Towards a Common System Architecture for Dynamically Deploying Network Services in Routers and End Hosts

Jae Woo Lee

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2012

ABSTRACT

Towards a Common System Architecture for Dynamically Deploying Network Services in Routers and End Hosts

Jae Woo Lee

The architectural simplicity of the core Internet is a double-edged sword. On the one hand, its agnostic nature paved the way for endless innovations of end-to-end applications. On the other hand, the inherent limitation of this simplicity makes it difficult to add new functions to the network core itself. This is exacerbated by the conservative tendency of commercial entities to "leave well-enough alone", leading to the current situation often referred to as the *ossification* of the Internet. For decades, there has been practically no new functionality that has been added to the core Internet on a large scale.

This thesis explores the possibility of enabling in-network services towards the goal of overcoming the ossification of the Internet. Our ultimate goal is to provide a common run-time environment supported by all Internet nodes and a wide-area deployment mechanism, so that network services can be freely installed, removed, and migrated among Internet nodes of all kinds–from a backbone router to a set-top box at home. In that vision of a future Internet, there is little difference between servers and routers for the purpose of running network services. Services can run anywhere on the Internet. Application service providers will have the freedom to choose the best place to run their code.

This thesis presents NetServ, our first step to realize the vision of network services running anywhere on the Internet. NetServ is a node architecture for dynamically deploying in-network services on edge routers. Network functions and applications are implemented as software modules which can be deployed at any NetServ-enabled node on the Internet, subject to policy restrictions. The NetServ framework provides a common execution environment for service modules and the ability to dynamically install and remove the services

without restarting the nodes. There are many challenges in designing such a system. The main contribution of this thesis lies in meeting those challenges.

First, we recognize that the primary impetus for adopting new technologies is economics. To address the challenge of providing economic incentives for enabling in-network services, we demonstrate how NetServ can facilitate an economic alliance between content providers and ISPs. Using NetServ, content providers and the ISPs operating at the network edge (aka *eyeball* ISPs) can enter into a mutually beneficial economic relationship. ISPs make their NetServ-enabled edge routers available for hosting content providers' applications and contents. Content providers can operate closer to end users by deploying code modules on NetServ-enabled edge routers. We make our case by presenting NetServ applications which represent four concrete use cases.

Second, our node architecture must support both traditional server applications and in-network packet processing applications since content providers' applications running on ISPs' routers will combine the traits of both. To address this challenge, NetServ framework can host a packet processing module that sits in the data path, a server module that uses the TCP/IP stack in the traditional way, or a combined module that does both. NetServ provides a unified runtime environment between routers and servers, taking us a step closer to the vision of the unified runtime available on all Internet nodes.

Third, we must provide a fast and streamlined deployment mechanism. Content providers should be able to deploy their applications at any NetServ-enabled edge router on the Internet, given that they have proper authorizations. Moreover, in some application scenarios, content providers may not know the exact locations of the target routers. Content providers need a way to send a message to install or remove an application module *towards* a network destination, and have the NetServ-enabled routers located in the path catch and act on the message. To address this challenge, we adopted on-path signaling as the deployment mechanism for NetServ. A NetServ signaling message is sent in an IP packet towards a destination. The packet gets forwarded by IP routers as usual, but when it transits a NetServ-enabled router, the message gets intercepted and passed to the NetServ control layer.

Fourth, a NetServ-enabled router must support the concurrent executions of multiple content providers' applications. Each content provider's execution environment must be isolated from one another, and the resource usage of each must be controlled. To address the challenge of providing a robust multi-user execution environment, we chose to run NetServ modules in user space. This is in stark contrast to most programmable routers, which run service modules in kernel space for fast packet processing. Furthermore, NetServ modules are written in Java and run in Java Virtual Machines (JVMs). Our choice of user space execution and JVM allows us to leverage the decades of technology advances in operating systems, virtualization, and Java.

Lastly, in order to host the services of a large number of content providers, NetServ must be able to scale beyond the single-box architecture. We address this challenge with the multi-box lateral expansion of NetServ using the OpenFlow forwarding engine. In this extended architecture, multiple NetServ nodes are attached to an OpenFlow switch, which provides a physically separate forwarding plane. The scalability of user services is no longer limited to a single NetServ box.

Additionally, this thesis presents our prior work on improving service discovery in local and global networks. The service discovery work makes indirect contribution because the limitations of local and overlay networks encountered during those studies eventually led us to investigate in-network services, which resulted in NetServ. Specifically, we investigate the issues involved in bootstrapping large-scale structured overlay networks, present a tool to merge service announcements from multiple local networks, and propose an enhancement to structured overlay networks using link-local multicast.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to thank my advisor, Prof. Henning Schulzrinne, whose influence on my life goes far beyond my research. I walk away with many lessons that will carry me through all my future endeavors. He taught me that things are not always what they seem at first, that I shouldn't believe everything I read, and that I should always think critically and for myself. He taught me to have a vision, to plan for long term, to see the big picture, but at the same time, to be practical and pay attention to detail. For the past six years, he has shown me that a highly accomplished person can be completely down-to-earth, infinitely patient, and simply nice. Hopefully some of those traits have rubbed off on me during the years.

I would like to thank my thesis committee: Prof. Gil Zussman, Prof. Roxana Geambasu, Dr. Volker Hilt, and Prof. Yechiam Yemini. Their criticisms and encouragements have been invaluable for reexamining and improving my work. I hope they allow me to keep seeking their advice as I continue my research in the future.

I would also like to thank Zoran Despotovic and Wolfgang Kellerer, who sponsored part of my research and provided much needed guidance during the early years of my research.

My heartfelt thanks to the NetServ team members: Roberto Francescangeli, Wonsang Song, Emanuele Maccherani, Jan Janak, Suman Srinivasan, Michael Kester, Eric Liu, and Salman Baset. The design and implementation of NetServ is a result of a group effort, thus I share the sense of ownership with the team.

Many thanks to Mark Berman and Niky Riga of the GENI Project Office for their continued support for NetServ.

Thanks to all my colleagues at the IRT lab. A genuine camaraderie on the team made this place such a great place to work. Thanks to other graduate students at Columbia University for their friendship.

Thanks also to Mauro Femminella and Gianluca Reali for feedback and discussion, and to Raynald Seydoux, Abhishek Srivastava and Nathan Miller for their assistance with implementation. I also acknowledge the funding organizations that sponsored my research.

Teaching has been a big part of my graduate school years. I thank all my former students and teaching assistants in the Advanced Programming classes for their enthusiasm.

My sincere thanks to John Kender, Mike Kester, Rosemary Addarich, Adam Cannon, and Shree Nayar for their support and encouragement for my teaching.

I must give thanks to my parents, Dong Hee Lee and Hae Rim Lee, who have endured the long years of their son not acting his age. There would be no thesis without their support and patience. And thanks, and a big hug, to my boys, Adrian and Tristan. None of these mean much without them.

Last but not least, Mia, my love, sorry it took a while. I'm done now.

To my family.

Chapter 1

Introduction

The architectural simplicity of the core Internet is a double-edged sword. On the one hand, its agnostic nature paved the way for endless innovations of end-to-end applications. On the other hand, the inherent limitation of this simplicity makes it difficult to add new functions to the network core itself. This is exacerbated by the conservative tendency of commercial entities to "leave well-enough alone", leading to the current situation often referred to as the *ossification* of the Internet. For decades, there has been practically no new functionality that has been added to the core Internet on a large scale. When there were needs, researchers and entrepreneurs had to find clever roundabout ways to implement the desired functionalities in the application layer rather than in the network layer even though the latter would have led to superior solutions.

This is changing now. Many researchers believe that it is no longer possible to *patch up* the Internet in the face of exploding bandwidth usage and ever-increasing demand for new in-network functionalities. Bold proposals that advocate clean-slate redesigns of the Internet architecture are no longer perceived as merely satisfying academic curiosity. In fact, the National Science Foundation of the United States has launched GENI [12], a large-scale project involving academic and industrial teams across the country, which aims to provide an Internet-scale testbed for such disruptive proposals that cannot be tested on the current Internet.

This thesis explores the possibility of enabling in-network services towards the goal of overcoming the ossification of the Internet. What types of in-network services are helpful for today's and tomorrow's Internet applications? What characteristics do those in-network services have in common? Can the current Internet architecture support such in-network services? If not, what new functions are needed? How do we get there?

We start by assuming that two fundamental characteristics of today's Internet will survive in the future: the packet transport (think IP today) and the concept of addressable services (think web servers today.) We jettison, however, the dichotomy that the two characteristics impose on the Internet actors today. On the one hand, ISPs operate routers which process packets in the network core–forwarding, monitoring and manipulating them–but routers do not normally provide addressable services. On the other hand, service providers and end users operate host computers which provide addressable services, but the providers and users do not usually have a means to process in-coming packets in the network core. What if service providers can deploy network services on a backbone router? What if end users can deploy a packet monitoring tool on their ISPs' edge routers?

Our ultimate vision calls for a common runtime environment supported by all Internet nodes and a wide-area deployment mechanism, so that network services can be freely installed, removed, and migrated among Internet nodes of all kinds–from a backbone router to a set-top box at home. In that vision of a future Internet, there is little difference between servers and routers for the purpose of running network services. Services can run anywhere on the Internet. Application service providers will have the freedom to choose the best place to run their code.

1.1 Main Contribution: Meeting the Challenges for In-Network Services Framework

This thesis presents NetServ, our first step to realize the vision of network services running anywhere on the Internet. NetServ is a node architecture for dynamically deploying in-network services on edge routers. Network functions and applications are implemented as software modules which can be deployed at any NetServ-enabled node on the Internet, subject to policy restrictions. The NetServ framework provides a common execution environment for service modules and the ability to dynamically install and remove the services without restarting the nodes.

There are many challenges in designing such a system. The main contribution of this thesis lies in meeting those challenges. In this section, we identify the challenges and describe how NetServ addresses the challenges.

Economic incentive

We recognize that the primary impetus for adopting new technologies is economics. Our argument for enabling in-network services will ring hollow if we fail to present compelling use cases that demonstrate clear economic benefits for all the stakeholders involved in developing, deploying and hosting the services.

To address the challenge of providing economic incentives for enabling in-network services, we demonstrate how NetServ can facilitate an economic alliance between content providers and ISPs. The two Internet stakeholders are currently in a tussle. As the traditional lines between different Internet actors begin to disappear, they frequently encroach what used to be each other's territory.

Using NetServ, content providers and the ISPs operating at the network edge (aka *eyeball* ISPs) can enter into a mutually beneficial economic relationship. ISPs make their NetServ-enabled edge routers available for hosting content providers' applications and contents. Content providers can operate closer to end users by deploying code modules on NetServ-enabled edge routers. We discuss this model in detail in Chapter 5. We make our case by presenting four concrete use cases. Chapter 7 presents the use cases through NetServ applications that we have built.

In fact, our focus on the economic story limits the scope of this thesis. We do not present NetServ as the solution to fulfill the ultimate vision of enabling services everywhere. NetServ is the first step towards the vision, which focuses on activating edge routers, where we found compelling economic story. Applying NetServ to other Internet nodes remains as future work and thus outside the scope of this thesis.

Unified runtime environment

Content providers' applications running on ISPs' routers will combine the traits of both traditional end-to-end network services and in-network router functions. Our node architecture must support both traditional server applications and in-network packet processing applications.

To address the challenge of accommodating both server applications and router functions in a single node, NetServ framework can host a *packet processing application module* that sits in the data path, a *server application module* that uses the TCP/IP stack in the traditional way, or a combined application module that does both. The showcase applications in Chapter 7 use both features.

NetServ provides a unified runtime environment between routers and servers, taking us a step closer to the vision of the unified runtime available on all Internet nodes.

Wide-area deployment

For the cooperation between content providers and ISPs to be effective, providing a fast and streamlined deployment mechanism is crucial. Content providers should be able to deploy their applications at any NetServ-enabled edge router on the Internet, given that they have proper authorizations.

Consider a scenario where a content publisher wants to deploy a web caching application module *near* a certain group of end users, due to a sudden surge in traffic from that region, for instance. In this case, the content provider needs to deploy its application module at a NetServ-enabled edge router in that general vicinity, but it does not need to know the precise location of the router. In fact, the location information may not be available to the content provider since ISPs may not wish to disclose their network topologies to their customers.

Content providers need a way to send a message to install or remove an application module *towards* a network destination, and have the NetServ-enabled routers located in the path catch and act on the message. To address this challenge, we adopted on-path signaling as the deployment mechanism for NetServ. NetServ signaling is based on the Next Steps in Signaling (NSIS) protocol suite [76], an IETF standard for signaling. A

NetServ signaling message is sent in an IP packet towards a destination. The packet gets forwarded by IP routers as usual, but when it transits a NetServ-enabled router, the message gets intercepted and passed to the NetServ control layer. We describe this mechanism in detail in Section 6.1.1.

Multi-user execution environment

A NetServ-enabled router must support the concurrent executions of multiple content providers' applications. Each content provider's execution environment must be isolated from one another, and the resource usage of each must be controlled.

To address the challenge of providing a robust multi-user execution environment, we chose to run NetServ modules in user space. This is in stark contrast to most programmable routers, which run service modules in kernel space for fast packet processing. Furthermore, NetServ modules are written in Java and run in Java Virtual Machines (JVMs). Our choice of user space execution and JVM allows us to leverage the decades of technology advances in operating systems, virtualization, and Java. We discuss these mechanisms in detail in Section 6.2.

Performance and Scalability

Running service modules in user space JVM raises the question on whether NetServ can provide adequate performance. Moreover, in order to host the services of a large number of content providers, NetServ must be able to scale beyond the single-box architecture.

Our evaluation of a Java service module performing deep packet inspection and modification indicates that the overhead is indeed significant, but not prohibitively so. The throughput achieved on a modestly equipped Linux server matches the average traffic seen by a typical edge router, indicating that the solution is quite usable in a low traffic environment.

The real answer to the performance and scalability challenge is the multi-box lateral expansion of NetServ using the OpenFlow [99] forwarding engine. In this extended architecture, multiple NetServ nodes are attached to an OpenFlow switch, which provides a physically separate forwarding plane. The scalability of user services is no longer limited to a single NetServ box. We describe the OpenFlow extension in Chapter 8.

1.2 Overview of the Thesis

This thesis is divided into two parts. Part I consists of Chapters 2, 3 and 4. This series of work, prior to our NetServ work, focuses on improving service discovery in local and global networks. They make indirect contribution because the limitations of local and overlay networks encountered during those studies eventually led us to investigate in-network services, which resulted in NetServ, our main contribution. Thus, Part I can be regarded as a prelude to Part II which describes NetServ. Part II consists of Chapters 5 to 9. A brief overview of each chapter is as follows.

Chapter 2 investigates the issues involved in bootstrapping large-scale DHT networks [95]. It has been claimed that the standard DHT join protocols are not adequate for bootstrapping large DHT networks from scratch. We debunk the claim by showing that Chord and Kademlia DHT take less than 20 seconds to form a stable overlay of 10,000 nodes. Kademlia bootstraps quickly without any modification to its JOIN protocol. Chord, however, requires a slight modification to its JOIN protocol. We analyze the behavior of the unmodified Chord JOIN protocol in detail and illustrate its failure mode, which reveals a danger inherent in designing a p2p system that may be subject to extreme churn.

Chapter 3 presents *mDHT*, a novel architectural enhancement to DHT using multicast service discovery [94]. In mDHT, a group of host computers in a subnet participate in a DHT overlay as a single node. A query is routed from subnet to subnet until it reaches the final destination subnet, where it is resolved among the hosts using link-local multicast. Under a reasonable deployment assumption, mDHT offers many benefits over standard DHTs, such as locality, easy bootstrapping, high availability, and near imperviousness to node churn.

Chapter 4 presents the Zeroconf-to-Zeroconf Toolkit (z2z) [93]. Zeroconf, better known as Apple Bonjour, solves configuration and service discovery problems in local networks using link-local multicast. Our z2z interconnects multiple Zeroconf subnets, extending the reach of existing Zeroconf-enabled applications beyond the local link.

Chapter 5 of Part II introduces NetServ [91,92,112], the main contribution of this thesis. We begin by developing the economic motivation of our work: activating ISPs' edge routers for the purpose of hosting content providers' services. This motivation leads to our design goals, which in turn leads to the high level node architecture we discuss in this chapter. We also describe the end-to-end service scenario that underlies the NetServ applications that we have built. Detailed descriptions of the applications are deferred until Chapter 7.

Chapter 6 describes the NetServ node implementation. We describe in detail each component of a NetServ node–signaling daemons, NetServ controller, forwarding engine, service container, and the building block and application modules–and discuss how they interact. We also consider the security features of the implementation and provide performance evaluation.

Chapter 7 presents four NetServ applications: ActiveCDN, KeepAlive Responder, Media Relay, and Overload Control. The applications demonstrate the economic benefits for the content providers and the ISPs who enter into a cooperative relationship using NetServ. ActiveCDN provides provider-specific content distribution and processing. The other three applications illustrate how NetServ can be used to develop more efficient and flexible systems for real-time multimedia communication. In particular, we show how Internet Telephony Service Providers (ITSPs) can deploy NetServ applications that help overcome the most common problems caused by the presence of Network Address Translators (NATs) in the Internet, and how NetServ helps to make ITSPs' server systems more resilient to traffic overload.

Chapter 8 describes the OpenFlow extension of NetServ, which provides a solution for scalability. We give a brief overview of how OpenFlow works, and then describes how it is integrated with the NetServ node architecture that we have described in Chapter 6.

Chapter 9 compares NetServ with active networks. The idea to enable in-network services is not new. Active networking articulated the same vision more than a decade ago. In fact, active networks went even further, advocating the infamous *integrated* approach, where every packet can carry a program that can alter the behavior of network nodes. We explain how NetServ addresses the main challenges of active networks. We argue, in fact, that NetServ can be viewed as the first fully integrated active network system that provides all the necessary functionality to be deployable, addressing the core problems that prevented the practical success of earlier approaches.

Part I

Prelude: Improving Service Discovery in Local and Global Networks

Chapter 2

Creating Global Networks: Bootstrapping Large-scale DHT networks

2.1 Introduction

Structured overlay networks based on distributed hash tables (DHT) have become a popular substrate for large-scale peer-to-peer (p2p) systems. DHTs provide efficient lookup of distributed data by assigning numerical keys to the peer nodes and the data items, storing each data item at the node whose key is the closest to the item's key, and locating in a small number of hops the node responsible for any given key. In order to achieve the small number of hops–usually a logarithm of the number of nodes in the network–DHTs dictate overlay topology by imposing certain constraints on the entries in each node's routing table. In Chord [115], for example, the keys are m-bit integers arranged in a circle modulo 2^m, and each node's routing table is filled in such a way that the i^{th} routing table entry is at a distance of at least 2^{i-1} in the key circle, aka the *Chord ring*.

Maintaining such an overlay structure when nodes are joining and leaving the network very frequently–referred to as a high *churn* rate–has been the focus of much academic research. However, we found only a handful of studies that investigate the extreme case of churn, namely, bootstrapping a DHT from scratch by having all nodes join at the same time. The massive join scenario is not contrived. A campus-wide or city-wide power outage will result in a large number of computers starting up at the same time. In fact, a large fraction of computers connected to the Internet today routinely reboot nearly at the same time when they receive a periodic operating systems update. In August 2007, one such massive reboot triggered a hidden bug in the Skype client software, causing a world-wide failure of the Skype voice-over-IP network for two days [34]. A DHT-based overlay multicast system such as SplitStream [56] provides another example. A large number of users may join the overlay at the same time for a live streaming of a popular event.

A few proposals address the problem of bootstrapping a DHT from scratch [44, 45, 83, 100, 119], but they all forgo the standard DHT join protocols in favor of their own distributed algorithms that build routing tables directly. They motivate their algorithms by claiming that the standard join protocols are not designed to handle the huge number of concurrent join requests involved in bootstrapping a DHT from scratch. (Some even assume, incorrectly, that the join protocols expect nodes to be inserted sequentially into an overlay.) We find the claim unsubstantiated.

Another problem with the existing proposals is the assumption that they make about the initial state of the nodes before they start forming a DHT. Each node is assumed to hold a handful of pointers to a random selection of other nodes in the network. These pointers in effect provide an unstructured overlay network (termed a *knowledge graph* by [45]) as a starting point for the algorithms of the proposals. The algorithms thus take the approach of transforming a pre-existing unstructured overlay into a DHT. We question the validity of the assumption. One can argue that such a pre-existing overlay might be a reasonable model for the state of a DHT after a massive failure, where each node still has the old routing table. However, it is certainly not applicable to bootstrapping a DHT from scratch, which is the scenario that the proposals claim to address.

We take a step back and examine how the standard join protocols behave under a large number of concurrent join requests. We will use the term “JOIN protocol” to refer to the standard join protocol of a DHT, and “JOIN call” to refer to the remote procedure call to implement the protocol. When the meaning is clear from context, we will simply use

"JOIN". Our system model is simple and realistic: we assume that every node knows a single global bootstrap server which is responsible for handing out a contact node when asked. Each node asks the bootstrap server for a contact node, and then sends a message to the contact node to start a JOIN. The bootstrap server keeps a list of every node that has successfully joined the overlay, and picks one at random when it is asked to provide a contact node. We simulate bootstrapping a Chord [115] and a Kademlia [98] DHT where a large number of nodes join the network nearly simultaneously, at a rate of 1,000 nodes per second, and measure the time it takes to form a stable overlay. In contrast to the claim made by the existing proposals, our results indicate that both Chord and Kademlia handle a large number of concurrent JOINs quite well. For 10,000 nodes joining at a rate of 1,000 nodes per second, Chord and Kademlia took less than 20 and 15 seconds, respectively, to form a stable overlay. It should be noted, however, that Chord requires a slight modification to its join protocol, proposed by Baumgart *et al.* [50], to achieve the fast bootstrapping performance. We discuss this modification in detail in Section 2.4.3.

This chapter makes two contributions to p2p research. First, we call into question the claim that JOIN protocols cannot handle a large number of nodes joining simultaneously, and provide empirical evidence disputing that claim. Second, we elucidate the reason why the Chord JOIN protocol needs a modification to deliver fast bootstrapping. We demonstrate the failure mode of the unmodified Chord JOIN protocol, which is not only interesting, but also edifying for the designers of distributed systems that may be subject to extreme conditions.

The rest of this chapter is organized as follows. Section 2.2 summarizes the existing proposals and discusses the common assumption that they all seem to make. Section 2.3 describes our system model. Section 2.4 presents our simulation results and discusses the modification to the Chord JOIN protocol. Finally, we conclude and discuss future work in Section 2.5.

2.2 Related work

2.2.1 DHT construction proposals

Angluin *et al.* [45] present a series of distributed algorithms that build a linked list of all nodes sorted by their identifiers. The sorted list can then be used as a basis to construct DHTs that are based on linear overlay topologies such as Chord [115] or SkipNet [77].

For DHTs using fixed key space partitioning schemes such as P-Grid [43] or Pastry [107], Aberer *et al.* [44] propose a decentralized algorithm motivated by the requirements of peer-to-peer database applications.

Montresor, Jelasity and Babaoglu [100] describe a gossiping algorithm called T-MAN. Each node starts out with a *view*, which is just a handful of pointers to other nodes, and keeps exchanging its view with the neighboring nodes. The overlay topology defined by the views converges to that of Chord as each node refines its view to bring it closer to a Chord routing table when exchanging views with neighbors. Jelasity, Montresor and Babaoglu [83], and Voulgaris and van Steen [119], present similar gossiping algorithms that build a Pastry overlay instead.

2.2.2 Common assumptions

In [45], it is assumed that the nodes are initially organized as a weakly-connected *knowledge graph* of bounded degree d, where an edge from node u to node v indicates that u knows v's network address. A knowledge graph is therefore a model for an unstructured overlay network where each node has a handful of pointers to other nodes. In fact, all other proposals described in Section 2.2.1 make the same assumption about the initial state of the nodes, although the others do not use the term knowledge graph. Random walks on a pre-existing unstructured overlay network start the algorithm in [44]. The starting point for the gossiping algorithms in [83, 100, 119] is an unstructured overlay built by running another gossiping protocol (similar but separate from the one that builds DHT) called NEWSCAST [82] proposed by some of the same authors.

The NEWSCAST protocol can build a knowledge graph from scratch. The authors show that NEWSCAST will produce a sufficiently randomized knowledge graph even when every node is initialized with only a pointer to a single well-known node. (This is clearly the minimal assumption one can make.) There are a few other proposals that construct networks which can be considered knowledge graphs as well [90,101,118].

Thus, with all the proposals described in Section 2.2.1, building a DHT from scratch–that is, making no assumption except the existence of a single well-known node–actually becomes a two-step process that requires a knowledge graph to be built first using one of the knowledge graph construction proposals. This escalates the complexity and cost of the algorithms. And at the root of all the proposals, there is the premise that JOIN cannot handle massive concurrency involved in DHT bootstrapping. Our results challenge that premise.

2.3 System model

We simulate a scenario where a large number of nodes join a Chord or Kademlia overlay nearly simultaneously. Each node starts the bootstrapping process by contacting the single well-known bootstrap server. The bootstrap server, which has been keeping a list of every node that has successfully joined the overlay, responds to the newly joining node by handing out a *contact node* selected at random from the list. The newly joining node then initiates a JOIN call by sending a message to the contact node that was handed out by the bootstrap server. When the new node receives a response from the contact node indicating that the JOIN was successful, the new node contacts the bootstrap server once again, so that the new node can be added to the list of successfully bootstrapped nodes. We assume that initially the bootstrap server has one node in its list of successfully bootstrapped nodes, so that it has something to hand out from the very beginning. That is, we start our simulation with an overlay already containing a single node. In practice, the initial node can be the bootstrap server itself if the bootstrap server participates in the overlay.

The exact meaning of "a large number of nodes joining nearly simultaneously" is defined by setting the *join rate*, which is the average rate at which the nodes contact the bootstrap server to start JOIN. This is a reasonable model for the real-life situation where the incoming requests would be throttled by the maximum throughput of the bootstrap server. We use the join rate of 1,000 nodes per second, which will generate 2,000 hits per second on the bootstrap server because every joining node contacts the bootstrap server twice. This hit rate is well within the capability of today's servers. Root name servers were required to handle 1,200 requests per second more than a decade ago [97], and a measurement in 2002 reports an average rate of nearly 2,000 requests per second, with much higher loads at peak times [120].

Note that there is a significant latency from the time that a node starts a JOIN protocol until the time that the node is considered successfully bootstrapped. The latency consists of the initial round trip to the bootstrap server, the JOIN remote procedure call (which can involve multiple hops), and the final notification to the bootstrap server. This means that, in the beginning, all newly joining nodes will send their JOIN calls to the single first node, because that node is the only contact node that the bootstrap server can hand out until the list of bootstrapped nodes starts growing after the initial bootstrapping latency. This seems to create another potential bottleneck in addition to the bootstrap server. However, unlike the load on the bootstrap server which is an integral part of our system model, the load on the single first node can be controlled easily by starting out with an overlay containing many nodes rather than a single node. We start with an overlay containing a single node for simplicity.

After initiating a JOIN protocol, each node begins to send a test query to a random key every second on average. The interval between the successive test queries by a node is drawn from the normal distribution with a mean of 1 second and a standard deviation of 0.1 second, truncated to non-negative values. At another fixed interval, we collect the total number of test messages sent by all nodes during that interval and the total number of messages that have been delivered successfully during the same interval, and calculate the ratio which we call *delivery rate*. We measure the *convergence time* of an overlay, which we define as the time it takes for the delivery rate to reach 95%. We also make sure that the average hop count of the delivered messages is $O(\log N)$ because an overlay should not be considered stable if messages are delivered in $O(N)$ hops. Since we focus on the bootstrapping stage where all nodes are just joining the overlay, we assume that the nodes stay alive after they join the overlay; i.e., we assume that there is no churn.

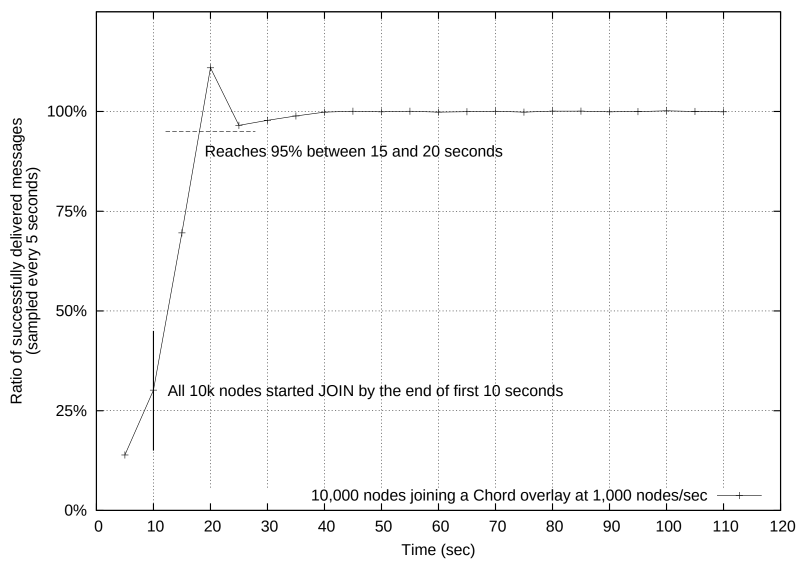
2.4 Simulation results

2.4.1 Simulation setup

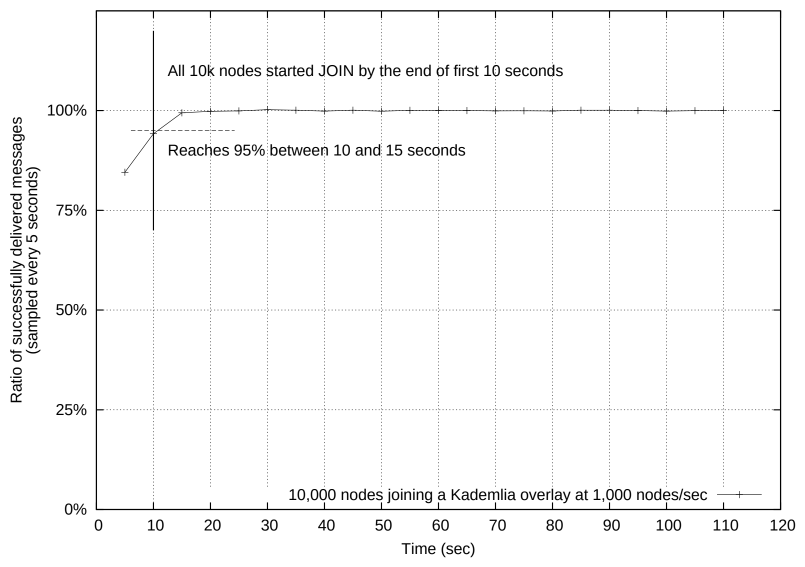
We used the OverSim simulator [50], version 20080919 [23]. Conveniently, the DHT implementations included in OverSim follow a bootstrapping procedure very close to that of our system model. A singleton object called the BootstrapOracle assumes the role of our bootstrap server, except that the BootstrapOracle is a global C++ object accessed by the nodes without incurring any network delay. We modified the Chord and Kademlia implementations to include communication delays when nodes access the BootstrapOracle. The join rate is controlled by a configuration parameter, which we set to 1,000 nodes per second. OverSim also includes a test application called KBRTestApp. We modified it to produce our measurements.

Between any pair of nodes, we used a constant packet delay of 50 milliseconds, plus a random jitter of around 10%. The random jitter was drawn from the normal distribution with a mean of 0 and a standard deviation of 5 milliseconds, truncated to non-negative values. This is a conservative estimate of the average packet delay experienced in the Internet today. We do not explicitly model the processing delays in each node, which should be nearly negligible, but not zero. We consider that the processing delays are absorbed into the packet delays.

Unless noted otherwise, we used the following configuration settings for Chord. We used iterative routing, and the successor list size was set to 8. When a JOIN call fails, a node waits 5 seconds before it sends another. The *stabilize*() maintenance procedure (STABILIZE) and *fix_fingers*() maintenance procedure (FIXFINGERS) are run immediately after a node successfully joins the overlay, and afterwards they are run every 5 and 10 seconds, respectively. The STABILIZE procedure updates a node's immediate successor and the FIXFINGERS procedure updates all other routing table entries. We found the STABILIZE and FIXFINGERS intervals of 5 and 10 seconds to be optimal for bootstrapping, but in general, the intervals would be considered somewhat aggressive. A real-life implementation can use an adaptive approach, where it switches to longer intervals once the overlay reaches a stable state.



(a)



(b)

Figure 2.1: (a) Delivery rate for 10,000 nodes joining a Chord overlay at the rate of 1,000 nodes per second. Convergence time is 20 seconds. (b) Delivery rate for 10,000 nodes joining a Kademlia overlay at the rate of 1,000 nodes per second. Convergence time is 15 seconds.

For Kademlia, we set $k = 16$, $\alpha = 1$, and $b = 1$. That is, the k-bucket size was set to 16, no parallel queries were used, and the optimization technique of considering IDs b bits at a time described in Section 4.2 of [98] was not used. OverSim's Kademlia implementation uses a sibling list, an extension described in [51]. We set the sibling list size to 8.

The simulations were performed on a PC equipped with Intel Core 2 Duo CPU and 4 GB RAM running Ubuntu Linux 8.04, except the simulations involving more than 50,000 nodes, which required more RAM. Those were run on a similar PC with 16 GB RAM.

2.4.2 Convergence time

Figure 2.1 shows our main result. It plots the delivery rates, sampled at every 5 seconds, when 10,000 nodes join a Chord and a Kademlia overlays at a rate of 1,000 nodes per second. Note that a delivery rate is simply the ratio of the number of successfully delivered messages to the number of messages sent in a given interval. Those messages that are sent and received across the interval boundaries add jitter to the delivery rates, and that is why some data points go over 100% in Figure 2.1. The join rate of 1,000 nodes/sec implies that all 10,000 nodes have started the JOIN calls after about 10 seconds. Figure 2.1(a) shows that it takes 10 more seconds for the nodes to form a stable Chord overlay that correctly routes over 95% of the queries, resulting in a fast convergence time of only 20 seconds. Figure 2.1(b) shows that Kademlia converges even faster, taking less than 15 seconds to reach the 95% delivery rate.

Convergence time remains fast when we increase the size of the overlay network. Figure 2.2 plots the increase in convergence time as we keep doubling the number of nodes. Convergence time seems to increase very slowly against the overlay size. The convergence time for Chord increases only to 45 seconds when the network size reaches 128,000 nodes. For Kademlia, the curve stays almost flat under 15 seconds.

The superior performance of Kademlia can be attributed to the symmetric nature of its routing architecture. In Kademlia, a node receives queries from the same nodes that it would contact when it wants to send a query. This enables Kademlia to use the information contained in regular queries to maintain the routing table, eliminating the need to have separate maintenance protocols. Every node sends a query every second in our simulation,

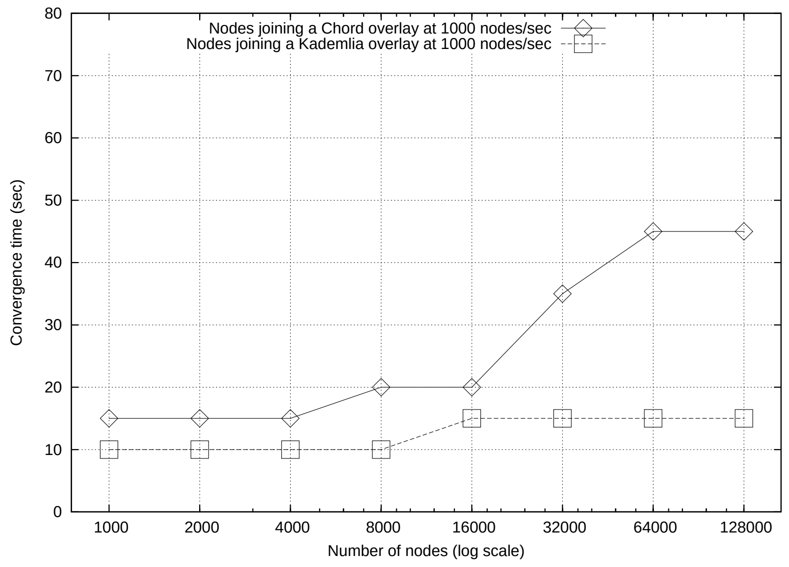


Figure 2.2: Increase in convergence time as the overlay size doubles.

resulting in a large number of messages from which the Kademlia nodes can extract useful information to adjust their routing tables. Chord does not have this symmetry, hence the need for the separate stabilization protocols, STABILIZE and FIXFINGERS.

For the large overlay sizes, a stable overlay can form before all nodes have started the JOIN calls. It takes 128 seconds for all 128,000 nodes to start the JOIN calls. In both Chord and Kademlia, stable overlays form well before all nodes have joined, and the overlays remain stable as the rest of the nodes join them.

2.4.3 Aggressive join

Our Chord simulations in Section 2.4.2 use a JOIN protocol called *aggressive join*, which contains a slight modification to the JOIN protocol described in the original Chord paper. Aggressive join was proposed by Baumgart *et al.* [50] and included in the OverSim's Chord implementation [23].

A node n joins a Chord ring by asking an existing node in the ring to find n's successor. In other words, n makes a JOIN call to an arbitrary contact node. Figure 2.3(a) shows the successor (solid line) and predecessor (dotted line) pointers among the three nodes, p, n and s, right after n has found its successor s. In the original Chord specification, the JOIN

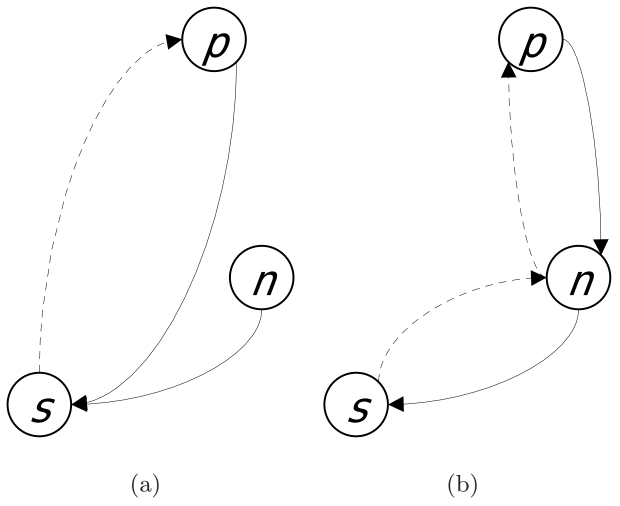


Figure 2.3: Node n joins a Chord ring between its successor and predecessor.

protocol stops here. Fixing up the rest of the successor and predecessor pointers to make the node n a full participant of the ring, as depicted in Figure 2.3(b), is delayed until the next STABILIZE cycle. The pseudocode for JOIN and STABILIZE from the Chord paper is reproduced in Figure 2.4.

Aggressive join, on the other hand, proceeds to complete all the pointers between the three nodes immediately after JOIN:

$$s.predecessor = n$$
$$n.predecessor = p$$
$$p.successor = n$$

When s receives a join request from n (or when the FIND_SUCCESSOR call initiated by n's JOIN call terminates at s, to be more precise), s immediately sets its predecessor to n, s then includes p in its join response message to n, enabling n to set its predecessor to p, and finally s sends a message to its old predecessor p, indicating that the newly joining node n would be a better successor for p.

It turns out that aggressive join is essential for fast convergence. Figure 2.5 shows convergence times when we disabled aggressive join. With our standard STABILIZE and FIXFINGERS intervals of 5 and 10 seconds, it took 4,500 seconds to form a stable overlay.

```
// create a new Chord ring
n.create() {
  predecessor = nil;
  successor = n;
}

// join a Chord ring containing node n'
n.join(n') {
  predecessor = nil;
  successor = n'.find_successor(n);
}

// called periodically:
// verifies immediate successor of n
// and tells the successor about n
n.stabilize() {
  x = successor.predecessor;
  if (x in (n,successor))
    successor = x;
  successor.notify(n);
}

// n' thinks it might be our predecessor
n.notify(n') {
  if (predecessor is nil
      or n' in (predecessor,n))
  predecessor = n';
}
```

Figure 2.4: Pseudocode for the original Chord stabilization protocols, reproduced from the Chord paper.

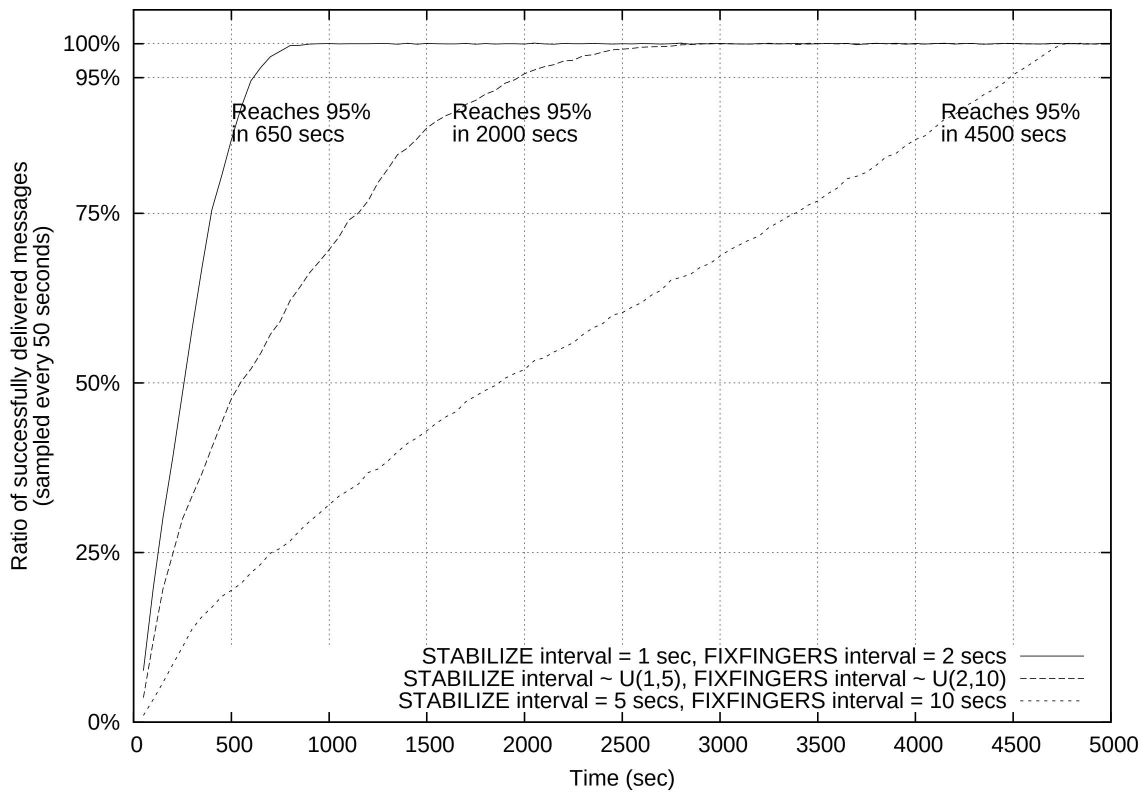


Figure 2.5: Delivery rates for three different STABILIZE and FIXFINGERS intervals, with aggressive join disabled. (1,000 nodes joining at the rate of 1,000 nodes/sec in all three cases.)

When we tried to compensate for the lack of aggressive join by running STABILIZE and FIXFINGERS extremely frequently–STABILIZE every second and FIXFINGERS every two seconds–we were able to bring the convergence time down to 650 seconds, but it is still orders of magnitude slower than the 15 seconds convergence time with aggressive join enabled. Randomizing the intervals did not improve the convergence time. When we set each STABILIZE interval from a uniform distribution between 1 and 5 seconds and each FIXFINGERS interval between 2 and 10 seconds, the convergence time was 2,000 seconds, indicating no significant benefit from randomization.

Figure 2.6 illustrates the circumstance under which bootstrapping a Chord network without aggressive join suffers from a slow convergence. It is a topology snapshot of 250 nodes trying to bootstrap a Chord network, taken at 50 seconds after they all started

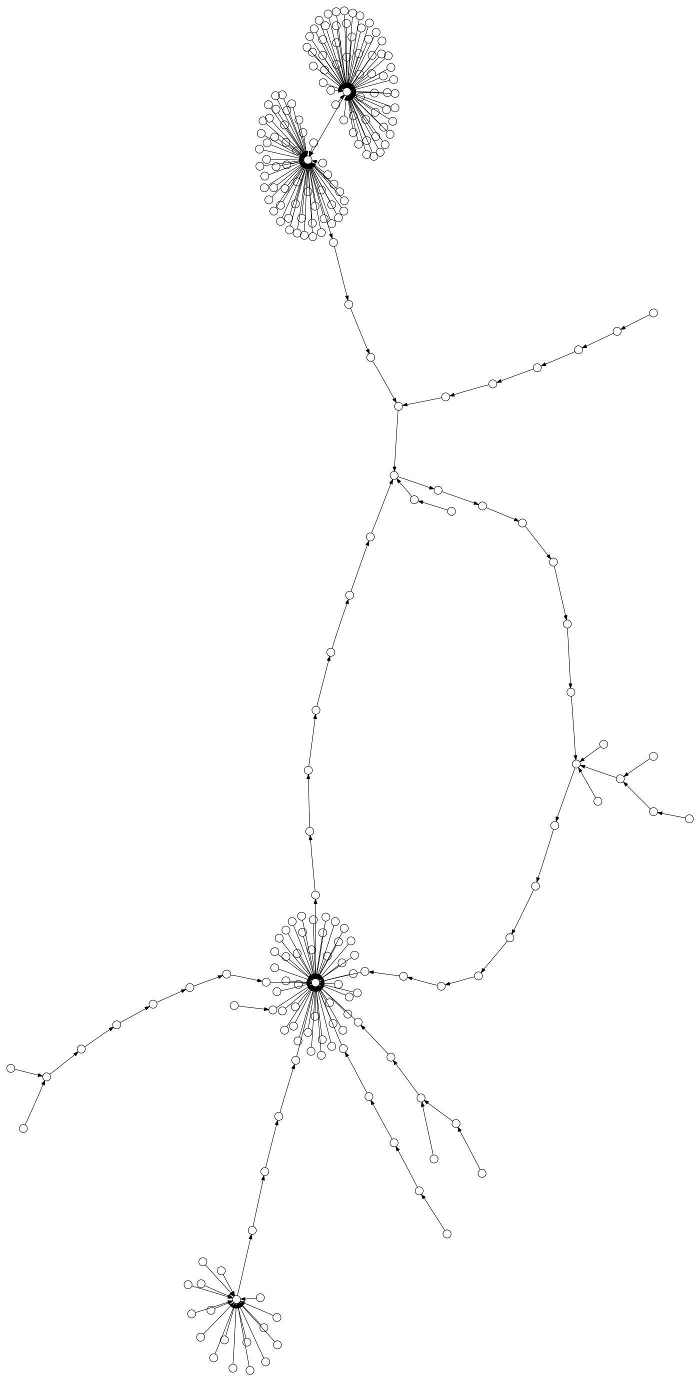


Figure 2.6: Topology snapshot of 250 nodes forming a Chord network, taken at 50 seconds after they all started JOIN.

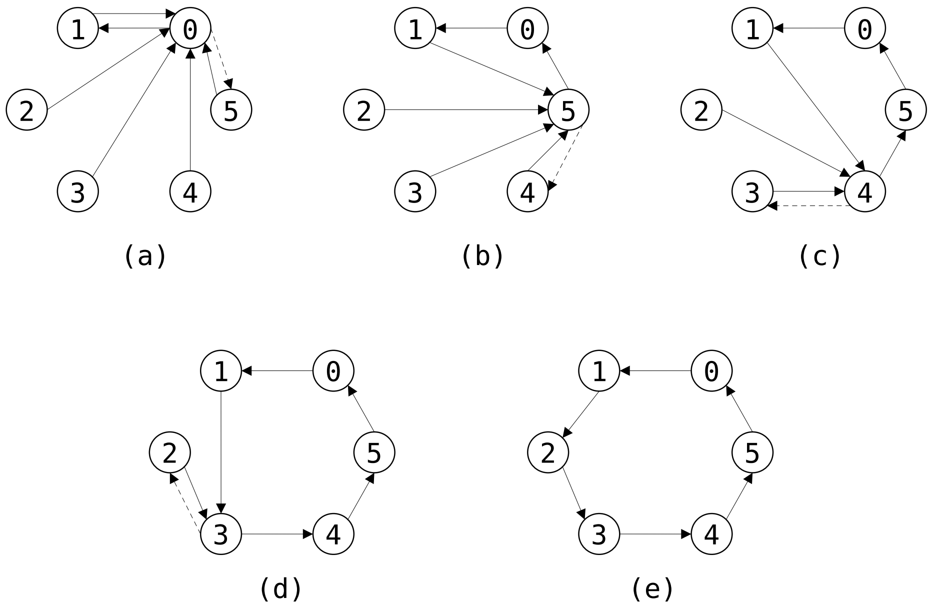


Figure 2.7: An extremely slow process by which a hub turns into a ring.

JOIN nearly simultaneously (at the rate of 1,000 nodes/sec.) The arrows indicate successor pointers. The massive number of concurrent join requests, combined with the fact that the standard Chord JOIN protocol delays fixing successor and predecessor pointers, resulted in a tree-like topology containing long chains and high in-degree hubs.

Without aggressive join, when fixing successors and predecessors are left to the STABILIZE procedure, such chains and hubs converge to a ring extremely slowly. Figure 2.7 illustrate the process for a hub. Nodes 1 to 5 all have their successors set to node 0, and thus send NOTIFY calls to node 0, telling node 0 that they might be 0's predecessors. As node 0 receives NOTIFY calls from different nodes, it keeps updating its predecessor, eventually settling on node 5, since node 5 is the best predecessor for node 0 among nodes 1 to 5. This is shown in Figure 2.7(a). During the next STABILIZE cycle, nodes 1 to 4 discover that node 0 has a new predecessor, and they all set their successors to node 5 because node 5 is a better successor for them than node 0, as shown in Figure 2.7(b). Node 5 then receives NOTIFY calls from node 1 to 4, and sets its predecessor to node 4, which in turn causes node 1 to 3 to flock to node 4 during the next STABILIZE cycle, as shown in Figure 2.7(c). This process continues until the nodes forms a ring, as in Figure 2.7(e). From

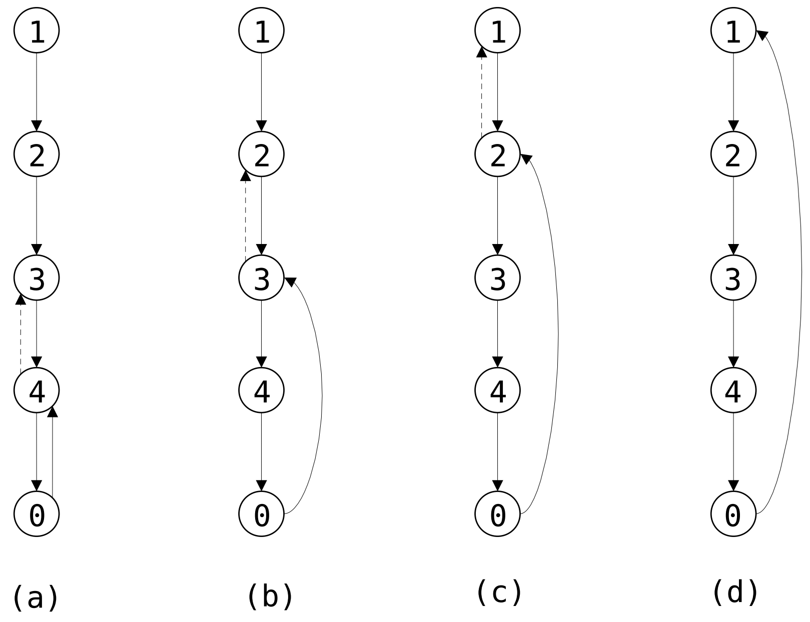


Figure 2.8: An extremely slow process by which a chain turns into a ring.

the STABILIZE algorithm in Figure 2.4, we see that a successor is changed only when the previous successor acquires a new predecessor, and that happens when NOTIFY is called. But in a hub topology, of all the NOTIFY calls sent by the nodes pointing to the central node, only one of them will have an effect, yielding just one chance for a successor change. Since the size of the ring can never be greater than the number of unique successors, at each iteration, the size of the ring increases only by one.

Figure 2.8 illustrates a similar process by a long chain rather than a hub. As the sequence of predecessor changes works its way backward towards the end of the chain, the ring gets expanded one node at a time. Note that a chain will necessarily have the nodes lined up in the order of their IDs, due to the fact that a JOIN process is finding the successor node and setting the successor pointer to it.

Aggressive join sets successors and predecessors immediately to the freshly joined nodes, therefore increasing the number of unique successors as quickly as possible. This behavior is enough to break the previous pattern, as evidenced by the fast bootstrapping performance.

The fact that aggressive join effectively solves the problem of the unmodified Chord JOIN protocol provides a clue as to why Kademlia does not suffer from the same problem.

Recall that Kademlia does not use separate protocols for maintaining routing tables; instead it simply uses the regular query traffic for the purpose. Kademlia's JOIN protocol is simply a series of lookup queries, from which the corresponding nodes update their routing tables as needed. In effect, Kademlia's JOIN protocol is already as "aggressive" as it can be. The lesson for a distributed systems designer seems to be that a decision to defer topology corrections to a later time should be carefully vetted to make sure there is no adverse effect, especially when the system may be subject to a high churn environment.

2.5 Conclusion

It has been claimed that the standard DHT join protocols are not adequate for bootstrapping large DHT networks from scratch. We debunk the claim by showing that Chord and Kademlia DHT take less than 20 seconds to form a stable overlay of 10,000 nodes. Kademlia bootstraps quickly without any modification to its JOIN protocol. Chord, however, requires a slight modification to its JOIN protocol. The modified JOIN protocol, called aggressive join, fixes the successor and predecessor pointers immediately after a node has joined, as opposed to waiting until the next STABILIZE cycle. We analyze the behavior of the unmodified Chord JOIN protocol in detail and illustrate its failure mode, which reveals a danger inherent in designing a p2p system that may be subject to extreme churn.

Chapter 3

Enhancing Global Networks Using Local Networks: Multicast-augmented DHT

3.1 Introduction

We have witnessed two significant advances in peer-to-peer (p2p) networking technology in recent years, driven by the consumers' desire to interconnect at the two opposite ends of networking scale. On the global scale, distributed hash tables (DHTs) [98, 107, 114] solved the scalability problem of the Internet-wide overlay networks. DHTs impose certain structures into the overlay topologies in order to achieve logarithmic-time lookup of a resource in the network. On the local scale, Zero Configuration Networking (Zeroconf) [39] all but eliminated the need to configure applications and devices to discover and talk to each other in a same subnet. Zeroconf implements service discovery by exchanging link-local multicast packets.

This chapter describes an architectural enhancement to DHT using Zeroconf multicast, which we call *mDHT*. In mDHT, the meaning of a "node" is changed from an individual host computer to an entire subnet, i.e., an entire subnet participates in a DHT overlay as a single node. A lookup query is routed from subnet to subnet in mDHT until it reaches the subnet for which the query is destined. The query is then resolved within the subnet using multicast.

Our mDHT architecture can be applied to any existing DHT system. Under a reasonable deployment assumption (the validity of which we reinforce with a measurement of an existing p2p system), mDHT offers numerous benefits including locality, load balancing, easy bootstrapping, high availability, and near imperviousness to node churn.

The rest of this chapter is organized as follows. We give background information on hierarchical DHTs and Zeroconf technology in Section 3.2. We delve into mDHT architecture in Section 3.3, starting with an overview and then describing each aspect of mDHT architecture in detail. In Section 3.4, we discuss the benefits of mDHT, and analyze our deployment assumption using a measurement of a real-world p2p system. We conclude in Section 3.5.

3.2 Background

3.2.1 Evolution of P2P Architecture

The p2p systems architecture has evolved continuously in order to accommodate the demand of ever-increasing scale of p2p overlay networks. The scalability limit of flat unstructured p2p systems such as the early version of Gnutella, which performed content lookup by flooding the network, inspired the development of structured p2p systems based on DHTs. A DHT network is characterized by an efficient algorithm to map an arbitrary string to a particular node in the network and to produce a routing path of a bounded number of hops from any node to that node. The mapping is deterministic and results in a balanced distribution of the strings among the participating nodes. This enables efficient lookup of distributed data items when each data item is associated with a string (a file name, for example) and the item is stored in the node to which the associated string maps.

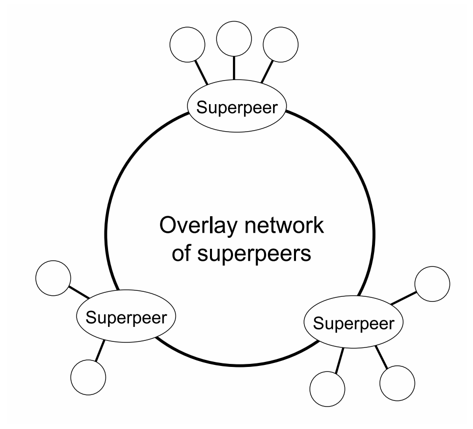
Another way to overcome the scalability limit of flat unstructured systems was to introduce hierarchy. In a two-tier hierarchical organization used in the later versions of Gnutella [13], the core overlay network is formed not by every participating node, but by a selected subset called *superpeers*. Each non-superpeer maintains a connection to a superpeer who will act as a gateway to the services offered by the overlay. Figure 3.1(a) illustrates this arrangement.

The idea of hierarchical overlay can be applied to DHTs as well as unstructured networks, and the depth of hierarchy can be more than just two levels. Many system proposals exhibit complexities that go well beyond that of the simple two-tier superpeer architecture [46, 70, 71]. In addition to the obvious advantage of reduced overlay size, hierarchical designs can offer various other advantages, such as taking account of physical network and node heterogeneity, better resilience to churn, administrative control and autonomy, and more efficient caching and load balancing strategies. These advantages apply both to DHTs and unstructured networks, but they are especially beneficial to DHTs, since the rigid overlay structures of DHTs make it harder to incorporate those considerations into flat overlays. Those hierarchical designs that are more complex than the simple two-tier superpeer architecture tend to focus on maximizing performance in one or two of those areas. The simple two-tier architecture, however, still provides some benefits in all those areas compared to the flat overlay design. Moreover, the simplicity of the two-tier architecture is an important advantage over other more complex hierarchical designs, when it comes to developing, deploying and maintaining a large scale overlay network. For these reasons, the simple two-tier superpeer architecture of Figure 3.1(a) is often the preferred choice for p2p networks on the Internet [16, 52].

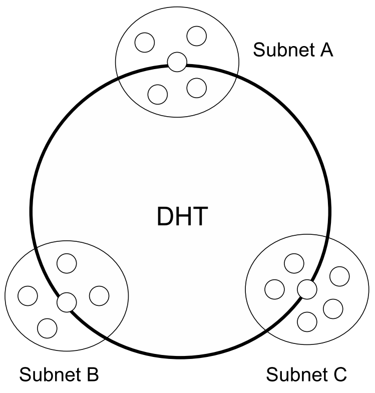
3.2.2 Zeroconf: Local Service Discovery Using Multicast

When multiple IP-enabled devices are physically connected with one another, Zeroconf makes it possible for one device to use the services provided by another without requiring the user to configure the devices manually. For example, when a user connects two computers either directly using an Ethernet crossover cable or via an Ethernet switch, he will be able to accomplish his file-transfer task by simply starting up the appropriate Zeroconf-enabled applications at both ends. The applications use Zeroconf technology to *discover* each other, without the user having to furnish them with the connection information such as IP addresses and port numbers.

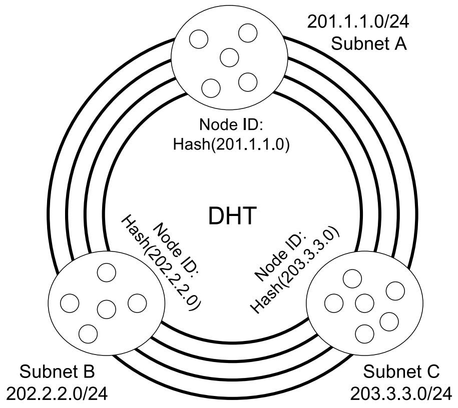
Zeroconf performs local service discovery by exchanging DNS packets via link-local



(a)



(b)



(c)

Figure 3.1: (a) Two-tier superpeer architecture. The overlay network of superpeers can be an unstructured network or a DHT. (b) A multicast-based superpeer architecture. (This is not our mDHT.) A superpeer is a single point of failure in a subnet. (c) Our mDHT architecture. A subnet is a node in a DHT. The node IDs are chosen by hashing the subnet IP addresses.

multicast, the details of which are described in a pair of specifications, DNS-based Service Discovery (DNS-SD) [58] and Multicast DNS (mDNS) [59]. DNS-SD defines a set of naming rules for certain DNS record types that it uses for advertising and discovering services. PTR records are used to enumerate service instances of a given service type. A service instance name is mapped to a host name and a port number using a SRV record. If a service instance has more information to advertise than the host name and port number, the additional information is carried in a TXT record.

The DNS records are stored in a collection of mDNS daemons, which are limited-functionality DNS servers running on each host in a local subnet. The mDNS daemons collectively manage a special top-level domain, ".local.", which is used for names that are meaningful only in a local subnet. The queries and answers are sent via link-local multicast using UDP port 5353 instead of 53, the conventional port for DNS. An application can advertise a network service to the subnet by creating appropriate DNS records and depositing them into the mDNS daemon running on the same host. The mDNS daemon will then respond with these records when it hears a multicast query for a matching service. Creating DNS records and storing them with mDNS are usually done by invoking API calls in a DNS-SD/mDNS client library implementation.

Zeroconf technology is widespread today. Bonjour, Apple [3]'s Zeroconf implementation, is an integral part of Mac OS X operating system. Bonjour is also installed on a large fraction of computers running Windows, thanks to the popularity of iTunes—Apple's music playing application—which installs Bonjour for Windows as part of its installation process. For UNIX-like platforms, there is a mature open-source implementation of Zeroconf called Avahi [4], which comes preinstalled in a number of major Linux distributions such as Ubuntu [33].

3.2.3 Multicast-based Superpeer Architecture

Figure 3.1(b) illustrates a possible hybrid of Zeroconf and superpeer-based DHT. A node in a subnet is elected as a superpeer and participates in the DHT. The other nodes in the subnet learn the identity of the superpeer through the superpeer's Zeroconf service announcement, and therefore are able to access the DHT through the superpeer.

At first glance, this architecture seems to offer a reasonable alternative to the regular superpeer architecture of Figure 3.1(a) if we assume that, on average, a significant number of nodes are found in a single subnet. It is unclear, however, that the use of multicast provides much benefit at all. Moreover, a major weakness of superpeer architectures is still present: the superpeer is a single point of failure among the nodes attached to that superpeer. We will not consider this model any further. It is presented here as a conceptual bridge leading to our mDHT architecture, and to ensure that the reader does *not* conjure up this model as his or her mental image of mDHT.

3.3 mDHT Architecture

3.3.1 Overview

Figure 3.1(c) illustrates our mDHT architecture. An entire subnet, not an individual host, becomes a "node" and participates in a DHT. A node identifier (node ID) must be assigned to a subnet as a whole, so it cannot be based on an IP address of any individual host. Figure 3.1(c) shows one way to assign node IDs on subnet level: node IDs are chosen by hashing subnet IP addresses. Other methods of ID assignment can be used as long as a single ID is assigned to an entire subnet and that ID is propagated to all participating hosts in the subnet.

Messages are routed in the same way they are routed in regular DHTs. A query is routed among the nodes until it reaches its destination according to the particular DHT algorithm used in the overlay. In mDHT, a node is a subnet. Once a query reaches the destination subnet, the query is resolved among the hosts in the subnet using link-local multicast.

Our mDHT can be applied to any DHT since its operation depends only on the generic facilities common to all DHTs such as routing tables or node identifiers. Nevertheless, it is often helpful to use a specific DHT when referring to a part of data structure or a maintenance procedure, since terminology varies across DHTs. In those cases, we use Chord [114]. Also, when we use the term *node*, we mean a logical entity that is given a node ID, which is a host computer in a regular DHT, but a subnet in mDHT. We use the term *host* to refer to individual computers.

3.3.2 Routing Table

We explained that a query is routed in mDHT just as it is routed in a regular DHT, passing through intermediate nodes and finally reaching the node to which it is destined. Once the query arrives in the destination subnet, it is resolved among the hosts in the subnet using multicast. But how does the message travel from a node to another when a node is a subnet, not a host? Message transfer is still based on TCP/IP networking, and there is no such thing as sending a message to a subnet.

Suppose a subnet *A* is a node in a mDHT overlay and, among the hosts in the subnet *A*, three hosts *a1, a2, a3* have joined the overlay. (The three hosts share a single node ID, *hash(A)* for example, as explained in Section 3.3.1.) Suppose a subnet *B* is also a node, with hosts *b1, b2, b3* participating. Imagine that *a1* has issued a query, and the DHT algorithm has determined that the node *B* is the next hop. In order for *a1* to send a message to the subnet *B*, it needs to know at least one specific host in that subnet. Let's assume that *a1* knows that *b1* and *b2* reside in *B*. The host *a1* randomly picks one of the two hosts in *B*, say *b2*, and sends the message. If *B* is the final destination for the message, *b2* will switch to multicast to resolve the query in its subnet. If not, it will find the next hop, say *C*, and proceed in the same way as *a1* did before. (This is assuming that recursive query routing is used in the DHT; if iterative routing is used, *b2* will tell *a1* where the next hop is and *a1* will repeat the procedure.)

Each host in a DHT carries a routing table that has a list of nodes. In Chord, a routing table entry (called a *finger*) points to a host node, and it consists of the node's ID, IP address and port number. In mDHT, a node is a subnet, and the routing table entry for a node needs to include a *host set*, the IP addresses and port numbers of the participating hosts in the subnet. For example, a mDHT implementation based on Chord may redefine the finger as a collection of the following information: SHA-1 hash of a subnet IP address as a node ID, the subnet IP address, and a host set of maximum 8 IP addresses and port numbers.

In our example of *a1* sending a message to *B*, *a1* randomly picked one host from the host set for *B*. When iterative query routing is used, there is another option. The same message can be sent to multiple hosts in the host set. In the example, *a1* can send the query to both *b1* and *b2*, and take the faster response. This will shorten the overall lookup latency by reducing timeout delays from failed hosts. It will also help maintain the host sets, as unresponsive hosts can be removed from them. This mechanism is similar to sending parallel queries to multiple adjacent nodes, available in some DHTs such as Kademlia [98]. The difference is that, in mDHT, parallel queries are sent to a *single node*.

3.3.3 Host Set Maintenance

The host sets in the routing tables need to be periodically updated. The authoritative list of active hosts in a subnet comes from the hosts in the subnet themselves. Each host monitors the multicast announcements sent by other hosts as they join and leave the overlay, and keeps track of the list of active hosts in the subnet. Zeroconf API makes this easy.

This list of active hosts in a subnet is propagated to all the routing table entries that point to this subnet using a combination of push and pull methods. Every host periodically refreshes its own routing table entries by contacting one of the hosts for an up-to-date list of active hosts. This can be incorporated into the regular DHT maintenance procedures such as Chord's FIX_FINGERS(). An updated list can also be pushed, on a join or leave event in a subnet, onto the neighboring nodes such as Chord's successor and predecessor.

3.3.4 Host Join and Leave

When there is no other participating host in a subnet, host join and leave in mDHT follow the same procedures of regular DHTs.

When a subnet already contains one or more participating hosts, joining and leaving the mDHT overlay from that subnet is almost trivial. A newly joining host simply makes a multicast announcement. The other hosts, which are monitoring the multicast announcements, add the new host into their lists of active hosts, which will eventually find their way to the routing tables of other nodes (Section 3.3.3). A leaving host also makes a multicast announcement, telling the other hosts to remove it from their lists of active hosts. In addition, a host may need to transfer data when joining or leaving, according to the data replication policy in place (Section 3.3.5).

3.3.5 Data Replication in Subnet

When DHT is used as a data storage and lookup facility, a data item is mapped to a particular node by the DHT algorithm. In the case of mDHT, where a node is a subnet, there is a question of which host (or which set of hosts) will store a data item mapped to the subnet.

The simplest strategy is to have one host store a particular data item, most likely the host that happened to receive the initial message for depositing the data item into the DHT. If a lookup request for that data item arrives at a different host in the subnet, it will issue a multicast query, to which the host owning the data will respond. When the host owning the data item leaves the overlay, it must transfer the data item to another participating host. This strategy does not provide any shield against host failures at mDHT level. However, the data replication schemes of regular DHTs such as Chord's successor-list can still be used.

On the opposite end, another strategy is to replicate all data items fully within the subnet. When a host receives a new data item, it is immediately propagated to all participating hosts in the subnet. This results in a high data injection cost, but the lookup latency caused by the multicast query is eliminated. A host failure is not a problem as long as there is another participating host in the subnet. On the other hand, a newly joining host needs to copy all existing data from a neighbor.

The optimal strategy is likely to be somewhere in the middle. A simple, yet reasonable approach might be to try to replicate data in a fixed number of hosts. Obviously the number represents a maximum since there may not be as many participating hosts in the subnet. Another possible strategy is to start with a low number of replicas and increase the number per data item as the node receives more and more lookup queries for the item.

3.4 Discussion

3.4.1 Benefits of mDHT

3.4.1.1 Immunity to Churn

High rate of churn—the continuous process of hosts joining and leaving an overlay—has been a difficult problem in DHT design. In mDHT, as long as there are other participating hosts in a subnet, hosts joining and leaving in that subnet has *no effect* on the DHT structure. The subnet remains as the same node in the DHT as individual hosts come and go, and there is no need to fix anything in the DHT structure.

Contrast this with the superpeer architecture we saw earlier in Figure 3.1(a). A superpeer represents a single point of failure among the nodes attached to it. If a superpeer leaves the overlay, presumably one of the non-superpeers can step up to become a new superpeer to hold the same group together, but even then, the new superpeer needs to be repositioned in the DHT overlay because its node ID is different from that of the previous superpeer.

3.4.1.2 High Availability

Many DHTs use data replication and parallel queries to increase the availability of data items stored in a DHT as a whole. Achieving high availability of a specific node, however, is not so straightforward. Our mDHT, on the other hand, can increase the availability of a node simply by adding more hosts to the subnet. This ability to strengthen a specific node would be particularly useful when it is used with a DHT that also provides administrative autonomy and controlled data placement, such as SkipNet [77].

3.4.1.3 Easy Bootstrapping

The use of multicast makes it easy for a new node to discover and join an mDHT overlay when there is another participating host already present in the subnet. This may reduce the load on the global bootstrapping servers in some systems, since only the first participating host in a subnet needs to contact a bootstrapping server.

3.4.1.4 Parallel Queries and Load Balancing on Single Node

As explained in Section 3.3.2, the fact that a node is a subnet consisting of multiple hosts enables mDHT to send parallel queries to a single node, as opposed to Kademlia's parallel queries which are sent to multiple adjacent nodes.

Within a subnet in mDHT, the participating hosts naturally share the load, since a random subset of the hosts receive each query. This is again in contrast with the superpeer architecture of Figure 3.1(a), where a superpeer represents a single point of failure and possibly a bottleneck.

There is a subtle issue, however, when we consider load balancing among all the hosts in the whole overlay. The fact that a node in mDHT is a subnet, and therefore contains a varying number of hosts in it, may conceivably result in a load deviation worse than those of standard DHTs. We conjecture that a DHT load balancing strategy such as [53] will be as effective on mDHT as it is on standard DHTs. Analysis and simulation to support this conjecture is planned as a future work.

Another point to note is that DHT load balancing is not such a serious problem in practice, since the expectation is that most hosts will only consume a small fraction of the host's resources. A more important task is mitigating "hot spots" caused by exceedingly popular items. Standard DHTs use replication to deal with hot spots. Our mDHT can do the same. In addition, node redundancy can be increased when the DHT algorithm allows controlled data placement (Section 3.4.1.2).

3.4.1.5 Awareness of Physical Proximity

A mDHT node represents a grouping of p2p participants in closest proximity to one another. A p2p application built on top of mDHT can take advantage of this locality property in various ways. For example, a file sharing application can reduce data traffic by having a host computer cache popular contents, not only for the purpose of repeated retrieval, but for making it available for the other hosts in the subnet. The cache inventories can then be exchanged via multicast among the hosts.

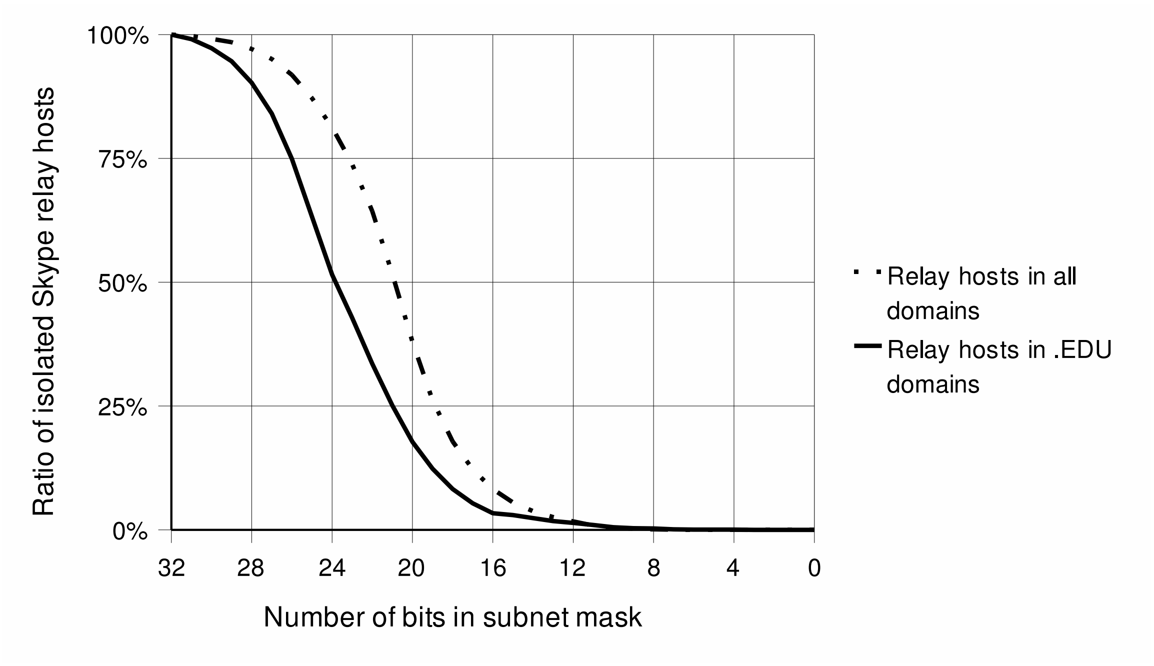


Figure 3.2: Ratio of the hosts that are participating alone in their subnets.

3.4.2 Analysis of Assumption

The claimed benefits of mDHT depend on the assumption that the majority of the subnets contain multiple hosts participating in the overlay network. In order to test this assumption on a real-world p2p network, we examined the IP addresses of 9582 Skype relay hosts, which were collected as part of an experiment by Kho *et al.* [88], measuring Skype relay calls over a three-month period.

Since it is difficult to determine the subnet mask of an arbitrary IP address, we simply fix a hypothetical subnet mask of certain number of bits, and group those IP addresses that fall into the same subnets under the fixed subnet mask. Figure 3.2 shows the ratios of the *isolated hosts*, the hosts that are alone in their subnets, as we vary the subnet mask from 32 to 0 bits. (The 32-bit subnet mask is the vacuous one where each host IP address becomes its own subnet; and the 0-bit mask is the degenerate one where the whole Internet is a single subnet.) The dotted line plots a result for the whole data set of 9582 hosts, and the solid line for a subset of 2150 hosts that belong in the .EDU domain.

The result from the .EDU hosts (the solid line) supports our assumption. With 24-

bit subnet mask (i.e., /24 subnet), half of the hosts have at least one other host in their subnets. Moreover, subnets in University campuses tend to be larger than /24. The 21-bit mask reduces the ratio of isolated hosts down to 25%.

It is harder to justify our assumption if we include not only the .EDU hosts, but all the hosts in the data set (the dotted line). It takes the 21-bit mask to achieve 50%, and the 19-bit mask to get down to 25%. It should be noted, however, that the majority of the IP addresses in this data set belong in the domains of residential ISPs [88], indicating that they are home computers. The access networks of residential ISPs are not likely to allow multicast between subscribers, so they represent hostile environments for mDHT. But we observe two encouraging trends that point to a future direction of residential networks that is more favorable to mDHT. First is the rise of home networks. Many households have multiple networked devices, and the use of a NAT router to form a home network is becoming commonplace. (The residential IP addresses in our data set do not include any host behind NAT, since Skype does not choose such a host as a relay.) Second, residential ISPs are increasingly concerned about the traffic generated by p2p applications, and they are looking for ways to reduce the traffic [14]. A residential ISP can reduce the traffic generated by a mDHT-based application simply by enabling multicast among the users in a neighborhood, so that they can share content cache as described in Section 3.4.1.5.

3.5 Conclusion

We presented mDHT, a novel hybrid architecture that augments DHT with multicast service discovery. Our mDHT shares some of the positive traits of the traditional two-level superpeer architecture, but we eliminate the single point of failure in a peer cluster. In addition, the redundancy within a node makes mDHT impervious to churn, and offers an easy way to increase node availability.

Chapter 4

Extending Local Services to Global Networks: Zeroconf-to-Zeroconf Bridging

4.1 Introduction

Zero Configuration Networking (Zeroconf) [39] solves the following problem: when multiple IP-enabled devices are physically connected with one another, one device should be able to use the services provided by another without requiring the user to configure the devices manually. For example, when a user connects two computers either directly using an Ethernet crossover cable or via an Ethernet switch, he should be able to accomplish his file-transfer task by simply starting up the appropriate applications at both ends. The applications should *discover* each other without the user telling them where to find them.

Today, Zeroconf technology is one of the most widespread solutions for service discovery in local area networks. Bonjour is Apple [3]'s Zeroconf implementation, and it is an integral part of Mac OS X operating system. Bonjour is also installed on a large fraction of the personal computers running Windows operating system, thanks to the popularity of iTunes—Apple's music playing application—which installs Bonjour for Windows as part of its installation process. For UNIX-like platforms, there is a mature open-source implementation of Zeroconf called Avahi [4], which comes preinstalled in a number of major Linux distributions such as Debian [7] and Ubuntu [33]. On the hardware side, virtually every printer sold today supports Zeroconf. The number of Zeroconf-enabled applications is rapidly increasing as well, as evidenced by the growing number of Zeroconf service types registered in [8].

The multicast-based design of Zeroconf, however, effectively limits its usage to the local subnet. This presents no problem for the discovery scenarios that are primarily motivated by hardware devices, such as discovering the printers in a network. But as the focus of Zeroconf is shifting towards more sophisticated services provided by software applications, the limited reach of the services often makes the technology unsuitable for many discovery scenarios that would otherwise be perfect candidates for Zeroconf. For example, a number of chat applications (such as Apple's iChat) use Zeroconf to discover other users in the local link and display them in their *ad hoc* buddies window. A straightforward extension of this mechanism is to discover those people who have convened for the same purpose even if their computers are not in the same local network, such as a group of people attending an academic conference scattered in a number of adjacent buildings, or using a mixture of wired and wireless networks which are usually separate subnets. As another example, iTunes lets a group of officemates share their music. It would be nice to include the coworkers working at home or at a remote satellite location.

In this chapter, we present an approach to extend the reach of Zeroconf service discovery, inspired by the recent innovation in peer-to-peer network research. Structured peer-to-peer overlay networks based on distributed hash tables (DHT) became popular as the substrates on which global-scale distributed systems are built. A DHT network is characterized by an efficient algorithm to map an arbitrary string to a particular node in the network and to produce a routing path of a bounded number of hops from any node to that node. The mapping is deterministic and results in a uniform distribution (or other desired distributions for some algorithms) of the strings among the participating nodes. This enables efficient implementations of a number of global-scale services such as file sharing and overlay multicast. Our approach is to connect multiple Zeroconf subnets using a DHT network. We have designed and implemented the Zeroconf-to-Zeroconf Toolkit (z2z) that connects Zeroconf subnets using OpenDHT, a publicly accessible DHT service. A z2z process running in a subnet *exports* locally available Zeroconf services into OpenDHT. Another z2z process running in a different subnet can then look up the services in OpenDHT, and *import* them into its own local network as if they had originated locally. Such imported services are indistinguishable from the real local services in the eyes of the applications, and thus the imported services simply show up along with other locally available services in the existing, unmodified Zeroconf-enabled applications.

Our contributions are twofold. First, we propose a hybrid architecture that combines the ease of Zeroconf with the scalability of DHT-based peer-to-peer networks. Second, we developed a practical tool that can extend the reach of any existing Zeroconf-enabled application without modification. The modular software design also makes it a suitable framework on which to build a global service discovery system based on Zeroconf.

The remainder of the chapter is organized as follows. Section 4.2 starts with background information on Zeroconf and OpenDHT, and ends with an architecture overview of z2z. Section 4.3 describes the usage of the z2z command line executable, provides a message flow based explanation of how it works, and finally delves into the implementation detail. Section 4.4 lists related work. Lastly, Section 4.5 discusses possible future directions of this effort.

4.2 Background and Approach

4.2.1 Zeroconf, mDNS, DNS-SD, and Bonjour

There is some confusion about what exactly the term Zeroconf means. The term came from the IETF Zero Configuration Networking Working Group [40], which was chartered to develop a requirements specification for networking in the absence of configuration and administration. The working group identified three requirements for zero configuration networks:

1. IP address assignment without a DHCP server;

2. Host name resolution without a DNS server;

3. Local service discovery without any rendezvous server.

For the first requirement, the working group produced the self-assigned link-local addressing standard (RFC 3927) [57], which is implemented in major operating systems today. The working group never reached a consensus regarding the second and third requirements, and it became inactive without producing any further specification.

Meanwhile, Apple introduced *Bonjour*. Bonjour is the implementation of Multicast DNS (mDNS) [59] and DNS-based Service Discovery (DNS-SD) [58] protocols, which are Apple's proposals for the second and third requirements of Zeroconf. As Bonjour became widespread, the term Zeroconf became synonymous with the abstraction that Bonjour implements, namely the mDNS and DNS-SD protocols. Our use of the term Zeroconf is in this spirit.

The self-assigned link-local addressing described in RFC 3927 establishes the foundation for Zeroconf by ensuring that IP networking is functional as long as the link layer is present. This aspect of Zeroconf is not relevant in our discussion of z2z, however, since we assume that the subnets are connected to the Internet.

The second requirement of Zeroconf is satisfied by mDNS. An mDNS daemon is essentially a DNS server. It uses the same DNS record types and the same packet layout. In fact, an application querying for a DNS record would not be able to tell whether a response came from mDNS or a conventional unicast DNS server. There are, however, a few important differences:

- mDNS is run by *every* host in a local link whereas a conventional DNS system runs on a single server host.
- Queries are sent via multicast to all hosts in the local link using UDP port 5353 instead of 53, the conventional port for DNS.
- All mDNS record names must end in ".local.". The resolution of such names are routed to mDNS by the operating system.

A mDNS daemon provides local host name resolution using A type records. For example,

```
Toms-Computer.local.  A  160.39.243.99
```

DNS-SD, together with mDNS, satisfies the third requirement of Zeroconf. DNS-SD defines the naming conventions for PTR, SRV, and TXT records carried by mDNS daemons. PTR records are used to enumerate the service instances of a particular type. The service instances are mapped to the host names and port numbers using SRV records. TXT records accompany the SRV records in order to provide additional information about the service instances. The following example illustrates this concept:

```
_daap._tcp.local.  PTR
    Tom's Music._daap._tcp.local.
_daap._tcp.local.  PTR
    Joe's Music._daap._tcp.local.

Tom's Music._daap._tcp.local.  SRV
    0 0 3689  Toms-Computer.local.

Tom's Music._daap._tcp.local.  TXT
    "Version=196613" "Password=false"
    "Media Kinds Shared=3"

Toms-Computer.local.  A  160.39.243.99
```

This is a textual representation (edited for clarity) of a few DNS records produced by Apple's iTunes music player application when its music sharing option is enabled. The PTR records are used to enumerate the two service *instances* (Tom's Music and Joe's Music) that are currently available in the local network for the "_daap._tcp" service *type*. The host name and port number for a specific service instance (Tom's Music in this case) is provided by a SRV record. A TXT record with the same name as the SRV record carries additional information about the service instance. Finally, an A record maps the local host name to an IP address.

The mDNS daemons running on each host in a local link collectively store and manage the PTR, SRV, TXT, and A records for the services registered in the local subnet. The queries and the answers are then exchanged via link-local multicast.

4.2.2 OpenDHT

OpenDHT is a publicly accessible DHT service [104]. It consists of 200–300 globally distributed hosts running the Bamboo DHT algorithm [30]. Each host also acts as a client gateway exposing a simple *put* and *get* interface. From a client application's point of view, it is simply a remote storage facility where the client application can *put* or *get* key-value pair data items.

The *put* and *get* operations are performed via XML RPC [37]. This black-box approach greatly simplifies application development because the client applications do not need to integrate DHT access libraries. On the flip side, since OpenDHT does not reveal the nodes in the DHT routing path, it is difficult to implement an application that uses such information, such as an overlay multicast built atop a DHT substrate [55].

We chose OpenDHT for the initial implementation of z2z, mainly because of its ease of use. OpenDHT is sufficient for our current use of DHT, which is limited to storing and retrieving service announcements. Other DHT algorithms and implementations can easily be substituted in the future when OpenDHT no longer satisfies our needs.

Any OpenDHT node can act as a gateway to which a client application sends a put or get request, but for the best performance, a gateway node should be chosen so that it is close to the client host in terms of the network topology. For locating the nearest gateway, OpenDHT uses an overlay anycast service called OASIS [68]. Our z2z uses the OASIS mechanism by default, but it also lets the user specify a particular OpenDHT gateway as a command line option.

4.2.3 Architecture Overview of z2z

The basic design of z2z is simple. A z2z process running in a Zeroconf subnet gathers all the service announcements of a particular type (specified by the user) and *exports* them into OpenDHT. Another z2z process running in a different subnet can then *import* those services

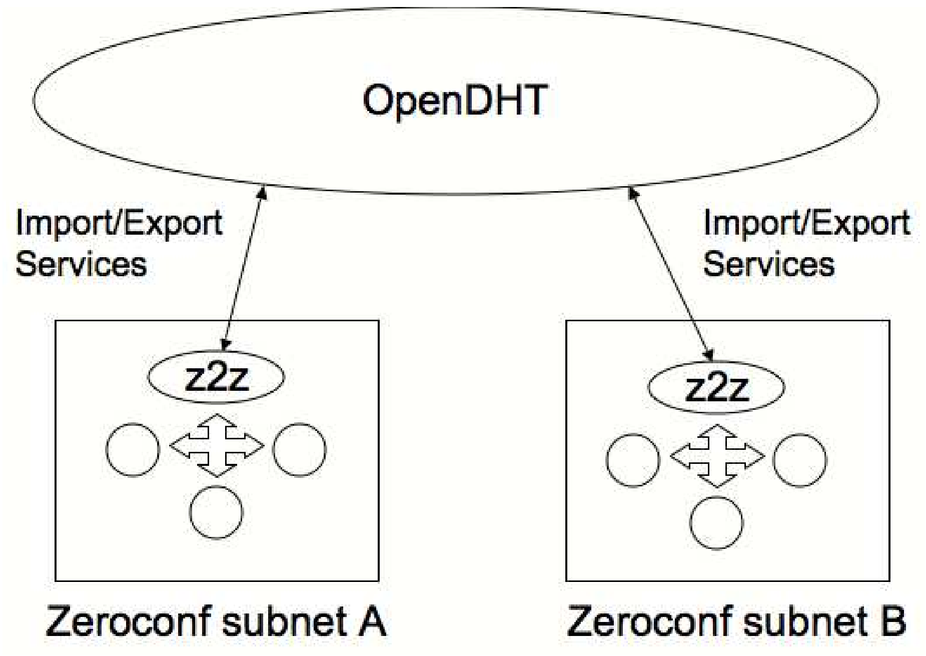


Figure 4.1: Two Zeroconf subnets A and B are exchanging local services with each other. Of course, z2z is not limited to only two subnets. Any number of subnets can export and import services to and from OpenDHT using z2z.

by *getting* those announcements from OpenDHT and register them in its own subnet as if they had originated locally. Figure 4.1 depicts such a scenario. Multiple z2z processes can be present in a single subnet as well. Section 4.5 discusses this case.

Since each data item in OpenDHT is a key-value pair, z2z associates a key with each service item that it exports into OpenDHT. By default, z2z uses the service name as the key (after prepending it with "z2z.opendht." to avoid name collision in OpenDHT). For example, an iTunes music share might be exported by z2z under the key, "z2z.opendht.Tom's Music", where "Tom's Music" is the name under which Tom is sharing his music library in iTunes. Section 4.3.1 explains this in more detail.

4.3 Design and Implementation

The current version of z2z is a command line program written in Java. This section starts with a few examples of command line usage to explain the basics. Then it shows how z2z works under the hood by following the message flows when exporting and importing service items. Finally we discuss some of the issues we encountered in implementing z2z.

4.3.1 Usage Examples

z2z exports local Zeroconf service announcements to OpenDHT, which then can be imported by other z2z processes anywhere in the world. For example:

```
z2z --export:opendht _daap._tcp
```

will export the iTunes music shares found in the local network to OpenDHT. When exporting to OpenDHT, z2z always stores each service using its service name as the key. For example, if one of the music shares exported by the command above is "Joe's Music", Joe's friend in a different network who wants to listen to Joe's music needs to issue the following command:

```
z2z --import:opendht --key "Joe's Music"
```

indicating that he wants to bring in any service stored under the name "Joe's Music". (If Joe's music share was password-protected, the friend should use as key "Joe's Music_PW" because iTunes adds the postfix to the service name of a protected share.) Also, any character that is neither a letter nor a digit will not be used in matching the key, and the comparison is case-insensitive, so the command above is same as:

```
z2z --import:opendht --key "joesmusic"
```

It is also possible to tell the exporter to use additional keys in addition to the service's own service name:

```
z2z --export:opendht _daap._tcp
    --key "music from office network"
```

will make z2z export the local iTunes shares not only under their own service names but also under the string "music from office network". This lets an employee working at home issue the following command to bring in *all* music shares of his office network.

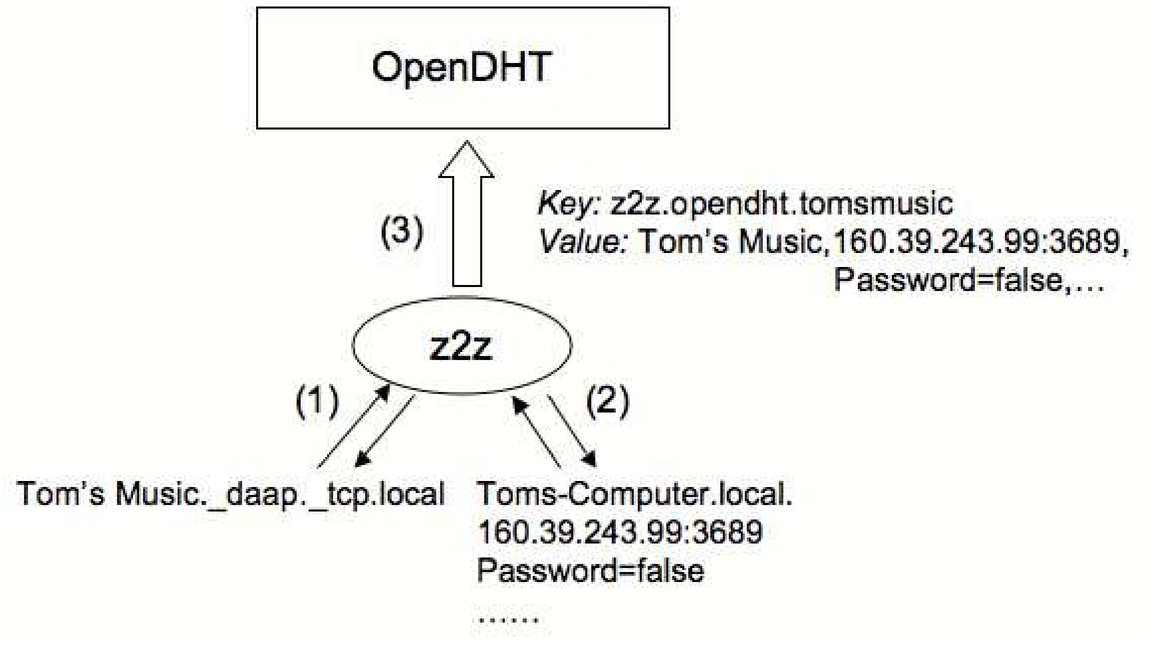

OpenDHT
(3)
Key: z2z.opendht.tomsmusic
Value: Tom's Music,160.39.243.99:3689, Password=false,...
z2z
(1)
(2)
Tom's Music._daap._tcp.local
Toms-Computer.local.
160.39.243.99:3689
Password=false
......

Figure 4.2: Exporting services: (1) z2z discovers a service instance of the type _daap._tcp by issuing a PTR query; (2) The service instance is further resolved to obtain the host name, IP address, and other additional information, using SRV, A, and TXT queries; (3) z2z constructs a key-value pair from the information and sends a put message to OpenDHT.

```
z2z --import:opendht
    --key "music from office network"
```

Multiple keys are also allowed in the command line, in which case z2z will store multiple records in OpenDHT for the same service, one for each specified key.

4.3.2 Message Flow

4.3.2.1 Exporting

Figure 4.2 shows how z2z exports a service announcement to OpenDHT. First, z2z sends out a PTR query via multicast to discover service instances of the type _daap._tcp. In Bonjour parlance, this is called *browsing*, and it is performed by calling a Bonjour API function. Tom's iTunes music share is shown here as the example service instance discovered.

The discovered instance is then *resolved* in order to obtain the details of the service. This is also done by calling a Bonjour API function, which makes SRV and TXT queries to obtain the local host name, port number, and any other additional information about the service stored in the TXT record. (Figure 4.2 has the Password attribute as an example of what is stored in the TXT records.) In a normal Zeroconf service discovery situation, the IP address is not needed since the local host name can identify the host in the local network. However, since the service information that z2z exports to OpenDHT can be used from anywhere on the Internet, the local host name is not sufficient to locate the host. For this reason, z2z resolves the local host name to its IP address and includes it in the service information that it publishes to OpenDHT. Currently z2z does not export the service if the IP address is in the private address space [103]. A future version will address this issue (Section 4.5).

Once z2z obtains all the relevant information about a service instance, it makes a *put* call into OpenDHT in order to store the service item under the specified keys (as explained in Section 4.3.1). Each data item in OpenDHT has a Time-To-Live (TTL) value associated with it. A record is expired in OpenDHT unless it is refreshed with its TTL. Sending the *put* message again refreshes the record. Thus, z2z keeps sending the put request to OpenDHT as long as the service instance is present in the local network. The TTL of the service item and the interval by which z2z resends the put request are by default 5 minutes and 60 seconds, respectively, and they can be changed using the command line parameters.

4.3.2.2 Importing

Figure 4.3 shows another z2z process in another network importing Tom's music share that had been previously exported. First, z2z makes a *get* call to OpenDHT to retrieve the records stored under the key, "tomsmusic". z2z then *registers* the retrieved service into its local network. All the hosts in the network (including the same host on which the z2z process is running) will see the service as if it had originated from the local network, i.e., the iTunes applications running on this network will show "Tom's Music" as one of the shared music libraries in the network. This is accomplished by the Bonjour API functions that inject PTR, SRV, TXT, and A records into the local mDNS daemon.

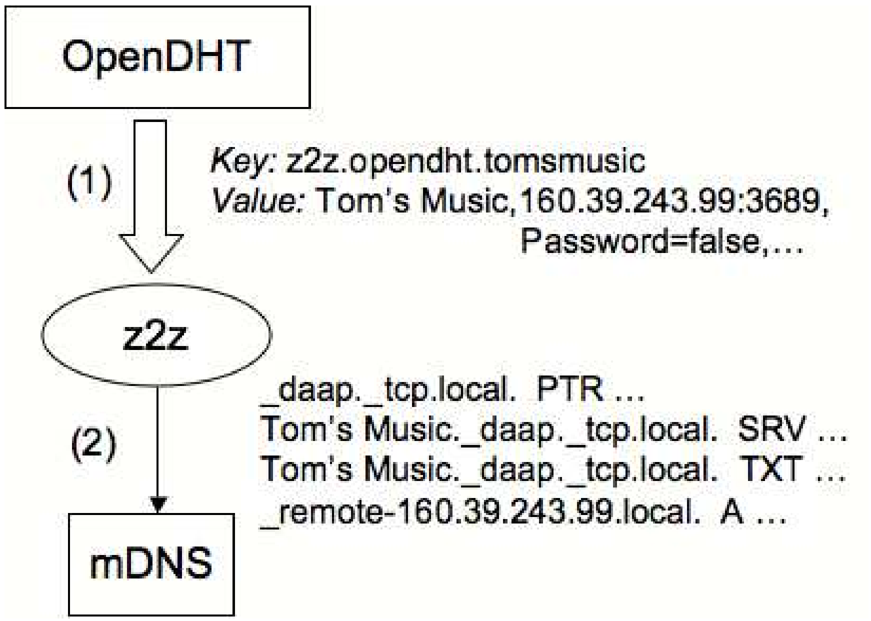

OpenDHT
(1)
Key: z2z.opendht.tomsmusic
Value: Tom's Music,160.39.243.99:3689, Password=false,...
z2z
(2)
_daap._tcp.local. PTR ...
Tom's Music._daap._tcp.local. SRV ...
Tom's Music._daap._tcp.local. TXT ...
_remote-160.39.243.99.local. A ...
mDNS

Figure 4.3: Importing services: (1) z2z retrieves a service item from OpenDHT by sending a get message for the key "tomsmusic". (2) The service item is registered as if it had originated locally. This is done by inserting PTR, SRV, TXT, and A records into the local mDNS daemon.

Note that an A type record for a *fake* host name is added to mDNS. (We use names such as "_remote-160.39.243.99.local.", but any name can be used as long as it ends with ".local." and does not conflict with other host names in the local network.) This record points to the remote IP address of the machine that is actually providing the service. This trick of registering a remote service masquerading as a local one is called *proxy registering* in Bonjour terminology.

It is tricky to manage the lifetime of an imported service because the only way to learn that the service has been expired from OpenDHT is to try to *get* it. The approach taken by z2z is as follows. z2z keeps making *get* calls cycling through the keys specified by the user. There can be multiple keys and for each key there can be multiple service items. For each service item retrieved, it imports it if it is a new service. If an already imported service is retrieved again, it updates its refresh time-stamp. There is a thread that collects stale services (those that have not been refreshed for a while) and removes them from the network. The interval between *get* calls and the stale threshold are by default 10 seconds and 5 minutes, respectively, and they can be changed using the command line parameters.

If there is another z2z process in the local subnet and it is exporting, the imported services will be discovered by that z2z exporter. We need a mechanism to prevent the exporter from exporting the imported service again. A short signature is added as a TXT attribute so that the exporter can distinguish the imported services from the native local services.

4.3.3 Implementation

4.3.3.1 C++ Prototype

The first prototype of z2z was implemented in C++ using the C version of the Bonjour client API. We developed and tested it in Mac OS X first and subsequently ported it to Windows. For OpenDHT access, we used the open-source `xmlrpc-c` library [36]. Using Cygwin environment [6], we were able to build and use the library in Windows as well.

This approach was problematic because the Bonjour client library in Windows uses the Winsock library, which is incompatible with Cygwin's socket-related functions. In particular, Cygwin's `select()` function fails when called with socket descriptors opened by the

Bonjour library. Our workaround was to build two separate executables: one under native Windows environment (Microsoft Visual C++ compiler) and another under Cygwin environment (gcc compiler). The two executables communicated through a socket connection.

4.3.3.2 Open-source Java Implementation

The porting issues of the C++ prototype led us to rewrite z2z from scratch in Java. We used the Java version of the Bonjour client API, and for OpenDHT access, we used Apache XML-RPC [2]. The Zeroconf-to-Zeroconf Toolkit, version 1.0, was released under BSD license and is now available for download from SourceForge.net [41].

It is developed and tested under Mac OS X and Windows. In Windows, it requires Bonjour for Windows available from Apple [5]. (Bonjour for Windows is also automatically installed when iTunes is installed.) The support for Linux or other POSIX-compliant platforms providing Zeroconf through Avahi is planned for a future version.

The Java classes that make up z2z are designed to be modular and have well-defined interfaces. They are intended to serve as the foundation for the future developments that go beyond the current z2z executable. (We outline some of those ideas in Section 4.5.) Here is a brief summary of the main classes:

Exporter browses and resolves the local Bonjour services of a given type, keeping the list of the currently active services. It provides start and stop operations, event notification interface for service additions and removals, and a method to get the current snapshot of the active services.

Importer provides the methods to add or remove remote services into or from the local network.

ExporterToStream and ExporterToOpenDHT use an Exporter to gather the local Bonjour services of a given type, and export them out to the standard output or to OpenDHT, respectively. ExporterToStream uses Exporter's event notification interface to export services as they come and go. ExporterToOpenDHT takes the snapshot of the active services periodically and refreshes them in the OpenDHT storage.

StreamToImporter and OpenDHTToImporter use an Importer to bring in the service announcements it retrieved from the standard input or from OpenDHT, respectively. StreamToImporter adds and removes the services as they come and go through the standard input. OpenDHTToImporter keeps retrieving the specified services periodically from OpenDHT, injecting them into the local network if they are new and refreshing it if they have been imported already. OpenDHTToImporter expires the imported services that have not been refreshed beyond their TTL.

OpenDHTClient provides the access interface to OpenDHT using XML-RPC as the underlying transport. The class is generic and it can be used outside z2z.

ServiceItem encapsulates a single Bonjour service. It provides the methods for serializing and deserializing it into and out of a stream.

4.3.3.3 Implementation Issues

The proxy registering mechanism described in Section 4.3.2.2 is unfortunately not available in the current Java Bonjour client API. The problem is that the current version of Java Bonjour API does not provide a way to inject a type A record into the local mDNS daemon. (The C API does provide this functionality.) As a workaround, z2z currently does a reverse lookup on the IP address and puts in the real, global host name as the value of the SRV record representing the service instance (as opposed to the fake .local name used when proxy registering is available). This eliminates the need of adding a type A record, but it makes it impossible to import services from those IP addresses that are reachable, but do not have global names associated with them. For example, two private address networks might be connected through a router.

A better workaround might be to use the fake .local host names, but instead of injecting a type A record into mDNS, z2z can listen for multicast and answer the A query itself. We will consider implementing this solution in a future version if the proxy registering API continues to be unavailable in the Java Bonjour client library.

The Bonjour API function for registering a service takes as a parameter the network interface index for which the service is registered. Usually it is set to a special value indicating all available interfaces. An interesting value one can pass here is one that indicates that the service should be registered for the local machine only. This option is supposed to register a service in such a way that it is only visible on the machine that registered the service, not any other host in the same local network. This is useful in z2z because it is sometimes undesirable to pollute the network with the imported services that are intended only for a single user. It is an issue especially in a large bridged wireless network where mDNS traffic can have a significant impact on the network performance. (See [21] for an example of such networks.)

Unfortunately, we were not able to incorporate this feature into z2z successfully. Under certain conditions, registering services for local machine only caused internal errors on the mDNS daemon in Mac OS X. Another problem with this option is that certain applications (iTunes being one of them) ignore the services registered in such a way, severely limiting the usefulness of the option.

4.4 Related Work

Apple's solution for Zeroconf beyond local link is Wide-area Bonjour [60]. Wide-area Bonjour replaces the Multicast DNS in Bonjour with the conventional unicast DNS, thereby removing the link-local confinement of Bonjour services. This comes at a cost of setting up and maintaining a real DNS server, which makes Wide-area Bonjour unsuitable for a discovery solution for transient or ad hoc services. Moreover, the client hosts need to know the DNS servers to which they can send queries and publish services. In short, Wide-area Bonjour requires *configuration*.

We believe that z2z is the first attempt at interconnecting Zeroconf subnets using a DHT-based peer-to-peer network. But there have been a number of attempts at making Zeroconf services available beyond the local subnet.

Rendezvous Proxy [24] offers a simple GUI interface for a user to enter the information about a remote Zeroconf service, such as the IP address and port number where the service can be found. It makes the service available locally by performing the proxy registration, the same technique described in Section 4.3.2.2. It is intended as a way to establish a simple point-to-point connection when the user knows the exact nature and location of the service that he wishes to bring into his local network.

LogMeIn Hamachi [17] is a peer-to-peer virtual private network (VPN) solution that provides a virtual LAN connectivity over the Internet. Service discovery is not the main focus of this solution, but Zeroconf is claimed to work in the virtual LAN environment. The fact that it operates on top of a virtual LAN imposes a practical limit on the number of networks it can connect.

Simplify Media [28] applies the idea of social network to iTunes music sharing. Instead of the open peer-to-peer network used by z2z, it uses a private social network to enable iTunes music sharing among friends. Currently Simplify Media is an iTunes-only solution, whereas z2z is a generic solution for all Zeroconf services.

4.5 Discussion

The current implementation of z2z allows multiple z2z processes running in a network to export or import the same set of services. Normally this is not a problem. When a service is exported to OpenDHT by multiple z2z processes, the effect is simply that the service gets refreshed more frequently. When a service is imported into a network by multiple z2z processes, Bonjour recognizes that the DNS resource records being registered are identical, and it treats them as the redundant announcements for a single service. In fact, Apple suggests this as a possible fault-tolerance mechanism [29].

The effect of such redundant registrations on a large local area network, however, needs to be investigated. The multicast traffic from mDNS can have a significant impact on the performance of a large network. This has led some network operators to employ filtering of mDNS traffic [21]. We plan to investigate if, and to what extent, the presence of z2z processes exacerbate the problem. If the redundant registration turns out to be a significant factor, it is straightforward to ensure that only one z2z process is responsible for importing a given remote service.

On the export side, we can eliminate the redundant OpenDHT refreshes by ensuring that only one z2z process is exporting a service type under a given key. This can be implemented using Bonjour. A z2z process can advertise a name constructed from the service type and the key that it intends to export, and then use Bonjour's built-in name conflict resolution mechanism to see if another z2z process is already exporting the type under the same key. It is unclear, however, that the reduction of OpenDHT calls outweighs the additional multicast traffic.

Privacy is another important consideration when z2z is used in a large network, especially when there are a large number of users, such as in a University campus network. When a user publishes a service using a Bonjour-enabled application (when a user shares his music library in iTunes for example), he expects his service to be available in the local network, but he may not be aware that the service can be carried outside the local network by z2z. Therefore we emphasize that z2z should be used in a way that respects the privacy of the users in the local network. It should be noted, however, that z2z does not introduce any new technology that facilitates the invasion of privacy. One can easily browse and resolve the local services using many other readily available tools, and then post the information on a web page, for example.

Part II

Enabling In-Network Services

Chapter 5

NetServ: Activating the Network Edge

5.1 Motivation

There are two types of Internet Service Providers (ISPs): content and eyeball [64]. Content ISPs provide hosting and connectivity for content publishers, and eyeball ISPs provide last-mile connectivity to a large number of end users. It has been noted that eyeball ISPs wield increased bargaining power in peering agreements because they *own* the eyeballs [64]. Eyeball ISPs have another unique asset, edge routers, which they are currently under-utilizing.

Content publishers[1] are motivated to operate at the network edge, close to end users, as evidenced by the success of Content Distribution Network (CDN) operators like Akamai [1] and Limelight [15]. The edge routers of eyeball ISPs, due to their proximity to end users, occupy an excellent location to host content and services. Placing content and services on edge routers would provide an alternate hosting platform for publishers, and a new revenue opportunity for eyeball ISPs (which we simply refer to as ISPs for the remainder of this

[1] We use the term content publishers in a broad sense, referring not only to CNN and YouTube who provide content, but also to Amazon and Skype who provide services like e-commerce and telephony. "Content, application, and service provider", sometimes referred to as CASP, might be a more descriptive term, but we use "publishers" when we need to clearly distinguish them from Internet Service Providers.

thesis).

Programmable routers [62,80,86,89], traditionally software routers based on commodity operating systems or more recently commercial routers with an SDK [87], have been used to implement new network functions. Many of the following functions have become ubiquitous: QoS, firewall, VPN, IPsec, NAT, web cache, rate limiting, and enhanced congestion control algorithms. This model, however, is inadequate for hosting publishers' custom functionality in edge routers. If a publisher wishes to deploy a specifically tailored function, it must go through a very slow, highly coordinated development cycle involving the developers at the publisher, the network administrators at the ISP, and in some cases even the router vendor.[2] This presents a barrier to many publishers, particularly if they want to deploy functions on edge routers across different ISPs. The deployment process is equally cumbersome. Deployment has been a secondary concern for previous programmable router platforms. Thus, adding functionality to a router usually means an administrator installing and configuring a software module. This may be acceptable for a limited set of functions that are largely static, but it is clearly inadequate if a publisher wants to dynamically reconfigure a function quickly and frequently.

5.2 Design Goals

We propose NetServ, a programmable node architecture with the primary focus of facilitating the interaction between ISPs and content publishers. From a technical standpoint, NetServ is similar to existing programmable router proposals. We start with a general purpose open-source operating system as the forwarding engine, and layer a dynamic module system on top of it so that new functions can be added and removed. However, the design decisions we have made reflect a significant rethinking of the role that we envision an edge router will play in the future. An edge router is recast as a *hosting* platform for publishers' content and services. The primary users of NetServ routers are not the network operators of the ISPs that own them, but the content publishers who deploy their services on them.

The shift of focus led us to the following goals in our design:

[2]Developing a Juniper SDK application requires a partnership agreement, for instance.

Wide-area deployment A content publisher should be able to deploy its functions at any edge router on the Internet, subject to policy restrictions. The publisher may not even know the precise target, as is the case when it wants to deploy a web cache *near* a certain group of end users, for example.

Multi-user execution environment The node architecture must support concurrent executions of functions from multiple publishers. Each publisher's execution environment must be isolated from one another and the resource usage of each must be controlled.

Economic incentive The current dynamic between content publishers and ISPs is clearly driven by economic concerns. Our proposal must provide clear economic incentives. Specifically, we must find compelling use cases that demonstrate economic benefits to both.

Unified runtime environment Content publishers' applications running on ISPs' routers will combine the traits of both traditional end-to-end network services and in-network router functions. Our node architecture must support both traditional server application and in-network packet processing application.

5.3 Node Architecture

Figure 5.1 depicts the architecture of a NetServ node. The service modules, represented as ovals, run in a virtual execution environment. The virtual execution environment provides a basic API as a building block layer, consisting of preloaded modules.

We took heed of Calvert's reflection on active networking in 2006 [54]. He noted that "late binding"–i.e., leaving things unspecified–did not help the case. We picked the JVM as the execution environment for service modules to achieve service mobility and a platform-independent programming interface. Java is the natural choice today. No other technology matches its maturity, features, track record of large-scale deployments, extensive libraries and wide-spread use among developers.

The execution environments communicate with the packet transport layer. The packet

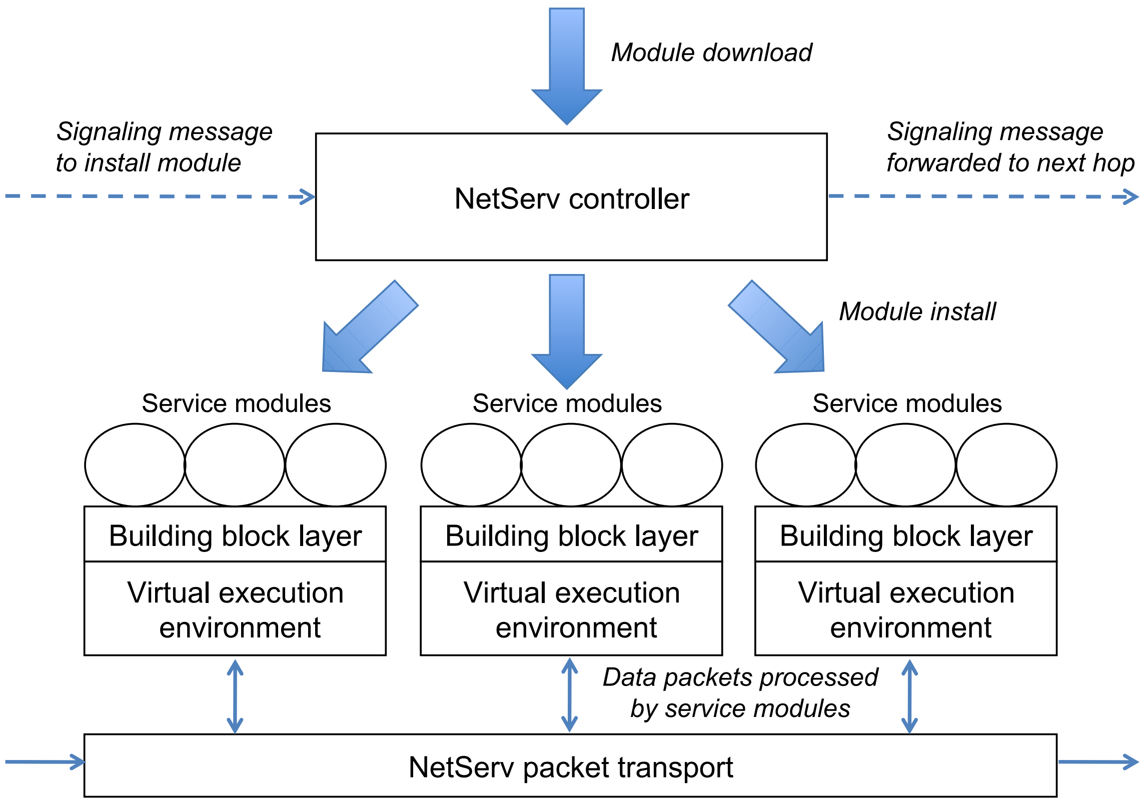

Module download
Signaling message to install module
NetServ controller
Signaling message forwarded to next hop
Module install
Service modules
Service modules
Service modules
Building block layer
Building block layer
Building block layer
Virtual execution environment
Virtual execution environment
Virtual execution environment
Data packets processed by service modules
NetServ packet transport

Figure 5.1: Overview of NetServ node architecture.

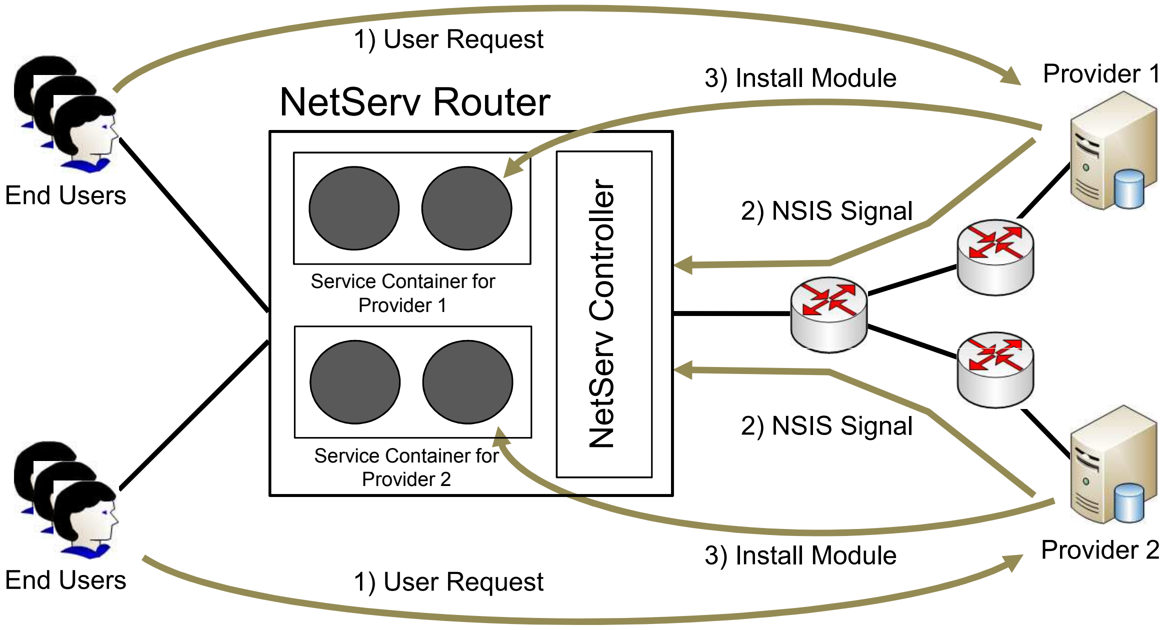


Figure 5.2: Deploying modules on a NetServ-enabled edge router.

transport layer provides the TCP/IP stack for server application modules. For packet processing application modules, the packet transport layer provides a mechanism to filter IP packets and route them to appropriate modules.

The NetServ controller downloads and installs an application module when it receives a signaling message. A user sends a signaling message towards a destination of his interest. Every NetServ node on-path intercepts the message, takes an appropriate action, and forwards it to the next hop.

5.4 End-to-end Service Scenario

Figure 5.2 places a NetServ node in a broader context of an end-to-end service. (1) End user requests are received by a content provider's server, triggering signaling from the server. (2) As a signaling message travels towards an end user, it passes through a mixture of regular IP routers and NetServ-enabled routers between the content provider and the user. Regular IP routers simply forward the message towards the destination. (3) When the message passes through a NetServ router, however, it causes the NetServ router to download and install an application module from the content provider. The exact condition to trigger signaling and what the module does once installed will depend on the application. For example, a content provider might send a signal to install a web caching module when it detects web requests above a predefined threshold. The module can then act as a transparent web proxy for downstream users.

Chapter 6

NetServ Node Implementation

6.1 NetServ on Linux Netfilter Transport

We implemented the NetServ architecture on Linux. We have released source code[1] in conjunction with NetServ tutorials we have held at the 11th, 12th, and 13th GENI Engineering Conferences (GEC11, GEC12, GEC13). We will continue to release new versions of our software and give NetServ tutorials at future GECs.

Figure 6.1 describes our Linux implementation. The arrow at the bottom labeled "signaling packets" indicates the path a signaling packet takes. The packet is intercepted by the signaling daemons, which unpack the signaling packet and pass the contained message to the NetServ controller. The controller acts on the message by issuing commands to the appropriate service containers, to install or remove a module, for example.

Service containers are user space processes with embedded JVMs. Each container holds one or more application modules created by a single user. The JVMs run the OSGi module framework [22]. Thus, the application modules installed in service containers are OSGi-compliant JAR files known as *bundles*. The OSGi framework allows bundles to be loaded and unloaded while the JVM is running. This enables a NetServ container to install and remove application modules at runtime. There are a number of implementations of the OSGi framework; we use Eclipse Equinox [9].

[1] http://www.cs.columbia.edu/irt/project/netserv

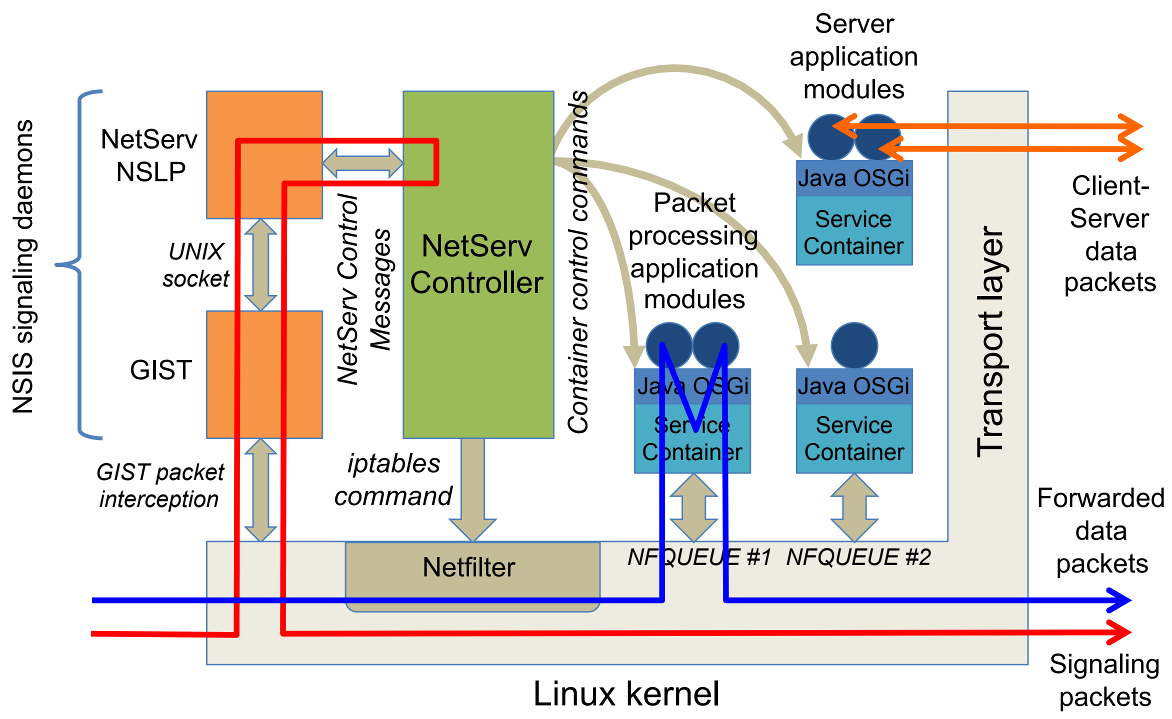


Figure 6.1: NetServ node implementation using Linux Netfilter packet processing framework and Java OSGi module system.

There are two types of application modules shown in Figure 6.1. *Server application modules*, shown as two circles on the upper-right service container, act as standard network servers, communicating with the outside world through the Linux TCP/IP stack. *Packet processing application modules*, shown as two circles on the lower-left container, are placed in the packet path of the router. The arrow labeled "forwarded data packets" shows how an incoming packet is routed from the kernel to a service container process running in user space. The packet then visits two modules in turn before being pushed back to the kernel.

The distinction between server module and packet processing module is a logical one. A single application module can be both. This is an important feature of a NetServ node: it eliminates the traditional distinction between a router and a server. The applications deployed by content publishers typically include both functionalities.

6.1.1 Signaling

We use on-path signaling as the deployment mechanism. Signaling messages carry commands to install and remove modules, and to retrieve information–like router IP address and capabilities–about NetServ routers on-path. We use the Next Steps in Signaling (NSIS) protocol suite [76], an IETF standard for signaling. NSIS consists of two layers, a generic *signaling transport* layer and an application-specific *signaling application* layer:

> "a 'signaling transport' layer, responsible for moving signaling messages around, which should be independent of any particular signaling application; and
>
> a 'signaling application' layer, which contains functionality such as message formats and sequences, specific to a particular signaling application." (RFC 4080 [76])

The two boxes in Figure 6.1, labeled "GIST" and "NetServ NSLP," represent the two NSIS signaling layers used in a NetServ node. GIST, the General Internet Signalling Transport protocol [110], is an implementation of the transport layer of NSIS. GIST is a soft state protocol that discovers other GIST nodes and maintains associations with them in the background, transparently providing this service to the upper signaling application layer.

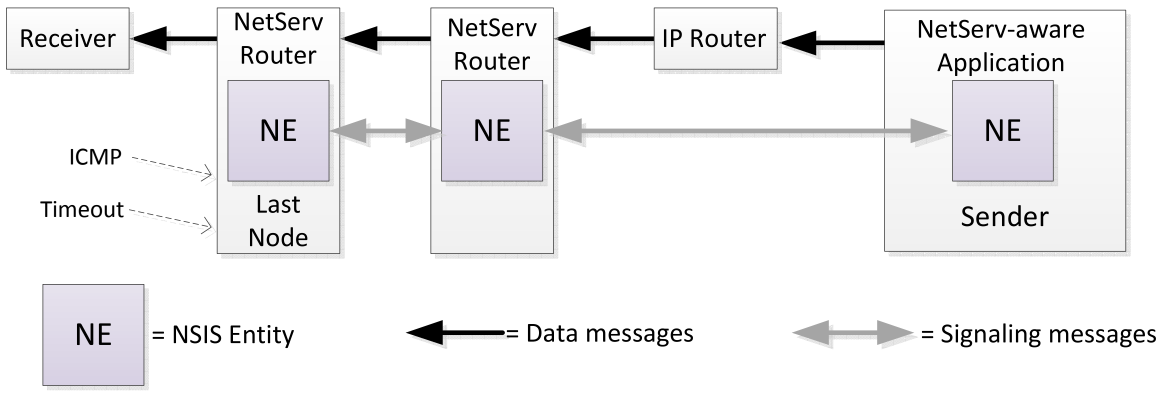


Figure 6.2: NetServ signaling flow.

NetServ NSLP is the NetServ-specific application layer of NSIS. It contains the signaling logic of NetServ and relays messages to the NetServ controller. The current implementation of the NetServ signaling daemons is based on NSIS-ka [20], whereas the previous version of our prototype used FreeNSIS [11].

GIST peer discovery depends on the ability to intercept certain UDP packets. GIST's standard method of intercepting packets is through the use of the IP Router Alert Option (RAO) [85]. However, the RAO is not well-defined in IPv4 networks and different devices tend to behave incongruously. As an alternative, packet filtering can be used to intercept packets destined for port 270, the port assigned by IANA for GIST. NSIS-ka uses this method. Specifically, it uses the Netfilter packet filtering system in Linux.

Figure 6.2 shows a possible NetServ signaling scenario. A signaling message is sent from an application, through several routers, to the receiver. The receiver and the generic IP Router are unaware of NSIS signaling. Thus, the IP router performs only IP layer forwarding. The sender and the two NetServ routers are NSIS enabled; once GIST associations between the nodes are set up, NSIS signaling messages can flow in both directions.

Content providers want to place content and services as close to end users as possible. Therefore, while setting up GIST associations, discovering the last NetServ node on-path becomes especially important. The GIST layer determines that its host is the last NSIS node on-path when it fails to discover a peer further along the path. It retransmits discovery packets with exponential back-off up to a predefined threshold. Depending on the threshold

```
SETUP NetServ.apps.NetMonitor_1.0.0 NETSERV/0.1
dependencies:
filter-port:5060
filter-proto:udp
notification:
properties:visualizer_ip=1.2.3.4,visualizer_port=5678
ttl:3600
user:janedoe
url:http://content-provider.com/modules/netmonitor.jar
signature:4Z+HvDEm2WhHJrg9UKovwmbsA71FTMFykVa0Y\xGclG8o=
<blank line>
```

Figure 6.3: An example of NetServ `SETUP` message.

this can take a long time. To shorten last node discovery time, we modified NSIS-ka to detect an ICMP *port unreachable* message. Although this is not always reliable, it shortens the discovery in many cases.

There are two kinds of NetServ signaling messages: requests and responses. Typically, a content provider's server sends a request toward an end user. The last on-path NetServ node generates a response to the server.

There are three types of NetServ requests: `SETUP`, `REMOVE`, and `PROBE`. The `SETUP` message is used to install a module on the NetServ nodes on-path. The `REMOVE` message uninstalls it. The `PROBE` message is used to obtain the NetServ nodes' statuses, capabilities, and policies. Figure 6.3 shows an example of a `SETUP` message. It requests that an application module called NetMonitor be downloaded from the given URL, installed in the packet path to process UDP packets for port 5060, and automatically removed after 3600 seconds. `REMOVE` and `PROBE` messages are similar. Table 6.1 describes available header fields used in request messages and in which type of request they may appear.

Headers	Where	Description
dependencies†	S	Lists the modules necessary to run the application module being installed
filter-port	S	Destination port of packets that should be intercepted & delivered to the module
filter-proto	S	Protocol of packets that should be intercepted & delivered to the module
notification†	SR	XML-RPC URL that should be called after the module has been successfully installed
node-id	SRP	Identifies a specific NetServ node
probe	P*	Identifies the information being probed
properties	S	Additional parameters for the module being installed
ttl	S*	The number of seconds after which the module is automatically uninstalled
signature	S*R*P*	The signature of the message authenticating the request
user	S*R*P	The owner of the NetServ service container

†Not fully implemented in the current version.

Table 6.1: List of available headers in NetServ requests. (S: `SETUP`, R: `REMOVE`, P: `PROBE`, *: mandatory)

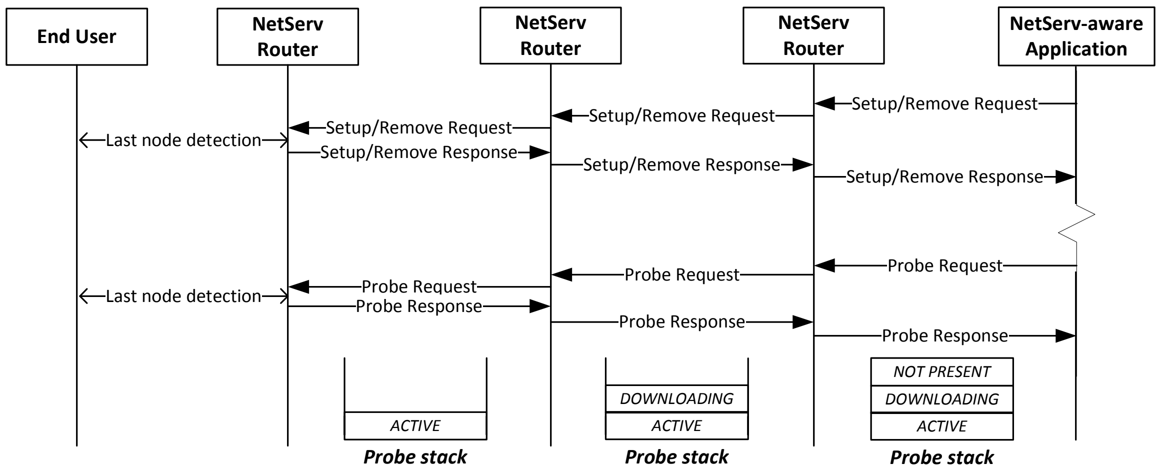


Figure 6.4: Request and response exchange.

Figure 6.4 shows how response messages are generated at the last node and returned along the signaling path back to the requester. The responses to `SETUP` and `REMOVE` requests simply acknowledge the receipt of the messages. A response to a `PROBE` request carries the probed information in the response message. As the message transits NetServ nodes along the return path, each node adds its own information to the response stack in the message. The full response stack is then delivered back to the requester. Figure 6.4 shows a response to a module status probe being collected in a response stack.

6.1.2 NetServ Controller

The NetServ controller coordinates three components within a NetServ node: NSIS daemons, service containers, and the forwarding plane. It receives control commands from the NSIS daemons, which may trigger the installation or removal of application modules within service containers, and in some cases filtering rules in the forwarding plane.

The controller is responsible for setting up and tearing down service containers. The current prototype pre-forks a fixed number of containers. Each container is associated with a specific user account. The controller maintains a persistent TCP connection to each container, through which it tells the container to install or remove application modules.

It uses an XML configuration file which specifies user name, public key, container IP address, ports authorized for listening, destination IP prefixes authorized for filtering, and the sandbox directory of the container.

6.1.3 Forwarding Plane

The forwarding plane is the packet transport layer in a NetServ node, which is typically an OS kernel in an end host or forwarding plane in a router. The architecture requires only certain minimal abstractions from the forwarding plane. Packet processing modules require a hook in user space and a method to filter and direct packets to the appropriate hook. Server modules require a TCP/IP stack, or its future Internet equivalent. The forwarding plane must also provide a method to intercept signaling messages and pass them to the GIST daemon in user space.

Currently we use Netfilter, the packet filtering framework in the Linux kernel, as the packet processing hook. When the controller receives a `SETUP` message containing `filter-*` headers, it verifies that the destination is within the allowed range specified in the configuration file. It then invokes an `iptables` command to install a filtering rule to deliver matching packets to the appropriate user space service container using Netfilter queues. The user space service container retrieves the packets from the queue using `libnetfilter_queue`.

The Linux TCP/IP stack allows server modules to listen on a port. The allowable ports are specified in the configuration file.

NetServ can use forwarding planes other than the Linux kernel. We have prototyped alternate forwarding planes for NetServ using the Click router [89] and the OpenFlow [99] switch. We are also currently porting NetServ to Juniper routers using the JUNOS SDK [87].

Click is a modular software router platform where a directed graph of *elements* represents the packet path. We implemented a packet processing hook using the IPClassifier, FromUserDevice, and ToUserDevice elements. A user space container retrieves packets from `/dev/fromclick`N which is backed by the ToUserDevice element, and similarly sends packets to `/dev/toclick`N which is backed by the FromUserDevice element. The IPClassifier element provides filtering functionality and can be controlled using a `proc`-like pseudo-file system.

Juniper provides the JUNOS SDK, an API for developing third-party plug-ins. JUNOS

SDK has two parts. *RE SDK* is intended for developing daemons running on the control plane of a Juniper router, called the Routing Engine. The Routing Engine can host the NSIS daemons and the NetServ controller. *Services SDK* provides APIs for developing packet processing applications running on a hardware board attached to the forwarding plane through a 10 Gb/s internal link. The board contains a multi-core network processor to perform packet processing at line rate. We plan to explore the possibility of running NetServ container on the board.

OpenFlow is a programmable switch architecture which exposes its flow table through a standard network protocol called the OpenFlow Protocol. OpenFlow provides an interesting possibility for NetServ: a physically separate forwarding plane. When a NetServ node is connected to an OpenFlow switch via a local 10 Gb/s link, the NetServ node acts as an outboard packet processing engine, which is dynamically configurable. In addition, the NetServ controller can control the OpenFlow switch using the OpenFlow Protocol. This *sidecar* approach has the performance advantage over the single-box approach, since multiple NetServ nodes can be attached to a forwarding plane.

6.1.4 Service Container and Modules

Service containers are user space processes that run modules written in Java. Figure 6.5 shows our current implementation. The service container process can optionally be run within lxc [18], an OS-level virtualization technology included in the Linux kernel.

When the container process starts, the container creates a JVM, and calls an entry point Java function that launches the OSGi framework.

The service container starts with a number of preinstalled modules which provide essential services to the application modules. We refer to the collection of preinstalled modules as the building block layer. The building block layer typically includes system modules, library modules, and wrappers for native functions. System modules provide essential system-level services like packet dispatching. Library modules are commonly used libraries like Servlet engine or XML-RPC. The building block layer can also provide wrappers for native code when no pure Java alternative is available. For example, our ActiveCDN application described in Section 7.1 requires Xuggler [38], a Java wrapper for the FFmpeg [10] video

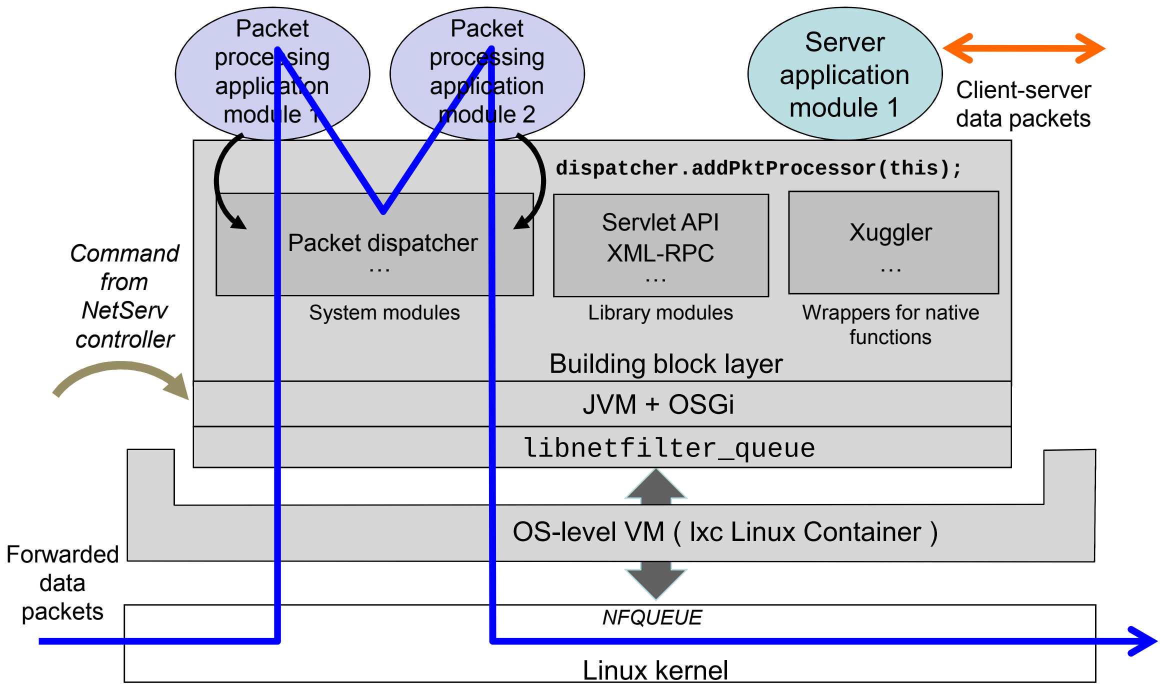

Packet processing application module 1
Packet processing application module 2
Server application module 1
Client-server data packets
dispatcher.addPktProcessor(this);
Packet dispatcher
…
Servlet API
XML-RPC
…
Xuggler
…
System modules
Library modules
Wrappers for native functions
Building block layer
Command from NetServ controller
JVM + OSGi
libnetfilter_queue
OS-level VM (lxc Linux Container)
Forwarded data packets
NFQUEUE
Linux kernel

Figure 6.5: User space service container process.

processing library.

The set of modules that make up the building block layer is determined by the node operator. An application module with a specific set of dependencies can discover the presence of the required modules on path using `PROBE` signaling messages, and then include a dependency header in the `SETUP` message to ensure the application is only installed where the modules are available. We plan to develop a recommendation for the composition of the building block layer.

Server application modules, depicted as the rightmost application module in Figure 6.5 use the TCP/IP stack and the building block layer to provide network services. An OSGi bundle is event driven. The framework calls `start()` and `stop()` methods of the `Activator` class of the bundle. Server modules typically spawn a thread in the start method. *Packet processing* application modules, the two application modules on the left, implement the `PktProcessor` interface, and register themselves with the packet dispatcher in order to receive transiting data packets.

The container process uses `libnetfilter_queue` to retrieve a packet, which is then passed to the packet dispatcher, a Java module running inside the OSGi framework. The packet dispatcher then passes the packet to each packet processing application module in turn. This path is depicted by the arrow labeled *forwarded data packets* in Figure 6.5. We avoid copying a packet when it is passed from C code to Java code. We construct a direct byte buffer object that points to the memory address containing the packet. The reference to this object is then passed to the Java code.

6.2 Security

We consider security risks that arise from the fact that multiple service containers belonging to different users coexist in a NetServ node.

***Resource control and Isolation*:** A single user should not be allowed to consume more than his fair share of the system resources such as CPU, memory, disk space or network bandwidth. Furthermore, a user's execution environment must be isolated from the others', in order to prevent intentional or accidental tampering.

***Authentication and Authorization*:** A user's request to install or remove a module must be verified to ensure that it is from a valid user. Installed modules are subject to further restrictions. In particular, a packet processing module must not be allowed to inspect or modify packets belonging to other users.

6.2.1 Resource Control and Isolation

We have multiple layers of resource control and isolation in the service container. First, because the container is a user space process, we can use the standard Linux resource control and isolation mechanisms, such as nice value, `setrlimit()`, disk quota, and `chroot`.

We control the application modules further using Java 2 Security [73]. It provides fine-grained controls on file system and network access. We use them to confine the modules' file system access to a directory, and limit the ports on which the modules can listen. Java 2 Security also allows us to prevent the modules from loading native libraries and executing external commands.

In addition, the container can optionally be placed within lxc[2], the operating system-level virtualization technology in Linux. Lxc provides further resource control beyond that which is available with standard operating system mechanisms. We can limit the percentage of CPU cycles available to the container relative to other processes in the host system. Lxc provides resource isolation using separate namespaces for system resources. The network namespace is particularly useful for NetServ containers. A container running in lxc can be assigned its own network device and IP address. This allows, for example, two application modules running in separate containers to listen on "*:80" without conflict. At at the time of this writing, a service container running inside lxc does not support packet processing modules.

OSGi provides namespace isolation between bundles using a custom class loader. The only method of inter-bundle communication is for a bundle to explicitly *export a service* by listing a package containing the interfaces in the manifest file of its JAR file, and for another bundle to explicitly *import* the service, also by using a manifest file. However, this

[2] lxc is also referred to as "Linux containers" which should not be confused with NetServ service containers. References to containers throughout this paper mean NetServ service containers.

isolation mechanism is of limited use to us because a container contains modules from a single publisher.

NetServ modules also benefit from Java's language level security. For example, the memory buffer containing a packet is wrapped with a `DirectByteBuffer` object and passed to a module. The `DirectByteBuffer` is backed by memory allocated in C. However, it is not possible to corrupt the memory by going out-of-bounds since such access is not possible in Java.

6.2.2 Authentication and Authorization

`SETUP` request messages are authenticated using the signature header included in each message. Currently, the NetServ node is preconfigured with the public key of each user. When a user sends a `SETUP` message, it signs the message with a private key, this signature is verified by the controller prior to module installation. The current prototype signs only the signaling message–which includes the URL of the module to be downloaded. The next prototype will implement signing of the module itself. As future work, we plan to develop a third party authentication scheme which will eliminate the need to preconfigure a user's public key. A clearinghouse will manage user credentials and settle payments between content providers and ISPs.

Authorization is required if the `SETUP` message for an application module includes a request to install a packet filter in the forwarding plane. If the module wants to filter packets destined for a specific IP address, it must be proved that the module has a right to do so. The current prototype preconfigures the node with a list of IP prefixes that the user is authorized to filter.

Our requirement to verify the ownership of a network prefix is similar to the problem being solved in the IETF Secure Inter-Domain Routing working group [27]. The working group proposes a solution based on Public Key Infrastructure (PKI), called *RPKI*. RPKI can be used to verify whether a certain autonomous system is allowed to advertise a given network prefix. We plan on using that infrastructure once it becomes widely available.

We also plan to support a less secure, but simpler verification mechanism that does not rely on PKI. It is based on a reverse routability check. To prove the ownership of an IP address, the user generates a one-time password and stores the password on the server with that IP address. The password is then sent in the SETUP message. Before installing the module, the NetServ controller connects to the server at the IP address, and compares the password included in the SETUP message with the one stored on the server. A match shows that the user of the module has access to the server. The NetServ node accepts this as proof of IP ownership.

Security checks used by a NetServ router are a matter of local configuration policy and will be determined by the administrator of the router.

6.3 Performance Evaluation of NetServ on Linux Netfilter

For NetServ signaling, we refer to previous work on the performance measurement of the NSIS signaling suite. Fu *et al.* [69], analyzed an implementation of the GIST protocol. Using a minimal hardware setup they measured the maximum concurrent signaling sessions, finding that the GIST node was able to maintain over 50,000 concurrent sessions. The main focus of our measurement results is not signaling but rather packet processing.

Regarding NSLP layer, there are previous performance measurements in [113] and [47]. From these papers, it is clear that the signaling architecture scales well with the complexity of the NSLPs installed and that most of the performance penalties reside in the logic complexity implemented in each NSLP. NetServ NSLP has been designed to keep at a minimum the logic complexity and we believe it will certainly be suitable in handling the amount of signaling requests needed to be processed in an average NetServ node.

We provide evaluation results for our Linux implementation. In particular, we measure the overhead introduced by placing packet processing modules in user space JVM. First, we measure the Maximum Loss Free Forwarding Rate (MLFFR) of a NetServ node with a single service container. We show the overhead associated with each layer in a NetServ node. Second, we perform a microbenchmark measurement to show the delay in each layer. Lastly, we run 100 service containers in a NetServ node and measure the throughput and memory consumption. Our results suggest that while there is certainly significant overhead, it is not prohibitive.

6.3.1 Setup

Our setup consists of three nodes connected in sequence: sender, router, and receiver. The sender generates UDP packets addressed to the receiver and sends them to the router, which forwards them to the receiver.

All three machines were equipped with a 3.0 GHz Intel Dual Core Xeon CPU, 4 x 4 GB DDR2 RAM, and an Intel Pro/1000 Quad Port Gigabit Ethernet adapter connected on PCIe x 4 bus which provided 8 Gb/s maximum bandwidth. All links ran at 1 Gb/s. We disabled Ethernet flow control which allowed us to saturate the connection.

For the sender and receiver, we used a kernel mode Click router version 1.7.9 running on a patched 2.6.24.7 Linux kernel. The Ethernet driver was Intel's igb version 1.2.44.3 with Click's polling patch applied. For the router, we used Ubuntu Linux 10.04 LTS Server Edition 64bit version, with kernel version 2.6.32-27-server, and the igb Ethernet driver upgraded to 2.4.12 which supports the New API (NAPI) [108] in the Linux kernel.

6.3.2 Results

First, we measured the sender and receiver's capacity by connecting them directly. The sender was able to generate 64 B packets and send them to the receiver at the rate of 1,400 kpps, which was well beyond the measured MLFFRs of each of our tests.

After verifying that the testbed had sufficient capacity, We measured the MLFFRs of six different configurations of the router. Figure 6.6 shows the different configurations of the router that were tested. Each configuration adds a layer to the previous one, adding more system components through which a packet must travel.

Configuration 1 is the plain Linux router we described above. This represents the maximum attainable rate of our hardware using a Linux kernel as a router.

Configuration 2 adds Netfilter packet filtering kernel modules to configuration 1. This represents a more realistic router setting than configuration 1 since a typical router is likely to have a packet filtering capability. This is the base line that we compare with the rest of the configurations that run NetServ.

Configuration 3 adds the NetServ container, but with its Java layer removed. The packet path includes the kernel mode to user mode switch, but does not include a Java execution

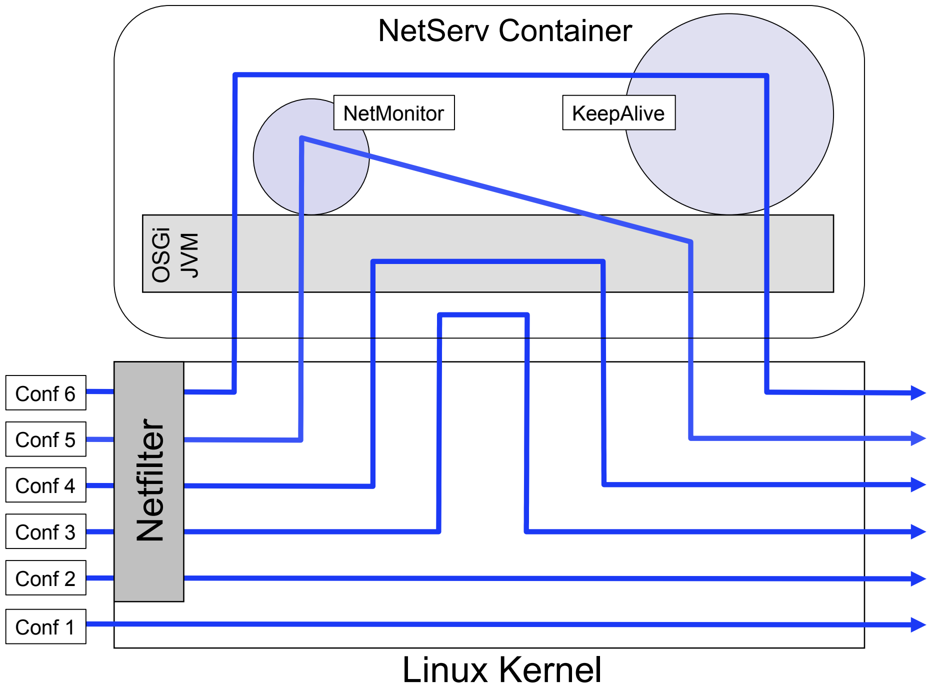


Figure 6.6: Test configurations 1 to 6.

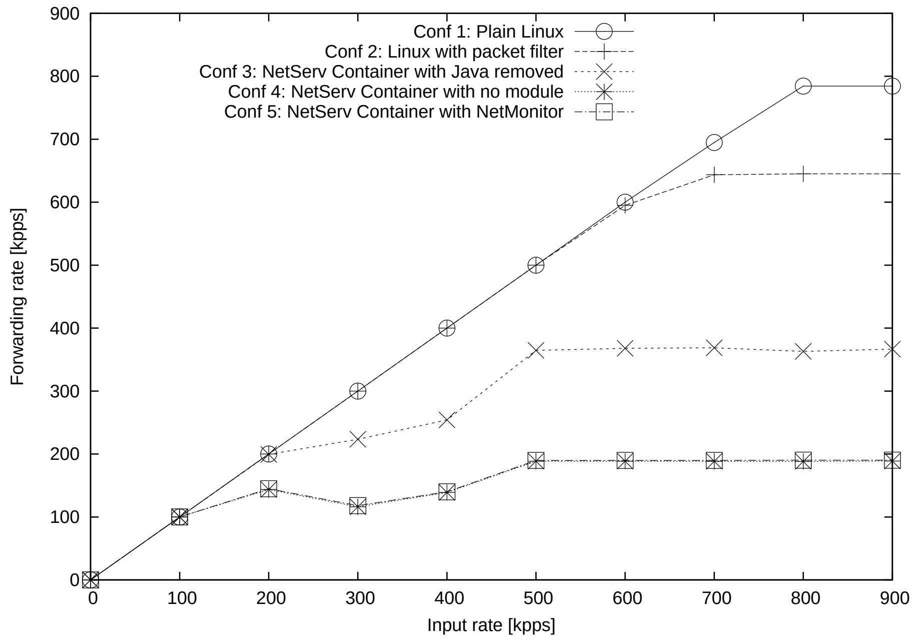
environment.

The packet path for configuration 4 includes the full NetServ container, which includes a Java execution environment. However, no application module is added to the NetServ container.

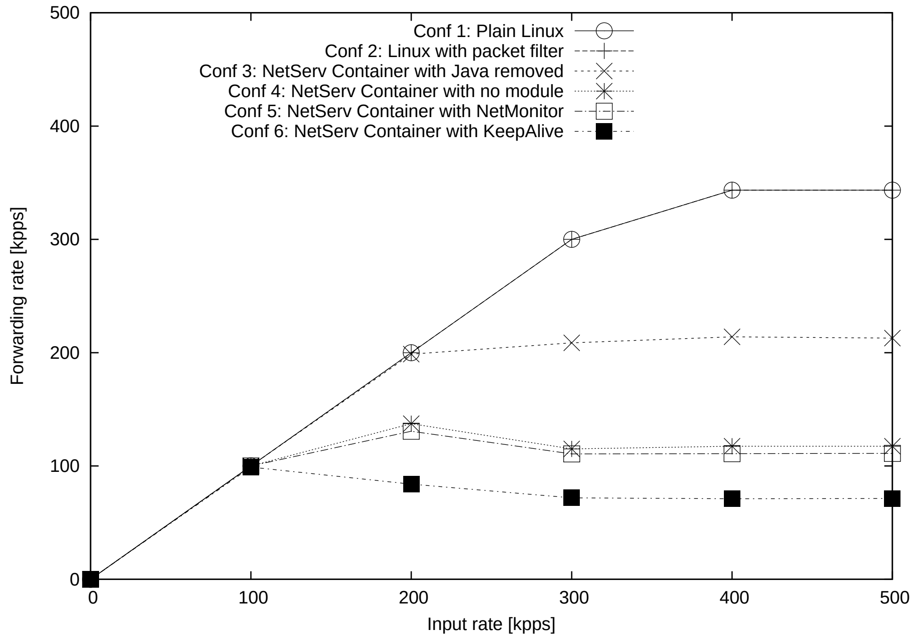
Configuration 5 adds NetMonitor, a simple NetServ application module with minimal functionality. It maintains a count of received packets keyed by a 3-tuple: source IP address, destination IP address, and TTL.

Configuration 6 replaces NetMonitor with the KeepAlive module described in Section 7.2. KeepAlive examines incoming packets for SIP `NOTIFY` requests with the keep-alive `Event` header and swaps the source and destination IP addresses. For the measurement, we disabled the address swapping so that packets can be forward to the receiver. This test represents a NetServ router running a real-world application module.

Figure 6.7(a) shows the MLFFRs of five different router configurations. The MLFFR of configuration 1 was 786 kpps, configuration 2 was 641 kpps, configuration 3 was 365 kpps, configuration 4 was 188 kpps, and configuration 5 was 188 kpps.



(a) Configuration 1 to 5 with 64 B packets.



(b) Configuration 1 to 6 with 340 B packets.

Figure 6.7: Forwarding rates of the router with different configurations.

The large performance drop between configurations 2 and 3 can be explained by the overhead added by a kernel-user transition. The difference between configurations 3 and 4 shows the overhead of Java execution. There is almost no difference between configurations 4 and 5 because the overhead of the NetMonitor module is negligible.

In configurations 3 through 5, we observed that there were some dips in the forwarding performance before it reached the peak rate. For example, in configuration 3, the forwarding rate of the router was 250 kpps when the input rate was between 200 kpps and 500 kpps, but it increased to 364 kpps at 500 kpps. This increase can be explained as a result of switching between the interrupt and polling modes of the NAPI network driver. Under heavy load, the network driver switched to polling mode. Thus, the NetServ process could use more CPU cycles without hardware interrupts. We verified this by comparing the number of interrupts per interface. The total number of interrupts on the receiving interface was 11,137 per second at 400 kpps, but there were only 1.4 interrupts per second at 500 kpps.

Figure 6.7(b) shows the repeated measurement but with 340 B packets, in order to compare them with configuration 6. For configuration 6, we created a custom Click element to send SIP `NOTIFY` requests, which are UDP packets. The size of the packet was 340 B, and we used the same SIP packets for configurations 1 through 5.

The MLFFR of configuration 1 was 343 kpps, configuration 2 was 343 kpps, configuration 3 was 213 kpps, configuration 4 was 117 kpps, configuration 5 was 111 kpps, and configuration 6 was 71 kpps.

There was no difference between the performance of configurations 1 and 2. The difference between configurations 2 and 3 is due to the kernel-user transition. The difference seen between configurations 3 and 4 is due to Java execution overhead. Both of these were previously seen above. Again, there is almost no difference between configurations 4 and 5. The difference between configurations 5 and 6 shows the overhead of KeepAlive beyond NetMonitor. There is a meaningful difference between the modules because the KeepAlive module must do deep packet inspection to find SIP `NOTIFY` messages, and further, we made no effort to optimize the matching algorithm.

As the size of packets increases from 64 B to 340 B, the number of packets our setup can generate decreases due to the bandwidth limitation. As a consequence, the forwarding rate

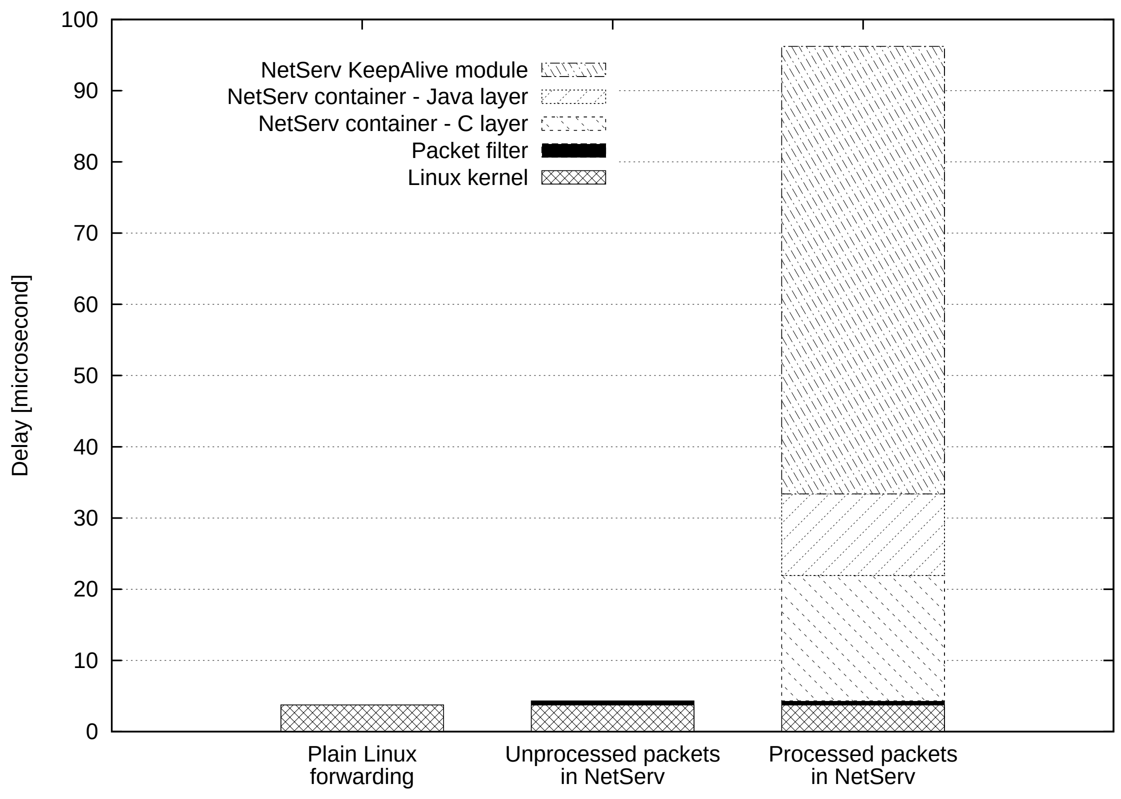


Figure 6.8: Microbenchmark.

of the router in configuration 1 and 2 reached the theoretical MLFFR of 343 kpps for the 1 Gb/s link.

Figure 6.8 shows our microbenchmark result. It compares delays as a packet travels through each layer in a NetServ node. The first bar shows only the delay in Linux kernel (configuration 1 in our MLFFR graphs), the second bar adds the delay from the kernel packet filter (configuration 2), and the third bar shows the delays in all layers up to the KeepAlive module (configuration 6). The second bar represents the delay experienced by packets transiting a NetServ node without being processed by a module. We note that the additional overhead compared to the first bar, plain Linux forwarding, is very small. The third bar, representing the full packet processing overhead, shows a significant amount of delay, as expected.

The overhead is certainly significant. Packets processed by the KeepAlive module achieve only 20% of throughput and incur 92 microsecond delay, compared to unprocessed

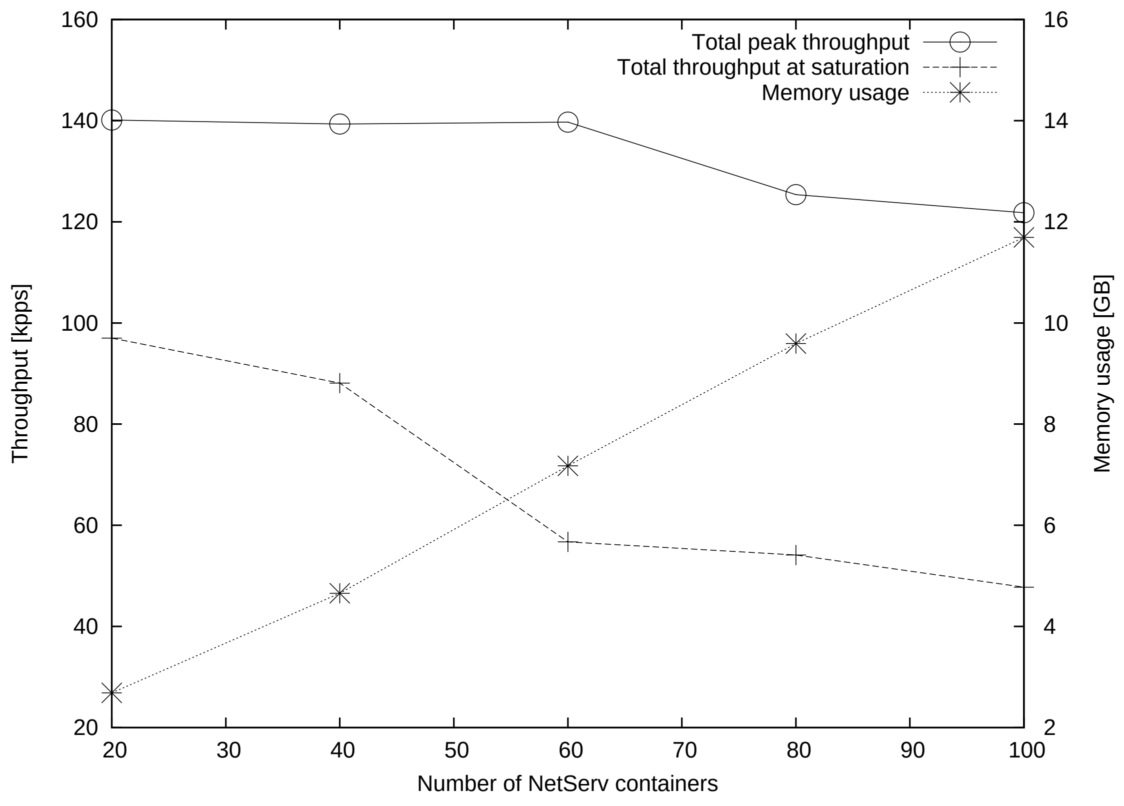


Figure 6.9: NetServ node with many containers.

packets. However, we make a few observations in our defense. The KeepAlive throughput of 71 kpps is on par with the average traffic experienced by a typical edge router [42]. Our tests were performed on modest hardware, and more importantly, a packet processing module would only be expected to handle a small fraction of the total traffic. Our Linux implementation, thus, is quite usable in low traffic environments. The OpenFlow extension in Section 8.3 provides a solution for high traffic environments.

Lastly, we observe the behavior of a NetServ node running many containers. Incoming traffic is equally distributed to each container. Figure 6.9 shows the total throughput and memory consumption as we increase the number of containers in a NetServ node. The total throughput gradually decreases, indicating the overhead of running many containers. The line labeled “peak throughput” is the maximum throughput reported just before the NetServ node started experiencing packet loss as the input rate increased. The line labeled “throughput at saturation” is the throughput when it plateaued against the increasing input rate. The overhead of running many containers, again, exacerbates the difference. Memory consumption is proportional to the number of containers. Each container consumes about 110 MB when the KeepAlive module is busy processing packets at the peak throughput. Each container contains a JVM, OSGi framework, and a collection of building block modules. Figure 6.9 shows that a NetServ node scales reasonably well as we increase the number of containers in it.

Chapter 7

Economic Model: NetServ Applications

We advocate NetServ as a platform that enables content providers and ISPs to enter into a new economic alliance. In this chapter, we present four example applications–ActiveCDN, KeepAlive Responder, Media Relay, and Overload Control–which demonstrate economic benefit for both parties.

ActiveCDN provides publisher-specific content distribution and processing. The other three applications illustrate how NetServ can be used to develop more efficient and flexible systems for real-time multimedia communication. In particular, we show how Internet Telephony Service Providers (ITSPs) can deploy NetServ applications that help overcome the most common problems caused by the presence of Network Address Translators (NATs) in the Internet, and how NetServ helps to make ITSPs' server systems more resilient to traffic overload.

7.1 ActiveCDN

We developed ActiveCDN, a NetServ application module that implements CDN functionality on NetServ-enabled edge routers. ActiveCDN brings content and services closer to end users than traditional CDNs. An ActiveCDN module is created by a content provider, who has the full control of the placement of the module. The module can be redeployed to dif-

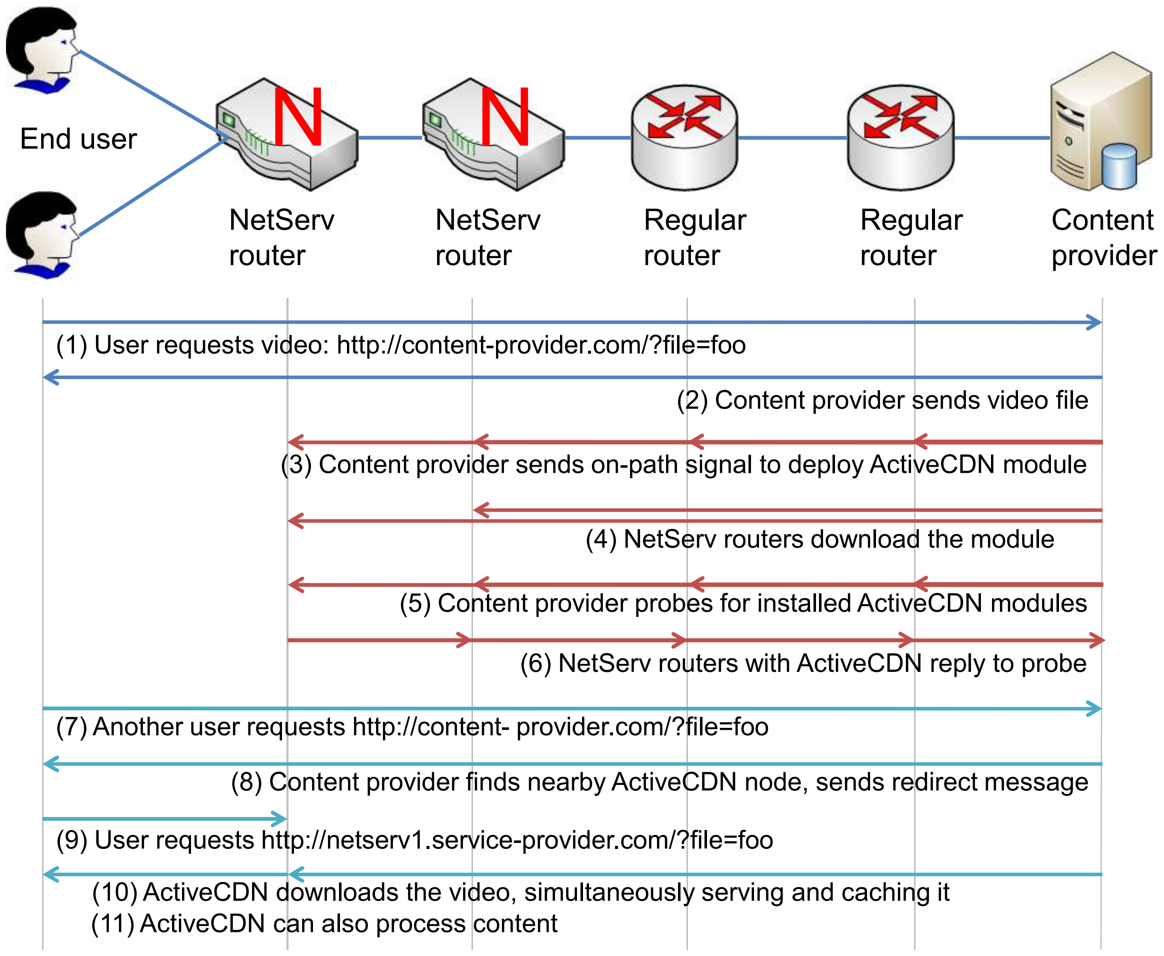

End user
NetServ router
NetServ router
Regular router
Regular router
Content provider
(1) User requests video: http://content-provider.com/?file=foo
(2) Content provider sends video file
(3) Content provider sends on-path signal to deploy ActiveCDN module
(4) NetServ routers download the module
(5) Content provider probes for installed ActiveCDN modules
(6) NetServ routers with ActiveCDN reply to probe
(7) Another user requests http://content- provider.com/?file=foo
(8) Content provider finds nearby ActiveCDN node, sends redirect message
(9) User requests http://netserv1.service-provider.com/?file=foo
(10) ActiveCDN downloads the video, simultaneously serving and caching it
(11) ActiveCDN can also process content

Figure 7.1: How ActiveCDN works.

ferent parts of the Internet as needed. This is in stark contrast to the largely preconfigured topology of existing CDNs.

The content provider also controls the functionality of the module. The module can perform custom processing specific to the content provider, like inserting advertisements into video streams.

Figure 7.1 offers an example of how ActiveCDN works. When an end user requests video content from a content provider's server, the server checks its database to determine if there is a NetServ node running ActiveCDN in the vicinity of the user. We use the MaxMind GeoIP [19] library to determine the geographic distance between the user and each ActiveCDN node currently deployed. If there is no ActiveCDN node in the vicinity, the server serves the video to the user, and at the same time, sends a `SETUP` message to deploy an ActiveCDN module on an edge router close to that user. This triggers each NetServ node on-path, generally at the network edge, to download and install the module. Following the `SETUP` message the server sends a `PROBE` message to retrieve the IP addresses of the NetServ nodes that have successfully installed ActiveCDN. This information is used to update the database of deployed ActiveCDN locations. When a subsequent request comes from the same region as the first, the content provider's server redirects the request to the closest ActiveCDN node, most likely one of the nodes previously installed. The module responds to the request by downloading the video, simultaneously serving and caching it. The content provider's server can send a `REMOVE` message to uninstall the module, otherwise the module will be removed automatically after the number of seconds specified in the `ttl` field of the `SETUP` message. The process repeats when new requests are made from the same region.

7.2 KeepAlive Responder

The ubiquitous presence of Network Address Translators (NATs) poses a challenge to communication services based on Session Initiation Protocol (SIP) [106]. After a SIP User Agent (UA) behind a NAT box registers its IP address with a SIP server, the UA needs to make sure that the state in the NAT box remains active for the duration of the registration. Failure to keep the state active would render the UA unreachable. The most common mechanism used by UAs to keep NAT bindings open is to send periodic keep-alive messages to the SIP server.

The timeout for UDP bindings appears to be rather short in most NAT implementations. SIP UAs typically send keep-alive messages every 15 seconds [49] to remain reachable from the SIP server.

While the size of a keep-alive message is relatively small–about 300 bytes when SIP messages are used for this purpose, which is often the case–large deployments with hundreds of thousand or even millions of UAs are not unusual. Millions of UAs sending a keep-alive every 15 seconds represent a significant consumption of network and server resources. This traffic wastes energy, adds to the operating cost of Internet Telephony Service Providers

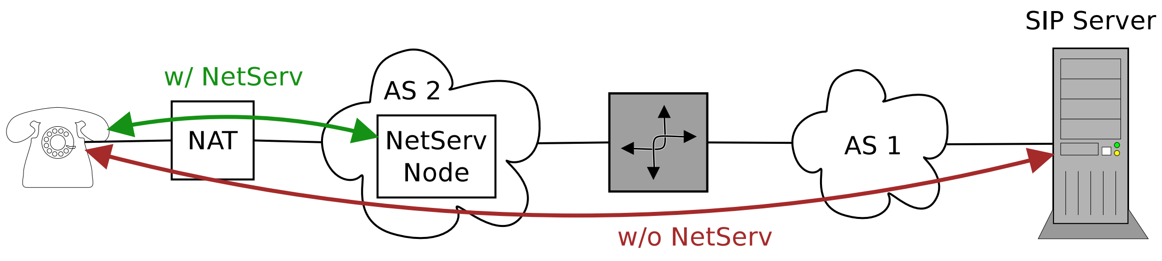


Figure 7.2: Operation of KeepAlive Responder.

(ITSPs), and serves no useful purpose–other than to fix a problem that should not exist in the first place. A surprising fact is that the keep-alive traffic can be a bottleneck in scaling a SIP server to a large number of users [49].

Figure 7.2 shows how NetServ could help offload NAT keep-alive traffic from the infrastructure of Internet Telephony Service Providers (ITSPs). Without the NetServ KeepAlive Responder, the SIP UA behind a NAT sends a keep-alive request to the SIP server every 15 seconds and the SIP server sends a response back. The NAT keep-alive packets can be either short 4-byte packets or full SIP messages. For our implementation, we are using full SIP messages because, to the best of our knowledge, this is what most ITSPs use for reliability reasons. When an NSIS-enabled SIP server starts receiving NAT keep-alive traffic from a SIP UA, it initiates NSIS signaling in order to find a NetServ router along the network path to the SIP UA. If a NetServ router is found, the router downloads and installs the KeepAlive module provided by the ITSP.

After the module has been successfully installed, it starts inspecting SIP traffic going through the router towards the SIP server. If the module finds a NAT keep-alive request, it generates a reply on behalf of the SIP server, sends it to the SIP UA, and discards the original request. Thus, if there is a NetServ router close to the SIP UA, the NAT keep-alive traffic never reaches the network or the servers of the ITSP; the keep-alive traffic remains local in the network close to the SIP UA.

The KeepAlive Responder spoofs the IP address of the SIP server in response packets sent to the UA. IP address spoofing is not an issue here because the NetServ router is on-path between the spoofed IP address and the UA.

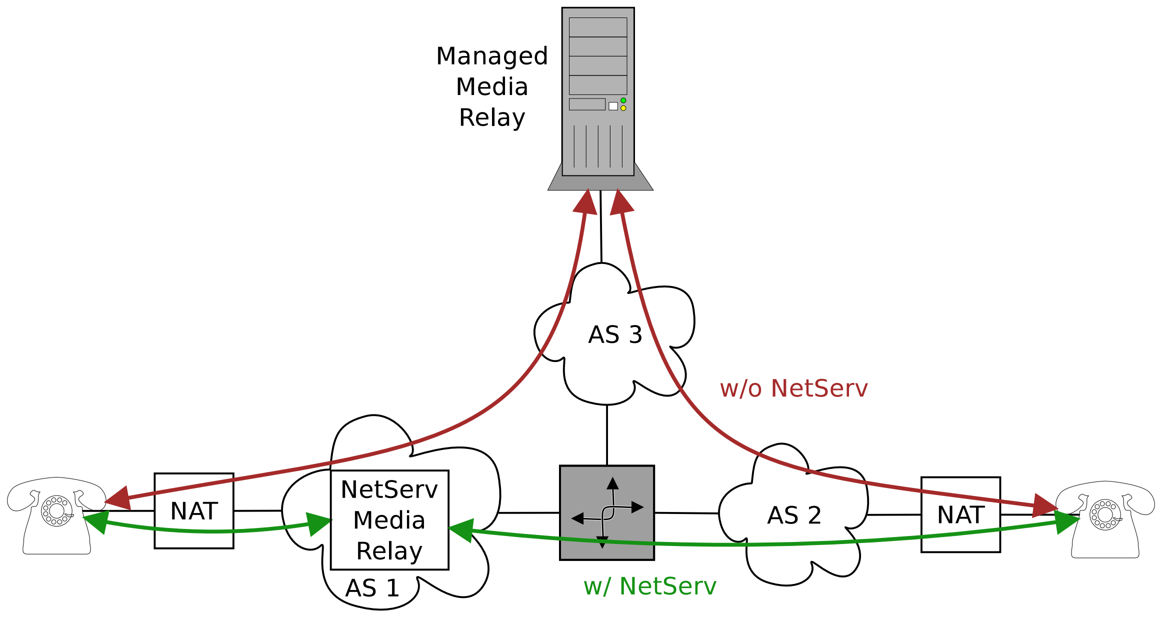


Figure 7.3: Operation of NetServ Media Relay.

7.3 Media Relay

ITSPs may need to deploy media relay servers to facilitate the packet exchange between NATed UAs. However, this approach has several drawbacks, including increased delay, additional hardware and network costs, and management overhead.

Figure 7.3 shows how NetServ helps to offload the media relay functionality from an ITSP's infrastructure. The direct exchange of media packets between the two UAs in the picture is not possible. Without NetServ the ITSP would need to provide a managed media relay server. When a NetServ router is available close to one of the UAs, the SIP server can deploy the Media Relay module at the NetServ node.

When a UA registers its network address with the SIP server, the SIP server sends an NSIS signaling message towards the UA, instructing the NetServ routers along the path to download and install the Media Relay module. The SIP server then selects a NetServ node close to the UA, instead of a managed server, to relay calls to and from that UA.

NetServ media relay servers that are deployed at the network edge nicely fit into the Internet Connectivity Establishment (ICE) [105] framework and can be used as TURN servers [96] within the framework. ICE-capable user agents (not necessarily SIP-based) can use the framework to discover whether a TURN server is required to establish a communication session. The algorithm to select an optimal server from publicly available TURN servers across the Internet is left unspecified in the framework. NetServ-capable nodes can facilitate deployment of media relay servers across the Internet. The capability of NSIS signaling to select a relay server close to one of the communicating UAs helps select relay servers that add no (or very low) additional delay to media packets.

The use of TURN-based media relay servers is not limited to SIP UAs. A large number of globally distributed media relay servers are required in many other communication scenarios, such as peer-to-peer file sharing, high definition multimedia communication and video streaming. NetServ nodes distributed across the Internet facilitates the deployment of a media relay network.

7.4 Overload Control

Considerable amount of work has been done on overload of SIP servers [79]. SIP servers are vulnerable to overload due to the lack of congestion control in UDP. The IETF has developed a framework for overload control in SIP servers that can be used to mitigate the problem [74,79]. The framework proposes to implement the missing control loop (otherwise implemented in TCP) in SIP. Figure 7.4 illustrates the scenario. The SIP server under load, referred to as the Receiving Entity (RE), periodically monitors its load. The information about the load is then communicated to the Sending Entity (SE), which is the upstream SIP server along the path of incoming SIP traffic. Based on the feedback from the RE, the SE then either rejects or drops a portion of incoming SIP traffic.

We implemented a simple SIP overload control framework in NetServ. Our Receiving Entity (RE) is a common SIP server based on the SIP-Router [31] project. We extended the SIP server implementation with functions needed to initiate NSIS signaling and monitor the load of the server. For the sake of simplicity we used a statically configured load threshold in our prototype implementation. In real-world scenarios the load of the SIP server would be calculated as a function of CPU load, memory utilization, database utilization, and other factors that limit the total volume of traffic the server can handle. When the load on the

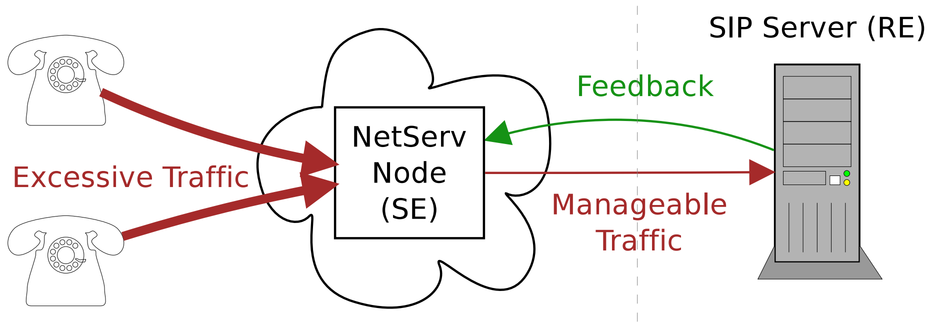


Figure 7.4: NetServ as SIP overload protection.

SIP server exceeds a preconfigured threshold, the SIP server starts sending NSIS signals towards the UAs in an attempt to discover a NetServ node along the path and install the SE NetServ module on the node. Once the module is successfully installed, it intercepts all SIP messages going to the SIP server. Based on the periodic feedback about the current volume of traffic seen by the SIP server, the module adjusts the amount of traffic it lets through in real time. The excess portion of incoming traffic is rejected with "503 Service Unavailable" SIP responses.

Without NetServ, an ITSP's options in implementing overload control are limited. The ITSP can put both the SE and the RE in the same network. Such configuration only allows hop-by-hop overload control, in which case excessive traffic enters the ITSP's network before it is dropped by the SE. Since all incoming traffic usually arrives over the same network connection, using different control algorithms or configurations for different sources of traffic becomes difficult.

With NetServ, the ability to run an SE implementation at the edge of the network makes it possible to experiment with control algorithms and configurations for different sources of traffic. Being able to install and remove a NetServ SE module dynamically makes it easy for an ITSP to change the traffic control algorithm. Since the NetServ SE module is installed outside the ITSP's network, excess traffic is rejected before it enters the ITSP's network, protecting not only the SIP server, but also the network connection.

7.5 Discussion on Signaling Mode

7.5.1 Reverse Data Path

The descriptions of the NetServ applications in this chapter assumed that the reverse data path is the same as the forward path. On the Internet today, however, this is often not the case due to policy routing.

For ActiveCDN and Media Relay, this is not an issue. The modules only need to be deployed *closer* to the users, not necessarily on the forward data path. The module will still be effective if the network path from the user to the NetServ node has a lower cost than the path from the user to the server.

For KeepAlive Responder and Overload Control, the module must be on-path to carry out its function. However, this is not a serious problem in general. First, NetServ routers are located at the network edge. It is unlikely that the reverse path will go through a different edge router. Even in the unlikely case that a module is installed on a NetServ router which is not on the reverse path, if we assume a dense population of users, it is likely that the module will serve some users, albeit not the ones who triggered the installation in the first place. If a module is installed at a place where there is no user to serve, it will time-out quickly.

If a reverse on-path installation is indeed required, there are two ways to handle it. First, the client-side software can initiate the signaling instead of the server. But this requires modification of the client-side software. Second, the server can use round-trip signaling. We implemented `TRIGGER` signaling message in NetServ NSLP. The server encapsulates a `SETUP` or `PROBE` in a `TRIGGER`, and sends it towards the end user. The last NetServ router on-path creates a new upstream signaling flow back to the server. This approach, however, assumes that the last NetServ node is on both forward and reverse path, and increases the signaling latency.

7.5.2 Off-path Signaling

In addition to on-path signaling, we envision that certain cases of off-path signaling would be useful for some NetServ applications. There is a proposal to extend NSIS to include epidemic signaling [67]. The proposed extension will enable three additional modes of signal dissemination: (1) signaling around the sender (*bubble*), (2) signaling around the receiver (*balloon*), and (3) signaling around the path between the sender and receiver (*hose*). The bubble and balloon modes will be useful for NetServ module deployment within an enterprise environment. The hose mode will be useful for the scenarios where NetServ nodes are not exactly on-path, but a couple of hops away. This mode can mitigate the aforementioned concerns about the divergent reverse path.

Chapter 8

Scaling NetServ using OpenFlow

8.1 Introduction

The Linux-based implementation that we described in Chapter 6 has a limitation in terms of performance. Multiple layers present in our execution environment introduce significant overhead when a packet is subject to DPI, as our evaluation has shown in Section 6.3. A more serious limitation is the fact that the scalability is limited to a single Linux box, even when no packet processing is performed. In general, a general-purpose PC cannot match the forwarding performance of a dedicated router. This makes our Linux implementation unsuitable for high traffic environment.

We can address this limitation by offloading the forwarding plane onto a physically separate hardware element, capable of forwarding packets at line rate. The hardware device must also provide dynamically installable packet filtering hooks, so that the packets that need to be processed by NetServ modules can be routed appropriately to one or more NetServ nodes which are attached to the hardware device.

The OpenFlow programmable switch architecture provides exactly the capabilities that we need. In this chapter, we briefly explain the OpenFlow architecture, and describe our prototype implementation of the OpenFlow extension of NetServ.

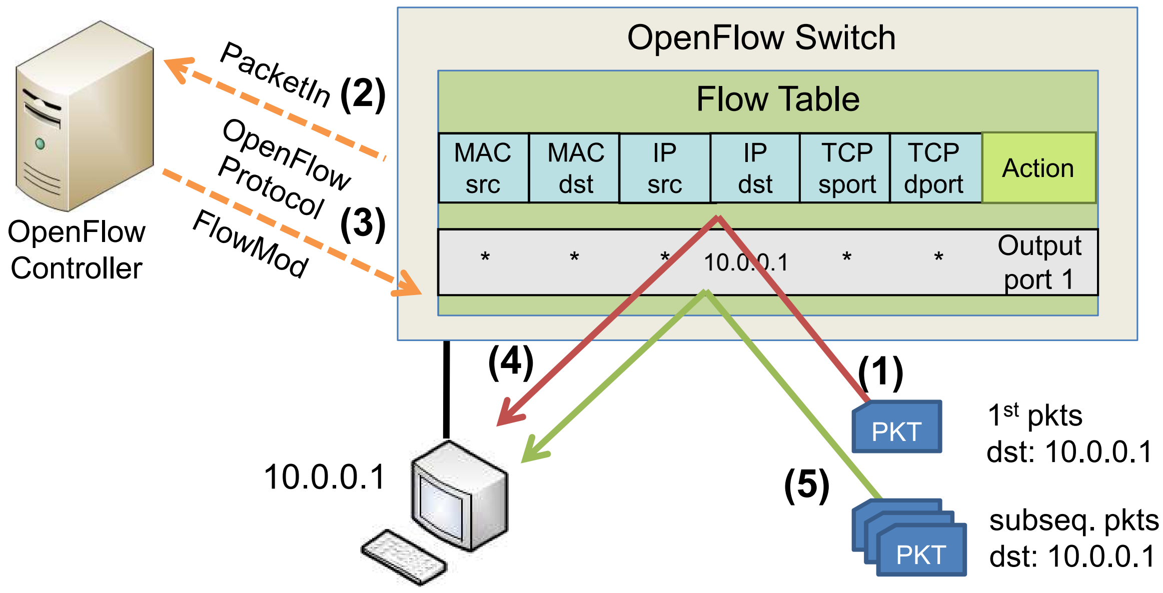


Figure 8.1: How OpenFlow works.

8.2 Overview of OpenFlow

An OpenFlow switch [99] is an Ethernet switch with its internal flow table exposed via a standardized interface to add and remove flow entries. The OpenFlow Controller (OFC), typically a software program running on a remote host, communicates with the switch over a secure channel using the standard OpenFlow Protocol. An entry in the flow table defines a mapping between a set of header fields–MAC/IP addresses and port numbers, for example–and one or more associated actions, such as dropping a packet, forwarding it to a particular port on the switch, or even simple modifications of header fields. When a packet arrives at an OpenFlow switch, the switch looks up the flow table. If an entry matching the packet header is found, the corresponding actions are performed. If no entry matches the packet header, the packet is sent to the remote OFC, which will decide what to do with the packet, and also insert an entry into the switch's flow table so that subsequent packets of the same flow will have a matching entry.

Figure 8.1 illustrates this process. (1) A packet destined for 10.0.0.1 arrives at an OpenFlow switch, which contains no matching entry in its flow table. (2) A `PacketIn` message is sent to the OFC. The OFC, after consulting its routing table, determines the switch port to which the incoming packet should be output. (3) The OFC sends a `FlowMod` command to the switch to add a flow table entry. (4) The command also include an instruction to forward the incoming packet, which has been sitting in a queue waiting for the verdict from the OFC. The packet goes out to the destination. (5) All subsequent packets destined for 10.0.0.1 match the new flow table entry, so the packets are forwarded by the hardware switch at line rate without incurring the overhead of making a round trip to OFC.

8.3 NetServ on OpenFlow

NetServ on OpenFlow integrates the two technologies in two ways. First, one or more NetServ nodes are attached to an OpenFlow switch. From NetServ's point of view, the OpenFlow switch provides a common forwarding plane for multiple NetServ nodes. From OpenFlow's point of view, the NetServ nodes are external packet processing devices. (The OpenFlow paper [99] envisions such devices based on NetFPGA.)

Second, the OpenFlow Controller (OFC) is now implemented as a NetServ module. As such, the OFC can be dynamically installed, updated, or moved to another node. Furthermore, there can be many OFCs, one per user, or even one per application. In conjunction with FlowVisor–a special purpose OFC that acts as a transparent proxy for a group of OFCs–NetServ-based OFCs will open up interesting possibilities like an in-network service that reconfigures network topology as needed. Further exploration is planned as future work.

Figure 8.2 shows what happens when a packet destined for 10.0.0.1 is being processed by a NetServ node attached to an OpenFlow switch. First of all, the OFC running in NetServ sends a proactive `FlowMod` command to direct all NSIS signaling messages to the NetServ controller. (1) When the NetServ controller receives a SETUP message–thanks to the proactive flow table entry–it installs the requested packet processing application module. In addition, the NetServ controller tells the OFC that an application module has requested a packet filter, and the OFC remembers the fact in preparation for incoming packets, but it does not add a flow table entry at this point. (2) A packet matching the filtering rule of

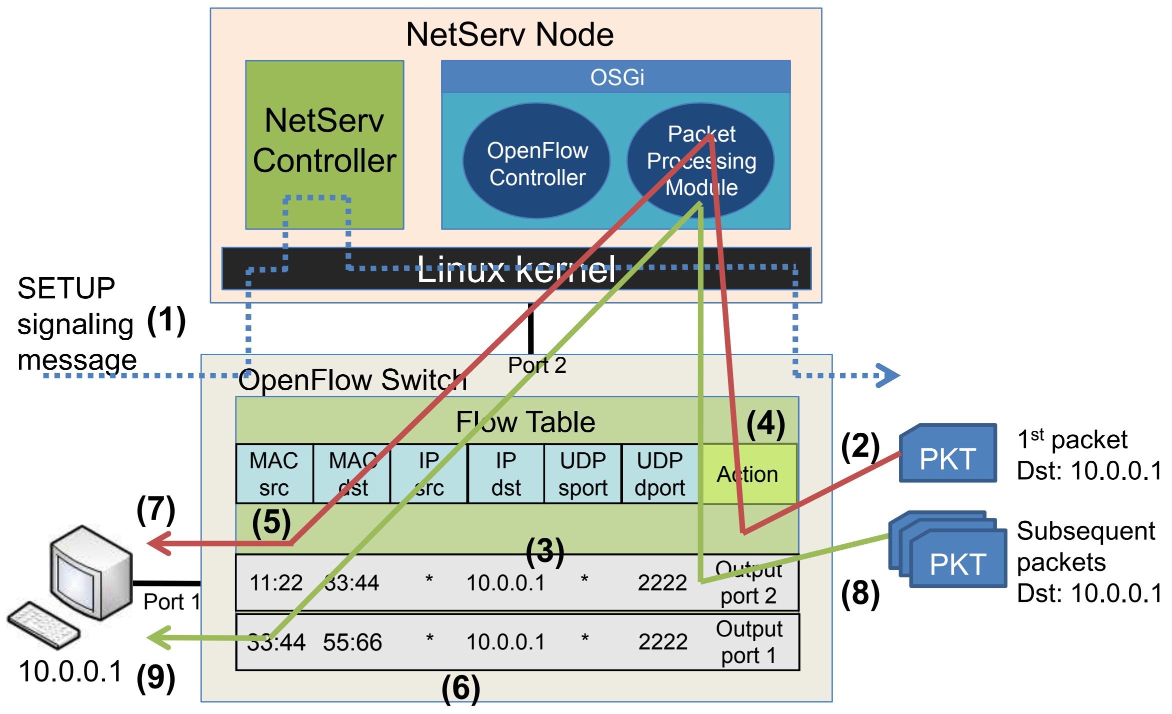


Figure 8.2: NetServ with OpenFlow extension.

the NetServ application arrives. There is no flow table entry for it, so it goes to the OFC. The OFC in NetServ, before it begins its usual OFC work of consulting its routing table, notices that the packet matches the application filtering rule that the NetServ controller has told the OFC earlier. (3) The OFC translates the NetServ application's packet filter into a flow table entry, and injects it into the OpenFlow switch so that the packet will be routed to the NetServ node. (4) The packet is delivered to the NetServ node, and then to the appropriate application module. (5) After processing the packet, the NetServ node sends it back to the switch. At that point, however, the packet must go back to the OFC, so that it can find its switching destination. When the OFC sees the packet the second time around, it knows that it has been processed by a NetServ module (from the input switch port), so it goes straight to the normal OFC work of consulting its routing table. (6) The OFC injects another flow table entry in order to output the packet. (7) The packet goes to the destination. (8) Subsequent packets matching the NetServ application's filtering rule are now routed directly to the NetServ node without going to the OFC first (because of

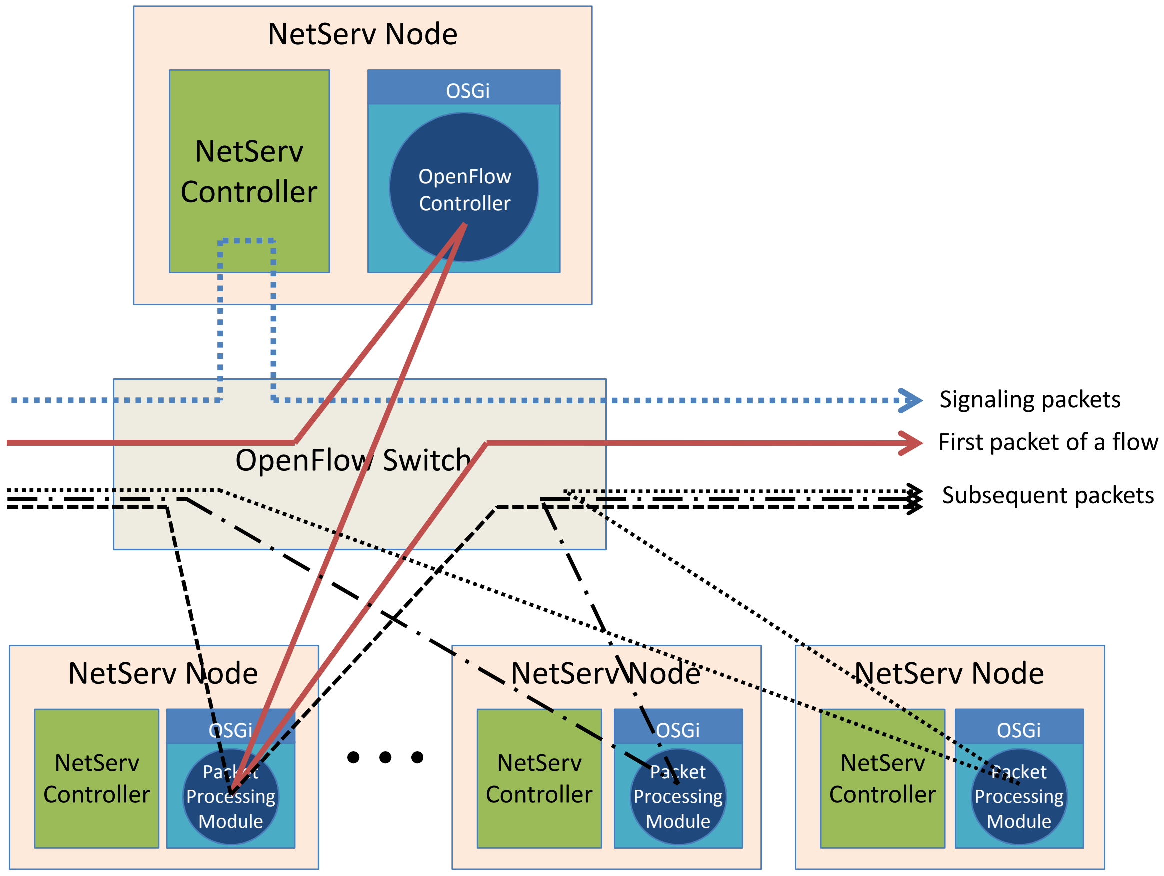


Figure 8.3: Multiple NetServ nodes attached to an OpenFlow switch.

the flow table entry added in step (3).) (9) And when those subsequent packets come back from the NetServ node to the switch, there is the flow table entry added in step (6) to guide them to the correct output port.

Having a separate hardware-based forwarding plane eliminates the performance problem for the packets that do not go through a NetServ node. For the packets that need to go through NetServ, the OpenFlow extension does not reduce individual packet processing time, but we can increase the throughput by attaching multiple NetServ nodes. Different flows can be assigned to different NetServ nodes, or depending on applications, a single flow can use multiple NetServ nodes. Figure 8.3 depicts this scenario.

Chapter 9

Active Networking and NetServ

9.1 Introduction

Tennenhouse and Wetherall presented the vision of an active network architecture in their seminal 1996 paper [116]. They noted that growing demand for in-network services resulted in the proliferation of middleboxes, overcoming "architectural injunctions against them." By adopting active technologies already available at end systems–mobile code between web server and client, for example–they proposed to activate network nodes, making in-network computation and storage available to users.[1] They argued that active networks not only consolidate the ad hoc collection of middleboxes into a common programmable node, but also accelerate the pace of innovation. The possibility of in-network deployment enables new network-based services, and those new ideas are no longer shackled by the slow pace of protocol standardization.

It is remarkable that, 15 years later, their voice rings even louder today. Middleboxes have continued to proliferate. NAT boxes are everywhere, from enterprise networks to home networks. Web proxies and load balancers are growing in numbers and capability, recently coining a new term, *application delivery controller*, to refer to the most sophisticated breed. Even traditional router vendors are jumping in with SDKs to allow third party packet processing modules [87].

[1] We use the term *users* broadly, referring not only end users, but also application service providers and content providers.

The ossification of the network layer has gotten to a point where researchers are no longer hesitant to call for a clean-slate redesign of the Internet, but we have yet to see a clear winner with a serious prospect of adoption. In the meantime, content and application providers' need for in-network services are filled by application-layer solutions that can make suboptimal use of the network. Witness the emergence of the Content Distribution Network (CDN) industry.

The rise of CDNs has also contributed to a recent trend: blurring of the lines between content providers and Internet service providers (ISPs). Some very large content providers–Google, for example–operate data centers at Internet exchange points. Some traditional ISPs, on the other hand, are getting into CDN market–Level 3 hosting and delivering Netflix's streaming video, for example. This trend highlights the benefit of operating services at the strategic points within the network.

Despite the far-reaching vision, however, the advocates of active networks ultimately failed to win over the networking community at the time. The biggest objections were the security risk and performance overhead associated with the extreme version of active networks where every user packet carries code within it. Another important factor, in our opinion, was the lack of compelling use cases.

Active networking was ahead of its time when it was proposed, but we believe its time has arrived. The technology advances in the past fifteen years provides a solid ground on which we can design an active networking system that strikes the right balance to address both security and performance concerns. Moreover, we observe that active networks present a compelling use case in today's Internet economy.

In this chapter, we consider NetServ as an active network system. In fact, we claim that NetServ can be viewed as a fully integrated active network system that provides all the necessary functionality to be deployable, addressing the core problems that prevented the practical success of earlier approaches.

In contrast to the earlier active network systems, NetServ has the following characteristics:

- A **hybrid approach** that combines the best qualities from the two extreme approaches to active networking: *integrated* and *discrete* active networks. User-initiated on-path signaling allows user-code injection into the network without the danger of code-carrying packets.

- The right balance between **security and performance**. User code modules run in isolated user space JVMs for security. Packet processing in user space Java code achieves a reasonable performance in low-traffic environments. Scalability in high-traffic environments is achieved by the NetServ OpenFlow extension, where multiple NetServ nodes can be attached to a hardware OpenFlow switch.

- A compelling **economic model** on top of the newly available in-network resources. Content providers can operate closer to end-users by deploying code modules on eyeball ISPs' network nodes. Eyeball ISPs find a new source of revenue. Four example applications illustrate this model.

In Section 9.2, we describe how these three characteristics address the general challenges faced by the earlier active network systems. In Section 9.3, we compare NetServ with other active network systems, and examine other closely related work on providing in-networking services, such as programmable routers, network testbeds, and content caching.

9.2 Addressing Three Challenges of Active Networks

9.2.1 Hybrid Approach: User Code, but Not in Data Packets

Active networking proposed two approaches to programming the network. In the *integrated* approach, every packet contains user code that is executed by the network nodes through which the packet travels. Many researchers attribute the ultimate demise of active networks to the security risk and performance overhead associated with user packets carrying code.

In the more conservative *discrete* approach, network nodes are programmed by out-of-band mechanisms which are separate from the data packet path. In other words, the discrete active network nodes are programmable routers. Indeed, since the active network proposal, the research community has seen many programmable router proposals [62,80,86,89] which are either considered a platform for active networking, or at least heavily influenced by it.

Notwithstanding the general view that associates programmable routers with active networks, we do not consider typical programmable routers an adequate platform to realize the active network vision. Typical uses of programmable routers center around the network functions required by the network operators, like QoS, firewall, VPN, IPsec, NAT, web cache, and rate limiting. The variety and sophistication of available services on programmable routers is a boon for network management, but it is far from the active network vision, where users inject custom functionality into the network. In fact, we argue that programmable routers, despite their root in active networks, compound the problem that motivated active networks in the first place: proliferation of middleboxes.

NetServ aims to be the vehicle to bring back the active networking vision of ubiquitous in-network services, not just another programmable router. NetServ must provide a mechanism to inject user code into the network. At the same time, we cannot repeat the same failure by adopting the integrated approach.

We take a hybrid approach. Like the discrete approach, we separate the data path and the control channel through which the network nodes are programmed. Like the integrated approach, however, it is the user who programs the network nodes. A user sends an on-path signaling message towards a destination of his interest, which will trigger the NetServ nodes on-path to download the user's code module and install it dynamically.

We provide further comparisons between NetServ and previous active network systems in Section 9.3.1.

9.2.2 Striking the Right Balance: Security and Performance

The user-driven software installation made security our top priority. Unlike previous programmable routers that ran service modules in (or very close to) kernel space for fast packet processing, NetServ runs modules in user space. Specifically, user modules are written in Java and executed on Java Virtual Machines (JVMs). A NetServ node hosts multiple JVMs, one for each user.

Our choice of user space execution and JVM allows us to leverage the decades of technology advances in operating systems, virtualization, and Java. NetServ makes use of isolation and resource control mechanisms available in all layers: OS-level virtualization, process control, Java 2 security, and the OSGi component framework. We have discussed these mechanisms in detail in Section 6.2.

Running service modules in user space, and in Java on top of that, inevitably raises the eyebrows of performance-minded critics. In Section 6.3, we explored the most worrisome case, namely, a Java service module sitting in the fast data path, and performing deep packet inspection (DPI) and modification. Every processed packet incurs the overhead of kernel packet filter, kernel-to-user (and back) transitions, transfer from native to Java code, and application code running in JVM. The evaluation of our Linux-based implementation shows that the overhead is indeed significant, but not prohibitively so. The throughput achieved on a modestly equipped Linux server matches the average traffic seen by a typical edge router, indicating that the solution is quite usable in a low traffic environment. And this is with every single packet being processed. Typically only a small fraction of incoming packets will be subject to DPI.

Our real defense against performance-related criticisms is the multi-box lateral expansion of NetServ using the OpenFlow [99] forwarding engine, described in Section 8.3. In this extended architecture, multiple Linux-based NetServ nodes are attached to an OpenFlow switch, which provides a physically separate forwarding plane. The scalability of user services is no longer limited to a single NetServ box.

We do not claim to have invented any of the individual technologies that we use for NetServ. Our challenge, and thus our contribution, lies in combining the technologies to strike the right balance between security and performance, culminating in a fully-integrated active network system that can be deployed on the current Internet while remaining true to the original active networking vision.

9.2.3 Economic Alliance between Content Providers and ISPs

It can be argued that active networks might have been more successful if there had been economically compelling use cases. Fifteen years ago, the Internet was a much simpler place with its actors playing well-defined roles. Perhaps it was difficult to imagine a workable economic model for deploying services across ownership boundaries. Things are different today. The stakeholders in the Internet market place are engaged in fierce competitions and swift alliances to occupy strategic positions as the lines between different actors begin to disappear.

We identify two Internet actors that are currently in a tussle, and suggest a way to use NetServ to enter into an economic alliance. We have already noted that the lines between content providers and ISPs are blurring, which highlights the importance of occupying strategic points in the network. Those strategic points are often at the network edge. Content providers are motivated to operate at the network edge, close to end users, as evidenced by the success of CDN operators like Akamai [1].

The network edge belongs to eyeball ISPs, as we explored in Chapter 5. NetServ can enable an economic relationship between the ISPs and the content providers. We have illustrated our vision using four example applications in Chapter 7. We envision that the economic alliance between content providers and ISPs will be facilitated by brokers who aggregate resources from different ISPs, arrange remuneration, and possibly provide value-added services. This is already happening in cloud computing [25,35].

9.3 Earlier Active Network Systems and Other Related Work

9.3.1 Active Networks

The most radical active networking proposals–collectively called *integrated* active networks–replace data packets with *capsules*, which carry code along with data [48,111,122]. Capsule-based systems were either unable to alleviate the security concerns or, in an attempt to secure the systems, limited in functionality. Some systems were catered to specific applications–Smart Packet [111] were tailored for network management, for example. Some systems were based on restricted programming languages specifically designed for active networks–PLAN [78], for example–rather than a general purpose programming language.

NetServ uses on-path signaling to achieve the same objective as capsules, namely, in-network user-code propagation and injection. However, the actual code modules are downloaded out-of-band, so they are not subject to size restrictions, and the code modules can go through more extensive security checks. NetServ uses Java, a mainstream general-purpose language. Performance improvements and the advances in resource control and isolation techniques in the past 15 years have made Java a viable proposition today.

Out-of-band code module download is not new. The more conservative active networking proposals–collectively referred to as *discrete* active networks–download code modules out-of-band. Packets in DAN [63] and ANTS [121] contain references to predefined functions rather than actual executable code. When a packet transits an active node, the function references in the packet triggers the invocation of the functions residing in the node. If the code module for an invoked function is not present in the node, the module is downloaded on demand. DAN uses client-server paradigm for code download, whereas ANTS uses peer-to-peer.

The on-demand code downloading of discrete active networks may look similar to NetServ. The critical difference is that the code modules for discrete active networks are not arbitrary user code. They are predefined set of protocol or service implementations that were prepared by the network administrator. Users *select* which function to invoke, but do not create the functions.

SwitchWare combines integrated and discrete active networking by letting packets carry restricted code for simple computations, but having them call predefined *Switchlets* for more complex computations.

NetScript [61, 123] proposes an abstraction for the entire network, rather than individual nodes. NetScript Virtual Network (NVN) is an overlay network composed of Virtual Network Engines (VNEs) and Virtual Links (VLs) between them. NetScript is a specialized language for data stream processing. NetServ's packet processing framework provides similar functionality. In fact, NetServ can be viewed as an alternate implementation of the NetScript vision using a mainstream general-purpose programming language and operating directly on the IP underlay rather a virtual overlay network.

The influence of active network concept can be felt in many areas of networking today. The slice-level programmability in global testbeds such as PlanetLab [102] or GENI [12] can be regarded as a form of discrete active networks. Software-Defined Networking (SDN) [32] based on OpenFlow [99] is another call for the return to the active network idea. Geambasu *et al.* [72] applied active network concept to overlay networks, allowing a common distributed hash table (DHT) substrate to be used for multiple applications that have diverse customization needs.

9.3.2 Programmable Routers

Many earlier programmable routers focused on providing modularity without sacrificing forwarding performance, which meant installing modules in kernel space. Router Plugins [62], Click [89], PromethOS [86], and Pronto [80] followed this approach. As we noted before, NetServ runs modules in user space. These kernel-level programmable routers, in fact, can be used as NetServ's forwarding plane. We described alternate NetServ forwarding engine based on Click router and JUNOS SDK in Section 6.1.3.

Another candidate for NetServ forwarding plane is Supercharged PlanetLab [117], a system based on network processor.

LARA++ [109] is similar to NetServ in that the modules run in user space. However, LARA++ focuses more on providing a flexible programming environment by supporting multiple languages, XML-based filter specification, and service composition. It does not employ a signaling protocol for dynamic code installation.

PromethOS [86] was the first to propose a signaling protocol for automatic deployment of modules in routers that were defined in the source routing path of the signaling message. In contrast, NetServ provides on-path signaling (no need to specify source routes). Moreover, their approach does not provide isolation of execution environments.

9.3.3 GENI

GENI [12] is a federation of many existing network testbeds under a common management framework. GENI is comprised of a diverse set of platform resources, which are shared among many experimenters.

NetServ is becoming a resident tool of the GENI infrastructure. NetServ's common execution environment can accelerate development, deployment and testing of experiments. NetServ's Java-based API makes GENI a gentler environment for educational use.

The Million Node GENI project [26], which is a part of GENI, provides a peer-to-peer hosting platform using Python-based sandbox. An end user can contribute resources from his own computer in exchange for the use of the overlay network. The Million Node GENI focuses on end systems rather than in-network nodes.

9.3.4 Content Caching

Google Global Cache (GGC) [75] refers to a set of caching nodes located in ISPs' networks, providing CDN-like functionality for Google's content. NetServ can provide the same functionality to other content providers, as we have demonstrated with ActiveCDN module.

One of the goals of Content Centric Networking (CCN) [81] is to make the local storage capacity of nodes across the Internet available to content providers. CCN proposes a replacement of IP by a new communication protocol, which addresses data rather than hosts. NetServ aims to realize the same goal using the existing IP infrastructure. In addition, NetServ enables content processing in network nodes.

Part III

Conclusions

Chapter 10

Conclusions

This thesis presents NetServ, a node architecture for dynamically deploying in-network services on edge routers. Network functions and applications are implemented as software modules which can be deployed at any NetServ-enabled node on the Internet, subject to policy restrictions. The NetServ framework provides a common execution environment for service modules and the ability to dynamically install and remove the services without restarting the nodes.

There are many challenges in designing such a system. The main contribution of this thesis lies in meeting those challenges. First, to address the challenge of providing economic incentives for enabling in-network services, we demonstrate how NetServ can facilitate an economic alliance between content providers and ISPs. ISPs make their NetServ-enabled edge routers available for hosting content providers' applications and contents. Content providers can operate closer to end users by deploying code modules on NetServ-enabled edge routers. Second, to address the challenge of accommodating both server applications and router functions in a single node, NetServ framework can host a packet processing module that sits in the data path, a server module that uses the TCP/IP stack in the traditional way, or a combined module that does both. Third, in order to provide a deployment mechanism where content providers can initiate module install without knowing the target routers, we adopted on-path signaling. A NetServ signaling packet gets forwarded by IP routers as usual, but when it transits a NetServ-enabled router, the message gets intercepted and passed to the NetServ control layer. Fourth, to address the challenge of providing a robust multi-user execution environment, we chose to run NetServ modules in user space JVM. This allows us to leverage the decades of technology advances in operating systems, virtualization, and Java. Lastly, in order to provide scalability beyond a single-box, we present a multi-box lateral expansion of NetServ using the OpenFlow forwarding engine. In this extended architecture, multiple NetServ nodes are attached to an OpenFlow switch, which provides a physically separate forwarding plane.

We built four NetServ applications: ActiveCDN, KeepAlive Responder, Media Relay, and Overload Control. The applications demonstrate the economic benefits for the content providers and the ISPs who enter into a cooperative relationship using NetServ. ActiveCDN provides provider-specific content distribution and processing. The other three applications illustrate how NetServ can be used to develop more efficient and flexible systems for real-time multimedia communication. In particular, we show how Internet Telephony Service Providers (ITSPs) can deploy NetServ applications that help overcome the most common problems caused by the presence of Network Address Translators (NATs) in the Internet, and how NetServ helps to make ITSPs' server systems more resilient to traffic overload.

We compare NetServ with active networks. The idea to enable in-network services is not new. Active networking articulated the same vision more than a decade ago. In fact, active networks went even further, advocating the infamous *integrated* approach, where every packet can carry a program that can alter the behavior of network nodes. We explain how NetServ addresses the main challenges of active networks. We argue, in fact, that NetServ can be viewed as the first fully integrated active network system that provides all the necessary functionality to be deployable, addressing the core problems that prevented the practical success of earlier approaches.

Additionally, this thesis presents our prior work on improving service discovery in local and global networks. The service discovery work makes indirect contribution because the limitations of local and overlay networks encountered during those studies eventually led us to investigate in-network services, which resulted in NetServ. Specifically, we investigate the issues involved in bootstrapping large-scale structured overlay networks, present a tool to merge service announcements from multiple local networks, and propose an enhancement to structured overlay networks using link-local multicast.

Part IV

Bibliography

Bibliography

[1] Akamai. http://www.akamai.com/.

[2] Apache XML-RPC. http://ws.apache.org/xmlrpc/.

[3] Apple Inc. http://www.apple.com/.

[4] Avahi. http://avahi.org/.

[5] Bonjour for Windows. http://www.apple.com/support/downloads/bonjourforwindows.html.

[6] Cygwin home page. http://www.cygwin.com/.

[7] Debian GNU/Linux. http://www.debian.org/.

[8] DNS-SD service types. http://www.dns-sd.org/ServiceTypes.html.

[9] Eclipse Equinox. http://www.eclipse.org/equinox/.

[10] FFmpeg. http://ffmpeg.org/.

[11] FreeNSIS. http://user.informatik.uni-goettingen.de/~nsis/.

[12] GENI. http://www.geni.net/.

[13] Gnutella protocol 0.6. http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html.

[14] ISPs experimenting with new P2P controls. http://www.networkworld.com/news/2008/061908-nxtcomm-isp-p2p.html?page=1.

[15] Limelight Networks. http://www.limelightnetworks.com/.

[16] LimeWire. http://www.limewire.com/.

[17] LogMeIn Hamachi. https://secure.logmein.com/products/hamachi/vpn.asp.

[18] lxc Linux Containers. http://lxc.sourceforge.net/.

[19] MaxMind GeoIP. http://www.maxmind.com/app/ip-location/.

[20] NSIS-ka. https://projekte.tm.uka.de/trac/NSIS/wiki/.

[21] OIT filters mDNS. http://www.net.princeton.edu/filters/mdns.html.

[22] OSGi Technology. http://www.osgi.org/About/Technology/.

[23] OverSim: The overlay simulation framework. http://www.oversim.org/.

[24] Rendezvousproxy: Tutorial. http://ileech.sourceforge.net/index.php?content=RendezvousProxy-Tutorial.

[25] RightScale Cloud Computing Management Platform. http://www.rightscale.com/index.php.

[26] Seattle, Open Peer-to-Peer Computing. https://seattle.cs.washington.edu/html/.

[27] Secure Inter-Domain Routing (sidr). http://datatracker.ietf.org/wg/sidr/charter/.

[28] Simplify Media. http://www.simplifymedia.com/.

[29] Technical Q&A QA1311: Registering a Bonjour service multiple times. http://developer.apple.com/qa/qa2001/qa1311.html.

[30] The Bamboo Distributed Hash Table. http://bamboo-dht.org/.

[31] The SIP-Router Project. http://sip-router.org/.

[32] TR10: Software-Defined Networking. http://www.technologyreview.com/biotech/22120/.

[33] Ubuntu. http://www.ubuntu.com/.

[34] What happened on August 16, 2007. http://heartbeat.skype.com/2007/08/what_happened_on_august_16.html.

[35] What is the cloud broker service market forcast? http://www.quora.com/What-is-the-cloud-broker-service-market-forcast.

[36] XML-RPC for C and C++. http://xmlrpc-c.sourceforge.net/.

[37] XML-RPC home page. http://www.xmlrpc.com/.

[38] Xuggler. http://www.xuggle.com/xuggler/.

[39] Zero Configuration Networking. http://www.zeroconf.org/.

[40] Zero Configuration Networking (zeroconf) Working Group charter. http://www.ietf.org/html.charters/OLD/zeroconf-charter.html.

[41] Zeroconf-to-Zeroconf Toolkit (z2z). http://sourceforge.net/projects/z2z/.

[42] Princeton University Router Traffic Statistics. http://mrtg.net.princeton.edu/statistics/routers.html, 2010.

[43] K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *CoopIS'01: Proceedings of the 9th International Conference on Cooperative Information Systems*, pages 179–194, London, UK, 2001. Springer-Verlag.

[44] K. Aberer, A. Datta, M. Hauswirth, and R. Schmidt. Indexing data-oriented overlay networks. In *VLDB '05: Proceedings of the 31st international conference on very large databases*, pages 685–696. VLDB Endowment, 2005.

[45] D. Angluin, J. Aspnes, J. Chen, Y. Wu, and Y. Yin. Fast Construction of Overlay Networks. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium*

on parallelism in algorithms and architectures, pages 145–154, New York, NY, USA, 2005. ACM Press.

[46] M. S. Artigas, P. G. Lopez, J. P. Ahullo, and A. F. G. Skarmeta. Cyclone: A Novel Design Schema for Hierarchical DHTs. In *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*, pages 49–56, Washington, DC, USA, 2005. IEEE Computer Society.

[47] M. Arumaithurai, X. Fu, B. Schloer, and H. Tschofenig. Performance Study of the NSIS QoS-NSLP Protocol. In *GLOBECOM*, 2008.

[48] A. Banchs, W. Effelsberg, C. Tschudin, and V. Turau. Multicasting Multimedia Streams with Active Networks. In *Local Computer Networks, 1998. LCN '98. Proceedings., 23rd Annual Conference on*, pages 150 –159, oct 1998.

[49] S. A. Baset, J. Reich, J. Janak, P. Kasparek, V. Misra, D. Rubenstein, and H. Schulzrinne. How Green is IP-Telephony? In *The ACM SIGCOMM Workshop on Green Networking*, 2010.

[50] I. Baumgart, B. Heep, and S. Krause. OverSim: A Flexible Overlay Network Simulation Framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, pages 79–84, May 2007.

[51] I. Baumgart and S. Mies. S/Kademlia: A Practicable Approach Towards Secure Key-Based Routing. In *ICPADS '07: Proceedings of the 13th International Conference on Parallel and Distributed Systems*, pages 1–8, Washington, DC, USA, 2007. IEEE Computer Society.

[52] D. Bryan, P. Matthews, E. Shim, and D. Willis. Concepts and Terminology for Peer to Peer SIP. Internet draft, 2007.

[53] J. W. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *IPTPS*, pages 80–87, 2003.

[54] K. Calvert. Reflections on Network Architecture: an Active Networking Perspective. *ACM SIGCOMM Computer Communication Review*, 36(2):27–30, 2006.

[55] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. Scribe: A Large-Scale and Decentralized Application-Level Multicast Infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 20(8):1489–1499, 2002.

[56] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-Bandwidth Multicast in Cooperative Environments. In *SOSP '03: Proceedings of the nineteenth ACM symposium on operating systems principles*, pages 298–313, New York, NY, USA, 2003. ACM.

[57] S. Cheshire, B. Aboba, and E. Guttman. Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927, May 2005.

[58] S. Cheshire and M. Krochmal. DNS-Based Service Discovery. Internet draft, 2006.

[59] S. Cheshire and M. Krochmal. Multicast DNS. Internet draft, 2006.

[60] S. Cheshire and D. H. Steinberg. *Zero Configuration Networking: The Definitive Guide*, chapter 5. O'Reilly Media, Sebastopol, CA, 2005.

[61] S. da Silva, Y. Yemini, and D. Florissi. The NetScript Active Network System. *Selected Areas in Communications, IEEE Journal on*, 19(3):538 –551, mar 2001.

[62] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next-Generation Routers. *IEEE/ACM Transactions on Networking*, 8(1):2–15, 2000.

[63] D. Decasper and B. Plattner. DAN: Distributed Code Caching for Active Networks. In *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 609 –616 vol.2, mar-2 apr 1998.

[64] P. Faratin, D. Clark, P. Gilmore, S. Bauer, A. Berger, and W. Lehr. Complexity of Internet Interconnections: Technology, Incentives and Implications for Policy. In *TPRC*, 2007.

[65] M. Femminella, R. Francescangeli, G. Reali, J. W. Lee, and H. Schulzrinne. An Enabling Platform for Autonomic Management of the Future Internet. *Network, IEEE*, 25(6):24–32, 2011.

[66] M. Femminella, R. Francescangeli, G. Reali, J. W. Lee, W. Song, and H. Schulzrinne. Future Internet Autonomic Management Using NetServ. In *Demonstrations of the IEEE Conference on Local Computer Networks (LCN) (LCN-Demos 2011)*, pages 1081–1083, Bonn, Germany, Oct. 2011.

[67] M. Femminella, R. Francescangeli, G. Reali, and H. Schulzrinne. Gossip-based Signaling Dissemination Extension for Next Steps in Signaling. Technical Report, paper under review (available at: `http://conan.diei.unipg.it/pub/nsis-gossip.pdf`).

[68] M. Freedman, K. Lakshminarayanan, and D. Mazieres. OASIS: Anycast for Any Service. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation*, San Jose, CA, May 2006.

[69] X. Fu, H. Schulzrinne, H. Tschofenig, C. Dickmann, and D. Hogrefe. Overhead and Performance Study of the General Internet Signaling Transport (GIST) Protocol. *IEEE/ACM Transactions on Networking*, 17(1):158–171, 2009.

[70] P. Ganesan, K. Gummadi, and H. Garcia-Molina. Canon in G Major: Designing DHTs with Hierarchical Structure. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 263–272, Washington, DC, USA, 2004. IEEE Computer Society.

[71] L. Garces-Erice, E. W. Biersack, K. W. Ross, P. A. Felber, and G. Urvoy-Keller. Hierarchical P2P Systems. In *Proceedings of ACM/IFIP International Conference on Parallel and Distributed Computing (Euro-Par)*, Klagenfurt, Austria, 2003.

[72] R. Geambasu, A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: An active distributed key/value store. In *Proc. of OSDI*, 2010.

[73] L. Gong. Java 2 Platform Security Architecture. `http://download.oracle.com/javase/1.4.2/docs/guide/security/spec/security-spec.doc.html`.

[74] V. Gurbani, V. Hilt, and H. Schulzrinne. SIP Overload Control. Internet-Draft draft-ietf-soc-overload-control-01, 2011.

[75] J. M. Guzmán. Google Peering Policy. `http://lacnic.net/documentos/lacnicxi/presentaciones/Google-LACNIC-final-short.pdf`, 2008.

[76] R. Hancock, G. Karagiannis, J. Loughney, and S. Van den Bosch. Next Steps in Signaling (NSIS): Framework. RFC 4080, 2005.

[77] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, March 2003.

[78] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 86–93, New York, NY, USA, 1998. ACM.

[79] V. Hilt, E. Noel, C. Shen, and A. Abdelal. Design Considerations for SIP Overload Control. Internet-Draft draft-ietf-soc-overload-design-04, 2010.

[80] G. Hjalmtysson. The Pronto Platform: a Flexible Toolkit for Programming Networks Using a Commodity Operating System. In *OPENARCH*, 2000.

[81] V. Jacobson, D. Smetters, J. Thornton, M. Plass, N. Briggs, and R. Braynard. Networking Named Content. In *CoNeXT*, 2009.

[82] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations. In *Middleware'04: Proceedings of the 5th ACM/IFIP/USENIX international conference on middleware*, pages 79–98, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[83] M. Jelasity, A. Montresor, and O. Babaoglu. The Bootstrapping Service. In *ICDCSW '06: Proceedings of the 26th IEEE International Conference Workshops on Distributed Computing Systems*, Washington, DC, USA, 2006. IEEE Computer Society.

[84] B. Jennings, S. van der Meer, S. Balasubramaniam, D. Botvich, M. ó Foghlú, W. Donnelly, and J. Strassner. Towards Autonomic Management of Communications Networks. *Communications Magazine, IEEE*, 45(10):112–121, 2007.

[85] D. Katz. IP Router Alert Option. RFC 2113, 1997.

[86] R. Keller, L. Ruf, A. Guindehi, and B. Plattner. PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing. In *IWAN*, 2002.

[87] J. Kelly, W. Araujo, and K. Banerjee. Rapid Service Creation using the JUNOS SDK. *ACM SIGCOMM Computer Communication Review*, 40(1):56–60, 2010.

[88] W. Kho, S. A. Baset, and H. Schulzrinne. Skype Relay Calls: Measurements and Experiments. *Computer Communications Workshops, 2008. INFOCOM. IEEE Conference on*, pages 1–6, April 2008.

[89] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[90] C. Law and K.-Y. Siu. Distributed Construction of Random Expander Networks. *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, 3:2133–2143, April 2003.

[91] J. W. Lee, R. Francescangeli, J. Janak, S. Srinivasan, S. Baset, H. Schulzrinne, Z. Despotovic, and W. Kellerer. NetServ: Active Networking 2.0. In *Communications Workshops (ICC), 2011 IEEE International Conference on*, pages 1–6. IEEE, 2011.

[92] J. W. Lee, R. Francescangeli, W. Song, J. Janak, S. Srinivasan, M. Kester, S. A. Baset, E. Liu, H. Schulzrinne, V. Hilt, Z. Despotovic, and W. Kellerer. NetServ Framework Design and Implementation 1.0. Technical Report cucs-016-11, Columbia University, May 2011.

[93] J. W. Lee, H. Schulzrinne, W. Kellerer, and Z. Despotovic. z2z: Discovering Zeroconf Services Beyond Local Link. In *Globecom Workshops, 2007 IEEE*, pages 1–7. IEEE, 2007.

[94] J. W. Lee, H. Schulzrinne, W. Kellerer, and Z. Despotovic. mDHT: Multicast-augmented DHT Architecture for High Availability and Immunity to Churn. In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1–5. IEEE, 2009.

[95] J. W. Lee, H. Schulzrinne, W. Kellerer, and Z. Despotovic. 0 to 10k in 20 seconds: Bootstrapping Large-scale DHT networks. In *Communications (ICC), 2011 IEEE International Conference on*, pages 1–6. IEEE, 2011.

[96] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, 2010.

[97] B. Manning and P. Vixie. Operational Criteria for Root Name Servers. RFC 2010, October 1996.

[98] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *International workshop on Peer-To-Peer Systems (IPTPS)*, pages 53–65, 2002.

[99] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[100] A. Montresor, M. Jelasity, and O. Babaoglu. Chord on Demand. In *P2P'05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*, pages 87–94, Washington, DC, USA, 2005. IEEE Computer Society.

[101] G. Pandurangan, P. Raghavan, and E. Upfal. Building Low-Diameter Peer-to-Peer Networks. *IEEE Journal on Selected Areas in Communications*, 21(6):995–1002, Aug. 2003.

[102] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences Building PlanetLab. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 351–366, Berkeley, CA, USA, 2006. USENIX Association.

[103] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918, Feb. 1996.

[104] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and Its Uses, 2005.

[105] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. Internet-Draft draft-ietf-mmusic-ice-19, 2007.

[106] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, 2002.

[107] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[108] J. Salim, R. Olsson, and A. Kuznetsov. Beyond Softnet. In *The 5th Annual Linux Showcase and Conference*, 2001.

[109] S. Schmid, J. Finney, A. Scott, and W. Shepherd. Component-based Active Network Architecture. In *ISCC*, 2001.

[110] H. Schulzrinne and R. Hancock. GIST: General Internet Signalling Transport. RFC 5971, 2010.

[111] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge. Smart Packets: Applying Active Networks to Network Management. *ACM Trans. Comput. Syst.*, 18:67–88, February 2000.

[112] S. Srinivasan, J. W. Lee, E. Liu, M. Kester, H. Schulzrinne, V. Hilt, S. Seetharaman, and A. Khan. Netserv: Dynamically Deploying In-network Services. In *Proceedings of the 2009 workshop on Re-architecting the internet*, pages 37–42. ACM, 2009.

[113] N. Steinleitner, H. Peters, and X. Fu. Implementation and Performance Study of a New NAT/Firewall Signaling Protocol. In *ICDCS Workshops*, 2006.

[114] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[115] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *IEEE Transactions on Networking*, 11, February 2003.

[116] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *ACM SIGCOMM Computer Communication Review*, 26(2):5–17, 1996.

[117] J. S. Turner, P. Crowley, J. DeHart, A. Freestone, B. Heller, F. Kuhns, S. Kumar, J. Lockwood, J. Lu, M. Wilson, C. Wiseman, and D. Zar. Supercharging Planetlab: A High Performance, Multi-Application, Overlay Network Platform. *ACM SIGCOMM Computer Communication Review*, 37(4):85–96, 2007.

[118] V. Vishnumurthy and P. Francis. On Heterogeneous Overlay Construction and Random Node Selection in Unstructured P2P Networks. *INFOCOM 2006. The 25th IEEE International Conference on Computer Communications*, pages 1–12, April 2006.

[119] S. Voulgaris and M. V. Steen. An Epidemic Protocol for Managing Routing Tables in Very Large Peer-to-Peer Networks. In *Proceedings of the 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, (DSOM 2003)*, pages 41–54. Springer, 2003.

[120] D. Wessels and M. Fomenkov. Wow, That's a Lot of Packets. In *Proceedings of Passive and Active Measurement Workshop (PAM)*, April 2003.

[121] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE OPENARCH 98*, 1998.

[122] D. J. Wetherall and D. L. Tennenhouse. The ACTIVE IP Option. In *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, EW 7, pages 33–40, New York, NY, USA, 1996. ACM.

[123] Y. Yemini and S. D. Silva. Towards Programmable Networks. In *in IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, 1996.

Part V

Appendices

Appendix A

Making Real-world Impact: NetServ on GENI

GENI [12] is a federation of many existing network testbeds under a common management framework. GENI is comprised of a diverse set of platform resources, which are shared among many experimenters.

We demonstrated NetServ, using two of our sample applications, at the plenary session of the 9th GENI Engineering Conference (GEC9).[1] Figure A.1 shows the screenshots from the ActiveCDN demo. The ActiveCDN module performed custom processing on cached content, watermarking a video stream with local weather information.

[1]The 14-minute demo video is at http://vimeo.com/16474575.

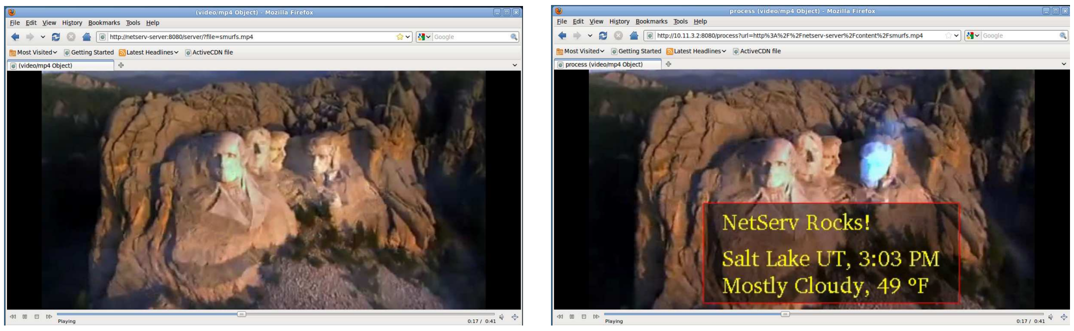

(video/mp4 Object) - Mozilla Firefox
http://netserv-server:8080/server/?file=smurfs.mp4
process (video/mp4 Object) - Mozilla Firefox
http://10.11.3.2:8080/process?url=http%3A%2F%2Fnetserv-server%2Fcontent%2Fsmurfs.mp4
NetServ Rocks!
Salt Lake UT, 3:03 PM
Mostly Cloudy, 49 °F

Figure A.1: ActiveCDN demo.

Appendix B

Autonomic Management Using NetServ

This appendix describes how NetServ can be used to implement autonomic network management. We demonstrated our idea at the 36th IEEE Conference on Local Computer Networks (LCN) [66]. The implementation was based on our earlier proposal [65].

B.1 Introduction

This demo proposal shows how the NetServ platform can be used for implementing autonomic management architectures for the Future Internet. This translates to capabilities of automatically deploying, configuring, and removing at runtime both Policy Decision Point (PDP) and Policy Enforcement Point (PEP) modules on network nodes, in order to provide network management with effective autonomic capabilities. In fact, the usage of programmable nodes able to host any service, made up by combining inferential, decisional, monitoring, and actuator modules, represents a powerful instrument to implement autonomic network management functions.

We present a novel solution for deploying autonomic network and service management architectures. We do not aim at introducing new management paradigms, but rather to increase the effectiveness of the existing ones by resorting to the potential provided by the NetServ project, which is a framework designed to deploy and execute networked services

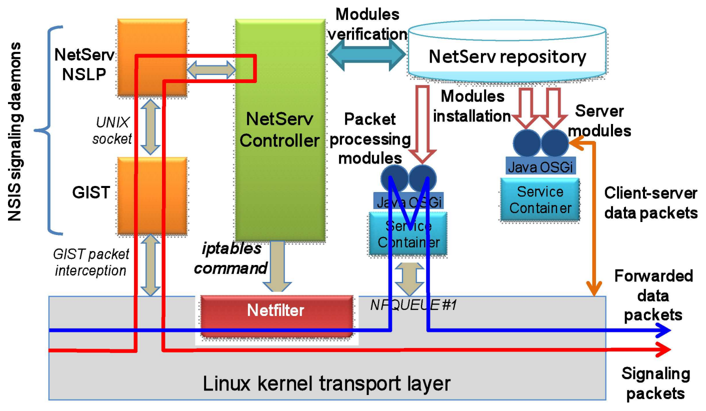


Figure B.1: NetServ node internal architecture.

at runtime over programmable routers. The use of the NetServ capabilities in the management planes represents a step forward the state of the art, since it increases the flexibility of management solutions, their dynamic response to event requiring management actions, decreases the relevant traffic, and decreases the response time. In order to show the effectiveness of the proposed solution, we show a case study which highlights how NetServ allows deploying self-protecting network functions. In this demo we show how the NetServ-based management architecture is able to counteract a DoS attack by selectively deploying monitoring and actuator modules at runtime.

B.2 Autonomic Management Architecture

The key element of our management architecture is the NetServ Autonomic Management Element (NAME). It is inspired by the FOCALE architecture shown in [84], which has been mapped into the service deployment architecture shown in Figure B.1, more details can be found in [65]. This decision follows from the consideration that the FOCALE architecture already includes most of enabling mechanisms for autonomic network management and its modularity allows integrating the unique features of NetServ that, we believe, may introduce significant dynamics in network and service management. The NetServ additional functions are included in this architecture by implementing it as a NetServ service, and by also introducing the PEP (policy enforcement point) deployment module, which can deploy management programs over the selected NetServ managed resources at runtime. These

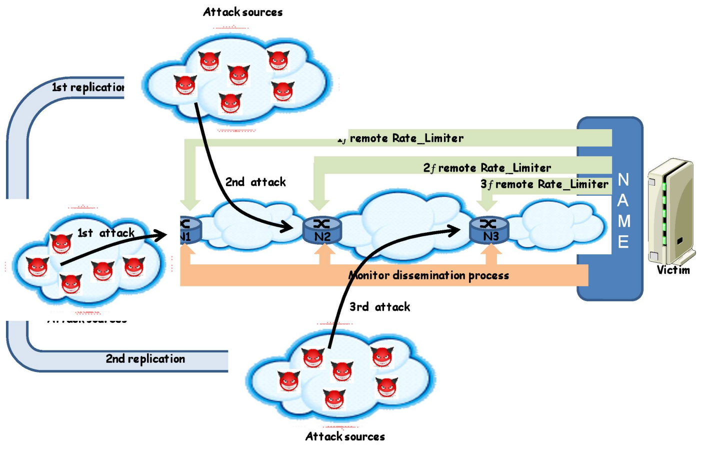


Figure B.2: Network topology for the DoS scenario.

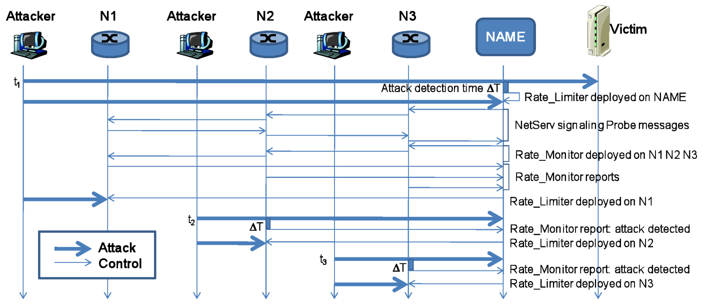

Attacker
N1
Attacker
N2
Attacker
N3
NAME
Victim
t1
Attack detection time ΔT
Rate_Limiter deployed on NAME
NetServ signaling Probe messages
Rate_Monitor deployed on N1 N2 N3
Rate_Monitor reports
Rate_Limiter deployed on N1
t2
ΔT
Rate_Monitor report: attack detected
Rate_Limiter deployed on N2
t3
ΔT
Rate_Monitor report: attack detected
Rate_Limiter deployed on N3
Attack
Control

Figure B.3: Signaling flow in the GENI experiment.

programs are stored in the NetServ repository.

B.3 Demo Scenario

This section describes the autonomic management scenario we show at the LCN demo session. The demo highlights the effectiveness of NAME in protecting a network resource from a DoS attack. The attack shown in the demo is a example of a generic DoS attack, but it is sufficiently structured to show the NetServ dynamic properties brought to the management architecture.

Figure B.2 shows the network topology in this experiment, which is deployed on GENI [12]. The victim, an application server, is protected by a NAME instance. The attack is a classic DoS flooding attack, performed by a number of hosts in different networks.

A lightweight NetServ service module, called Rate_Monitor, is executed in the NAME itself and evaluates the rate of incoming traffic and notifies the PDP module. Figure B.3 shows that, when the attack starts at time t1, the local Rate_Monitor notifies the NAME engine the value of the incoming rate above the alarm threshold. This information reveals that the network has entered an unacceptable state. The set of actions deemed necessary for leading the system to an acceptable state are:

- the retrieval of a Rate_Limiter module from the NetServ repository and its deployment on the local interface, in order to protect the victim against the overwhelming service requests;
- the deployment of a number of Rate_Monitor modules in the NetServ nodes all around the NAME instance, by means of epidemic signalling or directory service, so as to identify the incoming attack directions and deploy additional Rate_limiter modules on nodes where the observed value of the incoming service requests are above a given threshold.

The objective of the second action is twofold. First, any attack direction can be identified and the attack can be faced upstream. Second, in this way we relieve the network from the traffic generated by the attackers (denial of network service).

In order to execute the second action, the NAME instance starts sending NetServ PROBE messages towards all directions from itself up to three IP hops, so as to identify the NetServ nodes able to host and execute an instance of the Rate_Monitor module, at shown in Figure B.3. Then, by using the NetServ deployment signaling, the NAME engine deploys a Rate_Monitor module on the selected nodes, which immediately start reporting incoming rate values.

Note that in this phase the application server is protected by the Rate_Limiter instance executed by the NAME itself. On the basis of reported values, which are the portion of interest of the new system state, the Action Planner of the NAME identifies the node N1 shown in Figure B.3 as the best candidate to deploy a remote Rate_Limiter module, since it is the most distant node (in terms of IP hops) from the NAME with an incoming rate above the alarm threshold. Thus, by using the NetServ signaling, the NAME can instantiate the

Rate_Limiter in N1. The Rate_Limiter module interacts with the NAME, which receives reports of all deployed Rate_Monitor modules, and changes the acceptable incoming rate threshold dynamically, depending on the number and frequency of detected requests. In this way, a further control loop is created so that each management action enforced by the NAME is dynamically adapted to possible context and state changes.

At time t2, the attacker adds additional sources of DoS packets in other networks, thus bypassing the deployed shield. Nevertheless, since the NAME instance has been executing the monitor and rate limiter module since attack beginning, it can both protect the server and argue that the previous remote counteracting action has been bypassed. If the previously deployed Rate_Monitor modules are still active, some of them start reporting values of the observed incoming rate beyond acceptable values. This context information allows the NAME to identify the NetServ node N2 as the best candidate to deploy another remote instance of the Rate_Limiter module. If the lifetime of the previously deployed Rate_Monitor modules has expired, they are redeployed.

Finally, the attacker starts a further attack session from another network at time t3. The self-protecting procedure is repeated again, thus deploying a further instance of the Rate_Limiter on N3 that decreases the service request rate once again to a value as close as possible to the target value. When the attack ends, all the monitor and rate limiter instances are no longer refreshed. Hence, they are automatically removed, without any additional signaling.

In order to actually estimate the end of attack condition at the NAME, the remote monitor modules track both forwarded and dropped service requests, and report back the relevant statistics.