team (and included in the base operating system). All devices were configured as members of the OSPF backbone area, with each daemon set to check the routing table once a second (the lowest configurable interval) for route updates.

All of the ENs and DNs ran Ubuntu 10.04.1 with the 2.6.32-26 kernel. Each system was configured to have TCP metric-saving disabled, to ensure that TCP behaved in the same manner over multiple test runs. The ESR ran OpenBSD 4.7, a commonly used software routing platform, due to its strong support for IP and firewall failover between hosts. The virtual machines were all hosted on a VMware ESXi host consisting of an Intel Xeon E5420 processor with 12 GB of ECC RAM. All virtual machines were configured with the same scheduling priority and each VM was assigned a single virtual processor and 256 MB of RAM. The virtual machine host was not dedicated to the test; other machines were also running, simulating the typical environment in which a virtual cluster would be deployed. The virtual NICs used the standard E1000 driver. The commodity machines all had a 1.2 GHz VIA processor and 1 GB of RAM, and utilized the onboard VIA NIC.

In each failover case it is assumed that a failure had been detected and the DN-FD process forcibly terminated the DN. This is simulated by sending a SIGKILL to the DN process, ensuring any packets in transit were not processed and that the connections were not properly closed. The encapsulation system has been configured to use a UDP data channel; that is, there is no assumption that an encapsulated packet will properly arrive at the DN. The raw test data is available online [19].

5.1 Packet Loss During Failover

For the system to be considered reliable, it is necessary not only to prevent the client from losing connectivity, it is also necessary to ensure that any interruptions in service are minimal, ideally small enough to be indistinguishable from normal network anomalies. To determine the interruption of service experienced during a failover by *Accoriem*, a constant stream of Internet Control Message Protocol (ICMP) packets were sent from the client to a remote server, and the number of packets that were lost when the primary DN failed

was measured. The rate at which the ICMP packets were sent depended on the test; the experiment was run with two rates, one of 10 packets per second (pps) and the other of 100 pps. Each of the packet rates was tested 45 times in each of the three test environments, yielding a total of 270 data points to characterize the behaviour of the system. In every case packets were lost consecutively — once the EN started receiving packets again, all subsequent packets were received.

Data on the loss rate experienced by clients sending at rates higher than 100 pps was not used because the different test environments had very different maximum packet transmission rates — when the server cluster was hosted on virtual machines it was possible to send packets at a rate of close to 1000 pps, yet when the server cluster was hosted on physical machines a rate of only ~215 pps was achieved. This difference was observed to be the result of the relatively high CPU requirements of the encapsulation system; the physical machine CPUs were hitting 100% utilization around 215 pps while the virtual machines did not reach 100% CPU until around 1000 pps. The tests were conducted using the ICMP protocol instead of TCP to avoid introducing retransmissions into the measurement — sending both retransmitted packets in addition to new packets could introduce additional packet delays and potentially reordering — the impact that failure has on TCP streams is examined in Section 5.2.

The first experiment, sending packets at a rate of 100 packets per second, has results that contain several key observations about the operation of the system. First, the behaviour of the system is quite similar regardless of the test environment, second, there exists a component in the system that causes a variable but bounded number of packets to be dropped, third, the majority of test runs on all platforms experienced approximately 1 second or less of loss. These results are summarized in Figure 5.2. For clarity, three outlying values were omitted from the figure; the Virtual/Virtual (V/V) environment lost approximately 250 packets twice, and the Physical/Virtual (P/V) environment lost approximately 430 packets once.

The results show that the three environments had very similar behaviour, with the majority of measurements showing between 20 and 110 packets lost. This corresponds to a loss of packets lasting between $\frac{1}{5}^{th}$ of a second and 1 second. As was discussed in

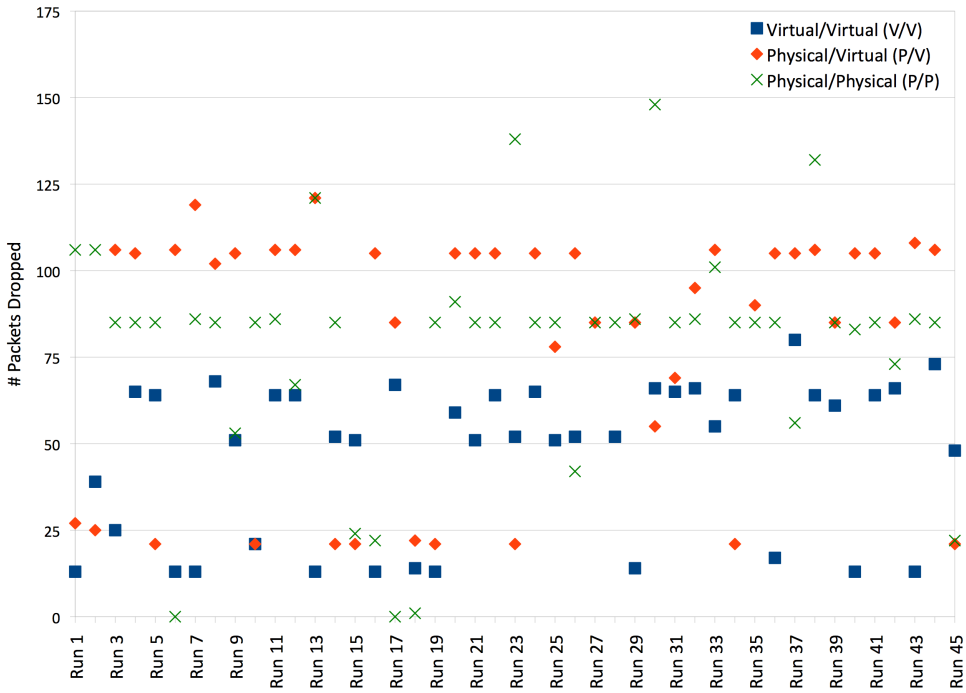

Figure 5.2: Packet Loss Count During A Failover (Rate of 100 Packets Per Second)

Section 3.4 there are three sources of potential delay in the failover process, and given that several experiments showed no packet loss it is possible to have all three of these sources of delay equal zero, resulting in a seamless failover. This occurs when the round trip time between the EN and DN is quite small, and the time between the injection of the route in the DN software occurs immediately before the routing daemon running on the DN polls the system routing table for updates. One or more of these sources of delay are not always zero, as the majority of tests experienced some packet loss. This is quite likely the result of the time it takes the routing daemon to scan the table to detect the EN route, as the number of packets lost is on the same order of magnitude as the scanning time. The impact of RTT on the results is evident in the differences between the V/V environment and the P/V and Physical/Physical (P/P) environments — when using virtualization the

cross-node latency is effectively 0 as the network link is essentially just a shared section of memory on the virtual machine host. In both environments that had physical latency to contend with the packet loss times were slightly longer, shifting all the results up by a constant factor (the RTT).

Considering other sources of delay in the system also suggests that the OSPF routing daemon behaviour is not the only component contributing to the observed packet loss behaviour. It is unlikely that either the EN and DN themselves are contributing to this behaviour — neither component contains a non-uniform logic path. The DN code is always the same regardless of a failure (the DN datapath was not modified to support failover), and the EN codepath simply detects a failure and starts writing to the same datapath contained in a different logical object. There is no reason that any of these elements would have a consistently non-uniform runtime. It is expected that both the operating system scheduler and virtual machine scheduler will schedule the DN process differently on each test run, however the similarity of the results across testbeds running on vastly different hardware suggests these scheduling differences do not have a significant impact on the system performance. While the OSPF daemon was configured to scan the routing table once every second, it is quite likely that this polling did not happen in an exactly uniform manner — network software often adds random delays in between periodic events like scanning to avoid potential conflicts with other nodes. One additional factor that likely introduced some variability into the results is the amount of time it took the operating system to detect that the DN process had been killed and subsequently close the TCP connection. This would impact the time between T_1 and T_2 independent of the RTT, however it was difficult to precisely measure the impact this factor had.

It is not quite as easy as just attributing the different packet loss rates to the routing daemon, however, as the results in Figure 5.2 shows that each test environment has results which tend to cluster along one of two values. For example, the V/V test environment has results which are clustered around 200 ms and 600 ms, while the P/V environment has results clustered near 250 ms and 1 s. This suggests that some factor like the speed of detecting DN termination and the operating system closing all the open connections on this DN could be responsible for the observed clustering results. There are also several

outliers, including several measurements showing more than 250 packets lost, that could have resulted from the DN operating system occasionally taking a long time to close the open DN sockets.

There is really only one factor that is determined by the EN that is likely to differ in the different test environments: the speed at which the link is determined to have died. In the tested implementation, the link is detected as dead when the DN operating system sends a TCP Close message to the client upon detecting that the DN daemon has died. The OSPF polling time is DN-side, the EN does not perform packet buffering, and those are really the only other major components that are involved during a failure event. The processing speed of the EN is relevant only with respect to how long it takes the client to detect link failure; it would not affect the number of packets processed because a failure does not introduce additional logic into the data path. Consequently, it appears that the more powerful client simply takes a shorter amount of time to detect that the link has died.

The experiment was also performed with a transmission rate of 10 pps, the results of these tests are shown in Figure 5.3. These results were obtained to provide a sanity check against the 100 pps results; it is expected that the drop counts in the 10 pps tests will be approximately $\frac{1}{10}^{th}$ of the results shown in Figure 5.2. This graph also omits two outliers for clarity; the V/V environment dropped approximately 50 packets once, and the P/V environment dropped approximately 45 packets once.

On first inspection the results in Figure 5.3 validates the results obtained in Figure 5.2 — a sending rate of 10 pps shows packet drop rates between 2 and 10 packets, which also corresponds to an approximate loss of one second of connectivity. Clearly, the results obtained by sending at a rate of 10 pps are likely to be a coarser representation of the packet drop rate, this is to be expected given that there are fewer data points available in each run to measure the performance of the system. One interesting observation which can be made about the 10 pps experimental results is that the V/V test environment does not show the same performance as it does in the 100 pps case — the results are clustered around 200 ms and 1.2 s as compared to 200 ms and 600 ms. Indeed, in the 10 pps case the P/P testbed the results show clusters around 200 ms and 1 s, showing perhaps slightly

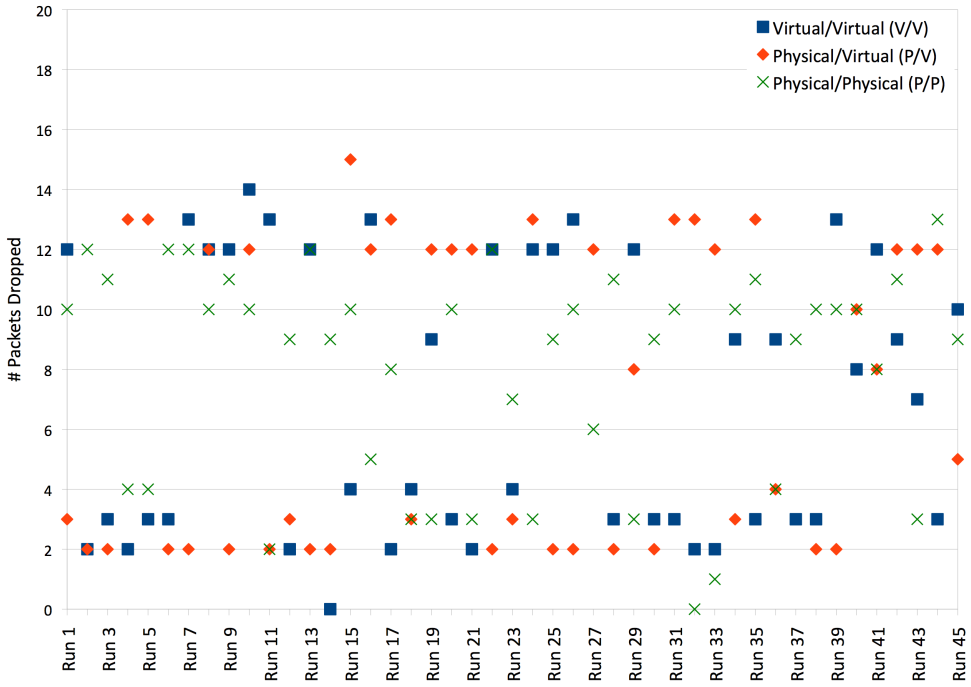

Figure 5.3: Packet Drop Count During A Failover (Rate of 10 Packets Per Second)

faster performance on the slowest testbed. The distribution of the clusters is also not the same between the two test setups — the 10 pps case had results that cluster around the smaller of the two values at approximately twice the rate experienced in the 100 pps case.

The difference between the results obtained for the different packet sending rates is likely the result of when the packets were sent and how the system detected a failure. In the implemented system, the backup DN asserts itself as the new primary after two events have occurred: the EN detects that the primary DN has failed, and the backup DN successfully receives the next packet sent by the EN. There is no out-of-band signaling involved — the backup DN only knows of the failure after the EN has sent its next data packet to the DN. As a result, a slower sending rate is going to mean that the ESR will, on average, receive the updated EN host route at a later point in time. This helps to explain

why the V/V test environment has longer failovers in the 10 pps setup, and why all three environments show similar failover rates when the packet transmission rate is lower.

The difference in clustering ratios between the 10 pps and 100 pps case does not have as clear an explanation; decreasing the rate of packet arrival at the backup DN would not logically increase the likelihood that fewer packets will be dropped by the system. One possible factor is that the lower packet rate reduces the processing overhead each system is handling, allowing the EN to detect the failure of the primary DN more quickly, reducing the number of packets lost. Another possible factor is that a lower packet rate enables the operating system to more quickly detect the termination of the DN software and more quickly close the sockets that were associated with the now-dead DN process. This would allow the EN to be notified more quickly, reducing the number of packets lost.

From a higher-level perspective, all of the environments experienced a very small period over which packets are lost, and the number of packets lost in all cases is also quite low. The visibility of this packet loss will depend upon the characteristics of the application running during a failure; web browsers will likely not experience a visible impact, streaming media with buffering will also likely not be disrupted, while a voice call could have a noticeable but limited interruption of service (a sub-second hiccup) with the observed packet-loss behaviour.

5.2 Impact on Long-Lived TCP Flows

In order to measure the effect of a failure in the middle of a TCP session, a HyperText Transfer Protocol (HTTP) file download was initiated by the EN to an Apache web server one network hop past the ESR. Once the EN received approximately 40% of the file, the primary DN experienced a forced failure, causing the EN to switch to the secondary DN. The failure was triggered at the 40% mark to provide the application TCP flow sufficient time to fully maximize its congestion window, while still allowing time for the stream to ramp up the congestion window again after the failure. The failure results are compared against an identical download that was conducted without the primary DN experiencing a failure. The test is repeated 30 times each for two different sizes of files, one that is 63 MB

and one that is 625 MB. The 625 MB file size was chosen to represent a typical video file, while the 63 MB file size was chosen to represent a reasonably small file yet still take long enough to download so that a failure could be introduced and measured (smaller files would download completely before a failure could be introduced). The amount of time the file takes to download is being used to measure the impact of a failure — if a failure increases the download time it is considered to have had a negative impact on the behaviour of the system.

While running the experiments on the completely physical testbed both the EN and DN were found to be CPU-capped; it was not possible to achieve rates higher than 4.7 MB/s because neither end could process packets faster than that. The testbed consisting of a physical client and a virtual server cluster resulted in only the EN being CPU-capped, limiting the file download rate to around 5.3 MB/s. The completely virtual testbed was able to obtain download rates of nearly 16 MB/s. The VMs were not able to saturate the Gigabit link due to packet and processing overheads. However, the relative speeds of these environments are not the focus of this set of experiments — TCP segment loss and total download times are.

It is hypothesized that a DN failure will not cause a noticeable delay in the download of a file, based upon the packet loss numbers observed in Section 5.1. In the following analysis, the null hypothesis is that a failure will not cause a noticeable increase in file download times; the alternative hypothesis is that a failure does cause a noticeable increase in file download times.

Table 5.1 shows the average download times (in seconds), the standard deviations of the download times (in seconds), and the 95% confidence ranges (in seconds) for the results of both file sizes in all of the test environments. Due to the physical client being CPU-bound it can be observed that the P/V and P/P tests had longer average download times. It is also observed that in all cases the average download time increased when a failure was simulated, and in almost all cases the standard deviation was higher for the failure case — this is expected based upon the packet loss rates discussed in Section 5.1.

To determine whether the increases in average download times and standard deviation under the failure cases are statistically significant, the following approach is used. First,

Table 5.1: Impact of Failure on HTTP File Download Times (UDP channel)

	63 MB File			625 MB File		
	Mean	Std Dev	95% Conf Int	Mean	Std Dev	95% Conf Int
Typical (V/V)	4.326	0.571	0.204	40.284	3.406	1.219
Failure (V/V)	5.784	1.471	0.526	42.993	2.892	1.035
Typical (P/V)	11.823	0.149	0.053	123.296	2.119	0.758
Failure (P/V)	14.060	2.204	0.886	123.605	2.421	0.866
Typical (P/P)	13.656	0.094	0.033	141.180	2.473	0.885
Failure (P/P)	15.342	1.703	0.609	142.603	2.723	0.974

to determine the type of test statistic to use, it is necessary to determine whether the raw test data follows a normal distribution. The one-sample Kolmogorov-Smirnov test was run on each of the 12 data sets of 30 runs each assuming a normal distribution. The results are in Table 5.2, which shows both the Kolmogorov-Smirnov Z-statistic and the 2-tailed asymptotic significance of each case.

In none of the three typical small file size cases it is possible to reject the null hypothesis of normality as the asymptotic significance exceeds 10%. In all cases where a failure was forced with the 63 MB file the normality of the test data is rejected at a 1% significance level. Nonetheless, since there is a sample size of 30 runs for both typical and failure cases it is possible to rely upon the Central Limit Theorem and conduct a standard difference-of-means test. For robustness, however, it is possible to augment the difference-of-means test with a difference-of-medians test for the 63 MB file size simulations. For the 625 MB file size simulations it is not possible to reject the null hypothesis that the test data follows a normal distribution at a 5% significance level in all cases, and at the 10% level in all but the typical virtual/virtual case. Thus, it is possible to rely upon the standard z-test for the difference of means.

Given that the null hypothesis is structured in an equals/not-equals fashion, the two-

Table 5.2: Kolmogorov-Smirnov Test for Normality on the HTTP File Download Times (UDP channel)

	63 MB File		625 MB File	
	K-S Z-Stat	Asympt Sig (2-tail)	K-S Z-Stat	Asympt Sig (2-tail)
Typical (V/V)	1.191	0.117	1.236	0.094
Failure (V/V)	1.746	0.005	1.089	0.187
Typical (P/V)	0.932	0.350	0.910	0.379
Failure (P/V)	1.760	0.004	0.852	0.462
Typical (P/P)	0.965	0.310	0.814	0.522
Failure (P/P)	2.130	0.000	0.562	0.910

tailed test for significance will be used. The test statistic under consideration is the t-statistic:

$$t = \frac{(\bar{x_1} - \bar{x_2}) - (\mu_1 - \mu_2)}{(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2})^{\frac{1}{2}}} \tag{5.1}$$

where $\bar{x}$ is the sample mean, μ is the hypothesized value of the sample mean, and s is the standard deviation. Given that the majority of the measurements have been determined to fit a normal distribution using the Kolmogorov-Smirnov test, the difference-of-means z-test was chosen to test the null hypothesis that the average download time for the failure case equals that of the typical case in order to demonstrate that a failure does not noticeably increase the download time of the file. The variances of the typical and failure cases in each environment have been measured as being different, so each sample variance is used to estimate the variability of the population. The results of the difference-of-means tests for each test situation can be found in Table 5.3.

To verify the above difference-of-means test, a Wilcoxon two-sample test is used to determine whether the typical and failure cases have the same median value. The median

Table 5.3: Difference-of-Mean HTTP File Download Times (UDP channel)

	63 MB File	625 MB File
V/V	1.458 s *	2.710 s *
P/V	2.238 s *	0.309 s
P/P	1.686 s *	1.423 s **

* Reject, 1% Significance

** Reject, 5% Significance

download times, and the differences, are shown in Table 5.4. In all cases except the 625 MB file being download by a physical EN through a virtual DN, both the difference-of-means test and the difference-of-medians test shows that the typical and failure download times differ by a statistically significant amount.

One final test is to see if introducing a failure changes the uncertainty about the download time. In this case, the uncertainty is measured using the standard deviation of the download time across the 30 test runs. It is possible to use the F-distribution to test the null hypothesis that the typical and failure test runs have the same variance. In this case the test statistic is defined as:

$$F = \frac{S_1^2}{S_2^2} \tag{5.2}$$

where S_1 is the standard deviation of Sample 1 and S_2 is the standard deviation of Sample 2. This test statistic needs to exceed 1 so Sample 1 is chosen to be the one with the larger value. This test statistic is distributed with (29,29) degrees of freedom. The null hypothesis of equal variance is rejected for all small file cases, but it is not possible to reject the null for the large file cases. Based upon these results, the null hypothesis has been dis-proven and a failure does cause a statistically significant increase in the download time of the file.

Given that there is a statistically significant difference in file download times, the question becomes whether this increase is significant to users of the system. Table 5.3 shows

Table 5.4: Difference-of-Median HTTP File Download Times (UDP channel)

	63 MB File			625 MB File		
	Median D/L Time			Median D/L Time		
	Typical	Failure	Delta	Typical	Failure	Delta
V/V	4.195 s	5.426 s	1.234 s *	40.025 s	21.723 s	1.698 s *
P/V	11.782 s	13.144 s	1.362 s *	122.556 s	123.21 s	0.661 s
P/P	13.620 s	14.877 s	1.258 s *	141.073 s	142.229 s	1.156 s **

* Reject, 1% Significance

** Reject, 5% Significance

that at most 2.7 seconds was added to the download time; with more than half of the experiments adding under 2 seconds to the download time. It is important to observe that the amount of the delay did not depend on the size of the file; the system introduces a reasonably fixed amount of time to each file download. This is entirely expected — the system downloads part of the file normally, experiences a brief period where the download slows down as it recovers from the failure, and then resumes downloading the file normally. In the 63 MB file download on the V/V testbed the slowdown period constituted a significant portion of the total download time because both of the normal periods elapsed quite quickly, however in the 625 MB file download the P/P testbed the slowdown period was a minor component of the total download time because both normal periods lasted much longer.

The size of the file being downloaded and the available bandwidth of the link then determines whether the slowdown caused by the failure is significant to the user. The testbeds used to perform these experiments are a somewhat unrealistic simulation of Internet behaviour as links rarely have an available rate of 1 Gbps. A smaller link rate will increase the amount of time it takes to download a file of equal size, and given that a failure has been shown to introduce a fixed amount of time to a download the relative impact of

a failure will decrease as the link rate decreases. Given that each application has different requirements it is not realistic to state that this solution is going to provide a suitable failover speed for all environments, but it is possible to say that a failure introduces a fixed, several second delay to an HTTP download.

5.3 Impact of TCP Retransmissions

The increase in download times was initially believed to be the result of packets lost in the failover process; measuring the TCP retransmission statistics on the web server showed that a variable number of segments were retransmitted during each run. The exact number of retransmissions were plotted against the total download time in Figure 5.4 and in Figure 5.5 for the P/P testbed. The scatter plots of retransmissions versus download times shows that there is no obvious correlation between the number of retransmitted packets and the amount of time the file took to download. Also, the retransmission behaviour of the system varied depending on the size of the file. In the 63 MB file case, there appears to be a few very distinct download times (15 seconds, 16 seconds, 20 seconds) that are measured with a wide range in the number of retransmitted packets — the system was able to download the file in 15 seconds while retransmitting between 3 and 73 segments. In the 625 MB file case, the amount of time the file took to download was not quite as strictly aligned, but it still exhibited a single download time (around 141 seconds) while retransmitting between 3 and 100 segments.

It is expected that the exact number of packets that need to be retransmitted will vary based upon a number of factors, particularly where the TCP stream was with respect to the outstanding acknowledgments when the failure occurred as well as a variety of other tuning parameters that are dependent on the current state of the TCP stream. However, it was initially expected that this would correspond to the amount of time the file ended up taking to download — because retransmissions slow down the TCP stream and increase download times in standard packet-loss scenarios. The measured results suggest that the increase in download times is not only being caused by TCP retransmissions; there is another aspect of the system that is influencing the behaviour of the TCP stream during

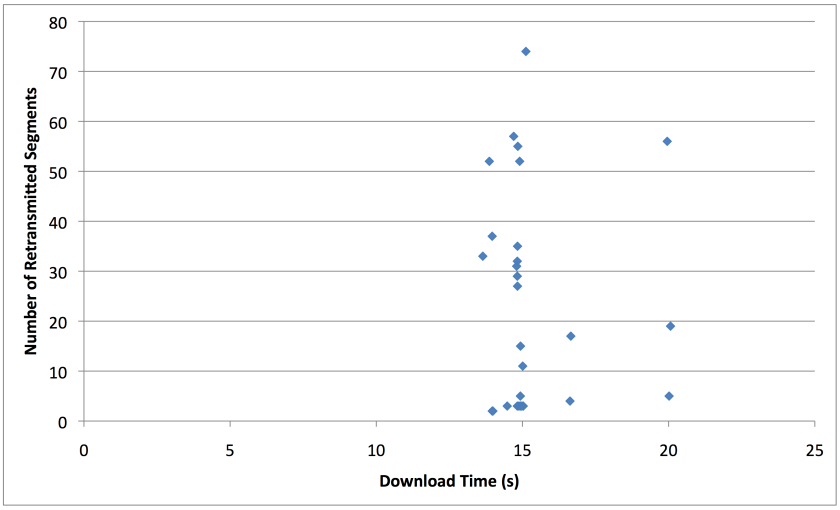

Figure 5.4: Impact of Retransmissions on Download Time (63 MB File, P/P)

the failover. If TCP retransmissions were solely responsible for the slowdown then Figure 5.4 and Figure 5.5 would show that test runs with more retransmissions would take longer; however, the results do not show this. Two other factors that contribute to the increase in download time have been previously discussed — the variable amount of time taken by the OSPF routing daemon to update the routing tables in the cluster, and the variable amount of time it takes the EN to detect that the primary DN has failed.

It is likely that the TCP retransmissions do have some impact on the performance of the system; as discussed in Section 5.1 the system experience between 200 ms and one second worth of packet loss when only ICMP packets were sent, yet Table 5.3 shows that a HTTP session experiences up to 2.7 seconds worth of delay during a failure. There are clearly some differences in the two scenarios — a significant difference is that the ICMP scenario did not send packets at the maximum rate supported by the link. As was observed in Section 5.1 the system had a higher probability of taking one second to recover from a failure when packets were sent at 100 pps as compared to 10 pps; this suggests that higher packet rates negatively impact the speed of recovery from a failure. This likely

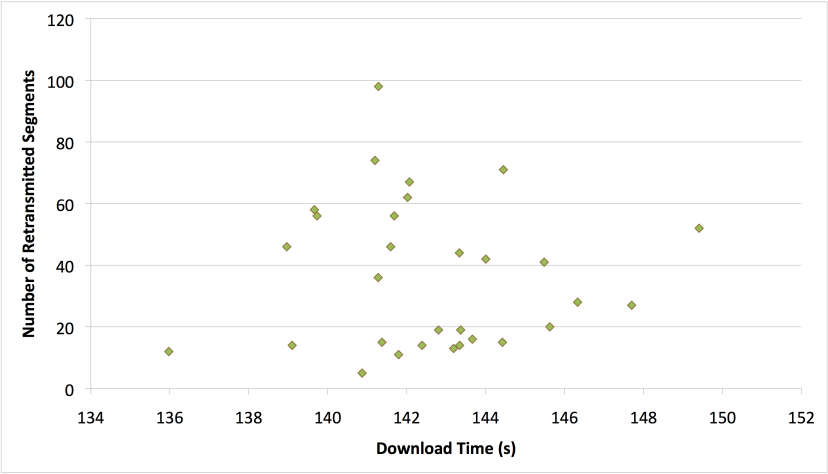

Figure 5.5: Impact of Retransmissions on Download Time (625 MB File, P/P)

contributes to TCP sessions experiencing a longer interruption than a stream of ICMP packets; it is also likely that TCP retransmissions contribute to TCP sessions exhibiting a slower recovery from DN failure than an ICMP stream.

The retransmission behaviour of TCP in the physical/virtual environment is similar to both the purely virtual and the purely physical environment as shown by Figure 5.6 and Figure 5.7.

As the scatter plot of segment retransmission count to download time shows in both Figure 5.8 and Figure 5.9, the fully virtual environment experiences the same results whereby the number of retransmitted segments does not correlate to the download time of the file.

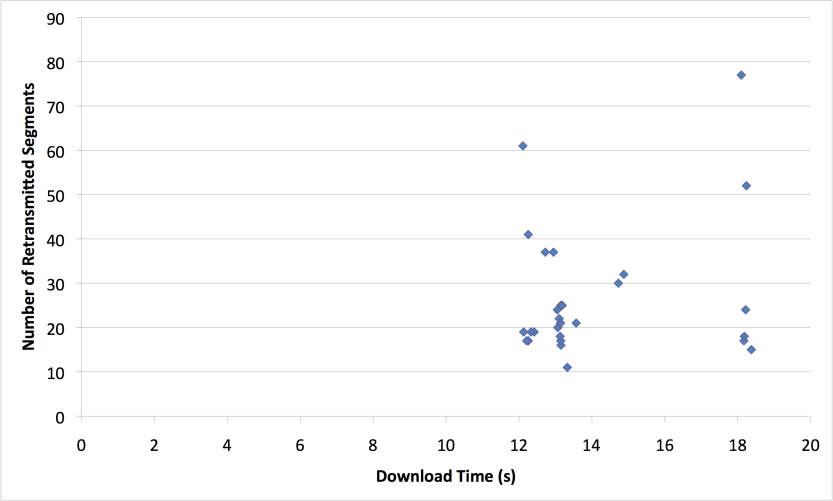

Figure 5.6: Impact of Retransmissions on Download Time (63 MB File, P/V)

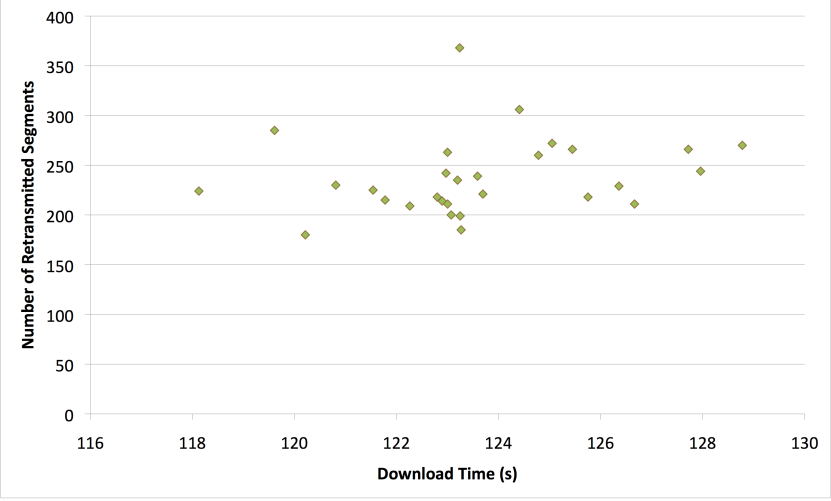

Figure 5.7: Impact of Retransmissions on Download Time (625 MB File, P/V)

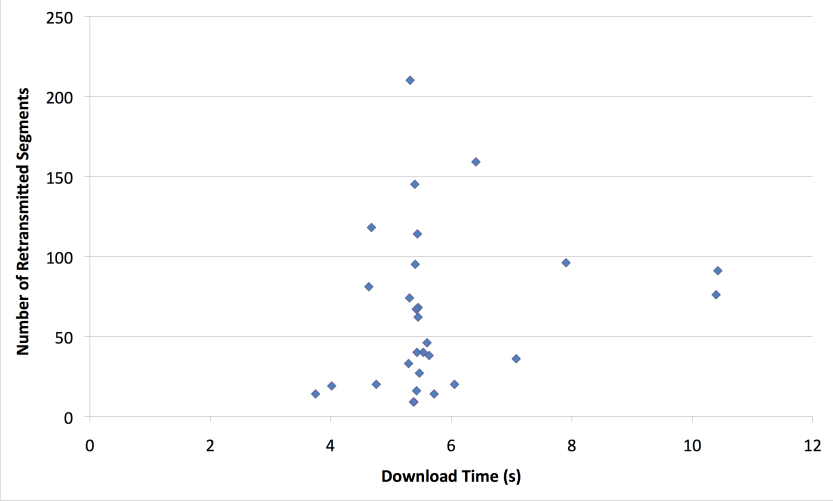

Figure 5.8: Impact of Retransmissions on Download Time (63 MB File, V/V)

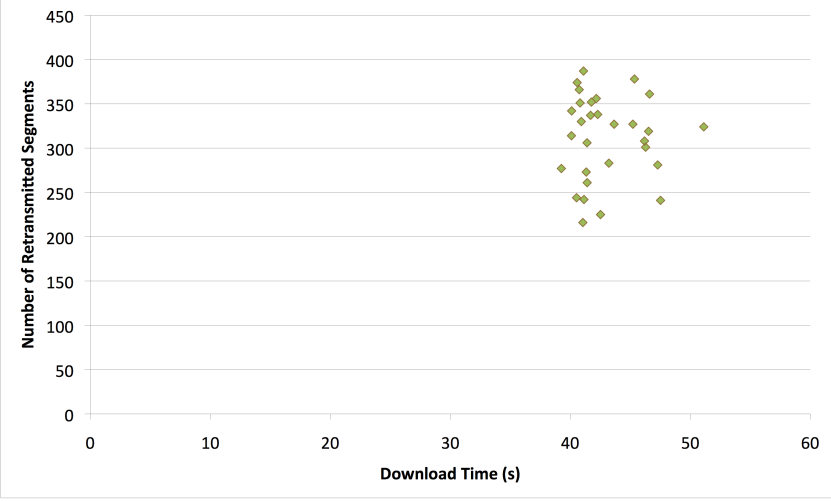

Figure 5.9: Impact of Retransmissions on Download Time (625 MB File, V/V)

5.4 Impact of Reliable Encapsulated Packet Delivery

The results shown in Table 5.1 show the behaviour of the system during a failover from a UDP data channel to another UDP data channel. While a UDP data channel is preferable over a TCP data channel for a number of previously elaborated reasons, it is sometimes necessary to utilize TCP for the data channel. The behaviour of a TCP channel is sufficiently different than that of a UDP channel to believe that the previously obtained results are not an accurate reflection of the failover times that a system with a TCP data channel would experience. This comes from the well-understood impact of running TCP over TCP [28] — namely that in the case of packet loss the back-off algorithm implemented by TCP causes the connection to degrade quickly. To determine the impact that using a reliable data channel has on failover times, the experiments run in Section 5.2 were repeated with *Accoriem* configured to utilize a TCP data channel. Instead of having 30 experimental runs for each case, only 15 were performed for each case.

The results of running the HTTP file download test on the P/P and V/V testbeds are shown in Table 5.5, which contains both the mean download times and the standard deviation of each set of download times. The run times without failure are similar to those found previously in Table 5.1. The use of a TCP tunnel introduces additional overhead into the system, however. The congestion-control algorithm limits the number of outstanding packets based upon the current packet-loss state; since UDP does not implement this there is a period of time as the TCP tunnel is ramping up that it transmits slower than UDP. This increases the time it takes to download an equivalently sized file. Also, the extra 12 bytes of header in every TCP packet increases the required number of bytes that are transmitted to completely download a file, also increasing the time a file download will take. This is demonstrated in Table 5.5, where the 63 MB file took an additional 0.2 seconds to download, and the 625 MB file took an additional 3.5 seconds to download. The header overhead negatively impacts larger files more than smaller files because per-packet overhead is cumulative, while the additional congestion-control overhead is the same regardless of file size because congestion-control overhead is determined by packet loss, not file size.

During a failure the impact of using a TCP tunnel is more visible; the 63 MB file takes

Table 5.5: Impact of Failure on HTTP File Download Times (TCP channel)

	63 MB File		625 MB File	
	Mean	Std Dev	Mean	Std Dev
Typical (V/V)	4.150 s	0.285 s	39.777 s	2.522 s
Failure (V/V)	8.382 s	1.990 s	48.519 s	2.702 s
Typical (P/P)	13.898 s	0.183 s	145.534 s	2.621 s
Failure (P/P)	16.924 s	0.937 s	150.279 s	6.949 s

approximately 4 seconds to recover from a failure, while the 625 MB file takes nearly 10 seconds to recover. This is a drastic change from the UDP tunnel results, where both the 63 MB file and the 625 MB file managed to recover in less than 3 seconds. The impact of packet loss rates with a UDP tunnel were explored in Section 5.2, and it was determined that the three main causes of delay in failure recovery were the OSPF daemon routing-table scan time, the amount of time it took the EN to detect a failure, and the HTTP TCP session retransmitting the lost packets. The results in Table 5.5 show that running a TCP-based HTTP download over a TCP-based data channel results in even worse performance than using a UDP tunnel — the question is why. The OSPF daemon routing-table scan time did not change, and the EN should be able to detect a failure just as quickly because the failure detection mechanism is independent of the tunnel transport protocol. Therefore, the increase in recovery time is likely due to the interaction of packet retransmission between the HTTP TCP session and the *Accoriem* TCP session. The HTTP download is forced to wait for more packets as *Accoriem* retransmits the packets that were lost in its TCP session. The loss of packets causes *Accoriem* to back off, which delays the acknowledgments to the HTTP session which forces it to back off. This effect causes the HTTP download to slow down much more than if it was handling packet loss recovery (as it does when using a UDP tunnel. This is, fundamentally, why TCP-in-TCP is considered to be such a bad idea — small slowdowns at one point in the link have a cascading effect on all of the internal traffic.

5.5 Experimental Observations

While evaluating the implementation it was noticed that, immediately following a failover, the IP time-to-live (TTL) values on packets being sent to the client through the encapsulation system were one hop smaller than expected for a short (~50 ms) period of time, before returning to the expected value. It was discovered that this was caused by the particular OSPF implementations used (*bird* on the DNs and *OpenOSPFd* on the ESR). While the OSPF update is received at both the primary DN and the ESR immediately following a failover (verified using packet traces), it appears that *bird* applies these updates immediately while *OpenOSPFd* applies these updates approximately 50 ms later. Consequently, packets being sent from the ESR were routed to the primary DN, which then routed them to the secondary DN, which then sent them to the client properly. While this does not impact our experimental results (because the primary DN was still accessible), this would clearly introduce another delay (of around 50 ms) into the system should the primary DN be unable to route packets (either due to a hardware failure or a routing table error).

While experimenting in the virtual environment, it was observed that the choice of virtual network driver impacts the speed of failover; the system experienced slightly slower failover when using the VMXNET3 driver as compared to the E1000 driver. The resulting performance difference could be a result of how the VMXNET3 driver handles the initial-connection process or it could be due to the VMXNET3 driver being manually compiled on the virtual machines (E1000 support is built into the Linux kernel, VMXNET3 is not). Either way, the E100 driver was selected for all of the experiments.

Chapter 6

Conclusion and Future Work

6.1 Conclusions

This thesis has presented a new architecture for developing fault-tolerant encapsulation systems that support IP continuity without requiring complex server-side state synchronization. Not only does this solution provide transparent failover between DNs, it does so in a manner that is relatively straightforward to configure and expand as additional DNs are required. The proposal was validated via an implementation that is integrated into an existing encapsulation solution, and it was demonstrated that the costs of adopting this approach are minimal. Experimentally, it was shown that this solution functions for both unreliable and reliable protocols (ICMP and HTTP, respectively), introducing only minor packet loss during a failure event while providing IP continuity across endpoints. This encapsulation reliability architecture has been adopted by a commercial encapsulation solution (Pravala's *Accoriem*) to allow for transparent failure of decapsulators.

6.2 Future Work

The solution previously presented has focused entirely on handling DN failures in a cluster that is co-located — the DNs and ESR are all located on the same subnet. This is required

to ensure that the encapsulated IP address exposed by each client is routed properly, with all traffic going through the ESR and the proper DN. While adding a high level of reliability to a cluster located in a single datacentre, it does nothing to allow the client to seamlessly fail over in the case of a datacentre failure. Dealing with datacentre failover is a much more complex problem because it becomes necessary to redirect traffic destined to a specific IP prefix to a new datacentre. The process of designing and developing the solution in this thesis included some work in solving this problem, but there are a number of issues that need to be handled before any solution is ready for deployment. This includes determining an efficient way to allow the same IP address subnet to be announced to the Internet from multiple, distinct physical locations, and quickly updating distributed routing tables in a manner to ensure that complete loss of any one location does not prevent the other locations from being able to quickly take over responsibility for the shared address space.

The encapsulation endpoint reliability architecture presented to this point has been entirely focused on providing IP continuity in the event of a DN failure; however there are many additional situations in which the ability to transparently migrate a client from one DN to another would prove to be useful. Another situation in which the above architecture can be utilized is DN load balancing, where clients from overloaded DNs can be migrated to less heavily loaded DNs without interrupting client traffic. Integrating dynamic load balancing into the design is conceptually a relatively straightforward process — when it is determined that a server has become too heavily loaded, it can simply close a set (all hosts in a subnet) of client connections and trust that they will fail over to their backup DN. Determining which set of hosts to kill is likely to be one of the most difficult parts of integrating this functionality into the system.

While implementing the architecture it became apparent that it may be possible to implement a similar solution without any changes on the DN; depending on the design of the DN a separate module running on the same machine could enable similar functionality. Such a module would enable existing encapsulation systems to support transparent failover with only client-side changes, simplifying deployment and speeding adoption. This module would work by sniffing incoming traffic on the DN client-connected interfaces and recording which IP address is assigned to which connection. The module would then sniff every packet

prior to its reception by the DN, look for any addresses not included in the DNs address pool, and perform IP replacement on the encapsulated IP header prior to its processing by the DN. This will enable the host to continue using its old IP address, while the DN receives the packet as if it came from a new one (a crude form of NAT, in effect). This approach requires application-level intelligence though, making it difficult to develop a generic module for all encapsulation systems. As well, this approach requires that the payload of the packets be unencrypted (though encryption could be handled by extending this approach into a full-blown proxy running in tandem with the DN). This solution is not investigated further here because the authors had full control over the implementation of both the EN and the DN; however, it could be worth investigating further if others encounter environments where the DN cannot be modified.

APPENDICES

Appendix A

List of Acronyms

CARP Common Address Redundancy Protocol

CPU Central Processing Unit

DFZ Default Free Zone

DN Decapsulation Node

DN-FD Decapsulation Node-Failure Detector

EN Encapsulation Node

ESP Encapsulating Security Protocol

ESR Edge Service Router

FT-TCP Fault-Tolerant Transmission Control Protocol

GRE Generic Routing Encapsulation

HSRP Hot Standby Router Protocol

HTTP HyperText Transfer Protocol

ICMP Internet Control Message Protocol

IP Internet Protocol

IPsec Internet Protocol Security

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

ISP Internet Service Provider

L2 Layer 2

L3 Layer 3

MAC Media Access Control

NAT Network Address Translation

OSPF Open Shortest Path First

P/P Physical/Physical

P/V Physical/Virtual

PA Provider Assigned

PI Provider Independent

RIP Routing Information Protocol

SCTP Stream Control Transfer Protocol

TCP Transmission Control Protocol

UDP User Datagram Protocol

V/V Virtual/Virtual

VM Virtual Machine

VPN Virtual Private Network

VRRP Virtual Routing Redundacy Protocol

References

[1] L. Alvisi, T. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagoodnov, "Wrapping server-side TCP to mask connection failures," in *Proceedings of the IEEE Conference on Computer Communications*, 2001. 7

[2] K. Argyraki, S. Baset, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, E. Kohler, M. Manesh, S. Nedevschi, and S. Ratnasamy, "Can software routers scale?" in *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*. New York, NY, USA: ACM, 2008, pp. 21–26. 14, 15

[3] H. Ballani, P. Francis, T. Cao, and J. Wang, "Making routers last longer with ViAggre," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2009, pp. 453–466. 15

[4] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 335–350. 20

[5] G. Candea and A. Fox, "Crash-only software," in *Proceedings of the 9th HotOS Workshop on Hot Topics in Operating Systems*. Berkeley, CA, USA: USENIX Association, 2003. 16

[6] J. Dean and S. Ghemawat, "MapReduce: similified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 335–350. 16

[7] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating System Principles*. New York, NY, USA: ACM, 2009, pp. 15–28. 15

[8] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy, "Towards high performance virtual routers on commodity hardware," in *Proceedings of the 2008 ACM CoNEXT Conference*. New York, NY, USA: ACM, 2008, pp. 20:1–20:12. 14

[9] O. Filip, L. Forst, P. Machek, M. Mares, and O. Zajicek, "BIRD internet routing daemon." [Online]. Available: http://bird.network.cz 40

[10] J. Hamilton, "On designing and deploying Internet-scale services," in *Proceedings of the 21st conference on Large Installation System Administration Conference*. Berkeley, CA, USA: USENIX Association, 2007, pp. 18:1–18:12. 16

[11] C. Huitema, "Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs)," RFC 4380 (Proposed Standard), Internet Engineering Task Force, Feb. 2006, updated by RFC 5991. [Online]. Available: http://www.ietf.org/rfc/rfc4380.txt 1

[12] A. Huttunen, B. Swander, V. Volpe, L. DiBurro, and M. Stenberg, "UDP Encapsulation of IPsec ESP Packets," RFC 3948 (Proposed Standard), Internet Engineering Task Force, Jan. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc3948.txt 7

[13] M. Isard, "Autopilot: automatic data center management," *SIGOPS Operating System Review*, vol. 41, pp. 60–67, 2007. 16

[14] S. Kent and K. Seo, "Security Architecture for the Internet Protocol," RFC 4301 (Proposed Standard), Internet Engineering Task Force, Dec. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4301.txt 1

[15] C. Kim, M. Caesar, and J. Rexford, "Floodless in SEATTLE: a scalable Ethernet architecture for large enterprises," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. New York, NY, USA: ACM, 2008, pp. 3–14. 12

[16] S. Nadas, "Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6," RFC 5798 (Proposed Standard), Internet Engineering Task Force, Mar. 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5798.txt 27

[17] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: a scalable fault-tolerant Layer 2 data center network fabric," in *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*. New York, NY, USA: ACM, 2009, pp. 39–50. 12

[18] C. Perkins, "IP Mobility Support for IPv4," RFC 3344 (Proposed Standard), Internet Engineering Task Force, Aug. 2002, updated by RFC 4721. [Online]. Available: http://www.ietf.org/rfc/rfc3344.txt 1

[19] R. Robinson, "Failover test results." [Online]. Available: http://ccng.uwaterloo.ca/~r3robins/masThesis/results 41

[20] J. Schmidke, "Practical multi-interface network access for mobile devices," 2010. [Online]. Available: http://ccng.uwaterloo.ca/~skjakubc/pravala/AccoriemTechDoc.pdf 32

[21] W. Simpson, "IP in IP Tunneling," RFC 1853 (Informational), Internet Engineering Task Force, Oct. 1995. [Online]. Available: http://www.ietf.org/rfc/rfc1853.txt 1

[22] C. Systems, "Cisco Secure Remote Access Cisco ASA 550 Series SSL/IPsec VPN Edition." [Online]. Available: http://www.cisco.com/en/US/prod/collateral/vpndevc/ps6032/ps6094/ps6120/prod_brochure0900aecd80402e39.html 13

[23] ——, "Hot Standby Router Protocol features and functionality." [Online]. Available: http://www.cisco.com/en/US/tech/tk648/tk362/technologies_tech_note09186a0080094a91.shtml 27

[24] ——, "Using HSRP for fault-tolerant IP routing." [Online]. Available: http://www.cisco.com/en/US/docs/internetworking/case/studies/cs009.html 8

[25] O. D. Team, "Common Address Redundancy Protocol." [Online]. Available: http://www.openbsd.org/cgi-bin/man.cgi?query=carp 27

[26] T. O. D. Team, "Open Shortest Path First daemon." [Online]. Available: http://www.openbsd.org/cgi-bin/man.cgi?query=ospfd 40

[27] T. Q. D. Team, "Quagga Routing Suite." [Online]. Available: http://www.quagga.net 40

[28] O. Titz, "Why TCP over TCP is a bad idea." [Online]. Available: http://sites.inka.de/bigred/devel/tcp-tcp.html 58

[29] C. Vogt, "Six/one router: A scalable and backwards compatible solution for Provider-Independent addressing," in *Proceedings of the 3rd International Workshop on Mobility in the Evolving Internet Architecture*. New York, NY, USA: ACM, 2008, pp. 13–18. 10, 11

[30] L. Wang, D. Jen, M. Meisel, B. Zhang, H. Yan, D. Massey, and L. Zhang, "Towards a new Internet routing architecture: arguments for separating edges from transit core," in *Seventh Workshop on Hot Topics in Networks*, 2008. 10, 11, 15

[31] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. Bressoud, "Engineering fault-tolerant TCP/IP server using FT-TCP," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2003. 7