NetServ: Reviving Active Networks

Jae Woo Lee[1], Roberto Francescangeli[2], Wonsang Song[1], Emanuele Maccherani[2], Jan Janak[1], Suman Srinivasan[1], Michael S. Kester[1], Salman A. Baset[3], Henning Schulzrinne[1]
[1]*Columbia University* [2]*Universita degli Studi di Perugia* [3]*IBM Research*

Abstract

In 1996, Tennenhouse and Wetherall proposed active networks, where users can inject code modules into network nodes. The proposal sparked intense debate and follow-on research, but ultimately failed to win over the networking community. Fifteen years later, the problems that motivated the active networks proposal persist.

We call for a revival of active networks. We present NetServ, 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.

We make the following contributions. We present a hybrid approach to active networking, which combines the best qualities from the two extreme approaches–integrated and discrete. We built a working system that strikes the right balance between security and performance by leveraging current technologies. We suggest an economic model based on NetServ between content providers and ISPs. We built four applications to illustrate the model.

1 Introduction

Tennenhouse and Wetherall presented the vision of an active network architecture in their seminal 1996 paper [28]. 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 [21].

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. 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.

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

We call for a revival of active networks. Active networking was ahead of its time when it was proposed, but we believe its time has arrived. We claim that 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.

We present NetServ, 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. Our contribution can be summarized in terms of resolving the following three conflicts:

Integrated vs. Discrete We present a hybrid approach that combines the best qualities from the two extreme approaches to active networking.

Security vs. Performance We built a working system that strikes the right balance between security and performance by leveraging the current technologies.

Content provider vs. ISPs We suggest an economic model on top of the newly available in-network resources between content providers and ISPs. We built four applications to illustrate the model.

We elaborate on the three conflicts in Section 2. Section 3 describes the resulting architecture meeting the goals and challenges. Section 4 describes our implementation on Linux. In Section 5, we expand on the security issues. In Section 6, 7, 8, we describe our four sample applications, talk about our activities on GENI, and discuss some issues regarding the deployment of the applications. In Section 9, we evaluate our Linux implementation. Section 10 describes the OpenFlow extension of NetServ, which addresses the performance limitation of our Linux implementation. Sections 11 discusses related work. We conclude in Section 12.

2 Goals and Challenges

2.1 Integrated vs. Discrete

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 [11, 18, 20, 22] 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, 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.

2.2 Security vs. 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 discuss resource control and isolation further in Section 5.

Running service modules in user space, and in Java on top of that, inevitably raises the eyebrows of performance-minded critics. In Section 9, we explore 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.

Our real defense against performance-related criticisms is the multi-box lateral expansion of NetServ using the OpenFlow [24] forwarding engine, described in Section 10. 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.

2.3 Content providers vs. ISPs

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.

The network edge belongs to a particular type of ISPs, often called *eyeball* ISPs. As opposed to *content* ISPs who provide hosting and connectivity to content providers, *eyeball* ISPs provide last-mile connectivity to end users. It has been noted that eyeball ISPs wield increased bargaining power in peering agreements because they *own* the eyeballs [12]. We argue that their presence at the network edge is another powerful asset. The edge routers of eyeball ISPs, due to their proximity to end users, are excellent candidates for NetServ nodes that can host cached content and custom service modules for content and application providers. For content providers, a large number of NetServ nodes spread out at the network edge would create an attractive alternative to CDNs. For eyeball ISPs, a cluster of NetServ nodes at the network edge provide another source of revenue.

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.

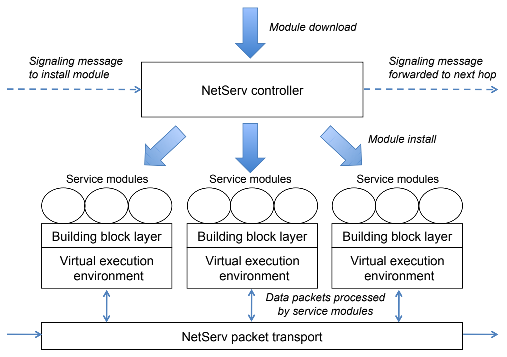

Module download
Signaling message to install module
NetServ controller
Signaling message forwarded to next hop
Module install
Service modules
Service modules
Service modules
Building block layer
Building block layer
Building block layer
Virtual execution environment
Virtual execution environment
Virtual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

Figure 1: NetServ node architecture.

In Section 6, we describe four sample applications that we have built in order to illustrate this opportunity. Building those applications had an interesting consequence on the NetServ architecture. Some of our application scenarios require that a user service module running on NetServ not only perform packet processing, but also provide traditional end-to-end network services. The end result is that a NetServ application module can be 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 combination of both. We consider this trait yet another way we conform to the active networking vision, or possibly even extend it, as we aim to eliminate the distinction between routers and servers. User code simply runs everywhere.

3 Architecture Overview

Figure 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 [10]. 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 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.

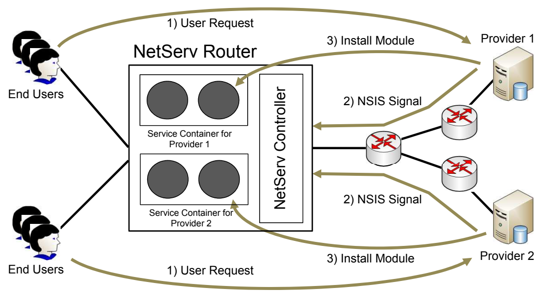


Figure 2: Deploying modules on a NetServ node.

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.

Figure 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. We will describe four specific examples of this application scenario in Section 6.

4 Implementation

We implemented the NetServ architecture on Linux. We released source code[2] in conjunction with a NetServ tutorial we gave at the 11th GENI Engineering Conference (GEC11). We will continue to release new versions of our software and give NetServ tutorials at future GECs.

Figure 3 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,

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

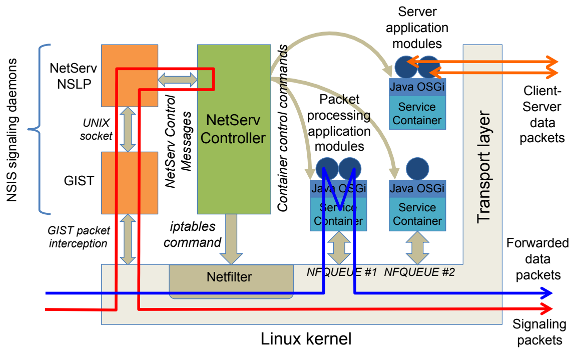


Figure 3: Linux implementation.

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 [4]. 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.

The two circles on the upper-right service container are server application modules, sending and receiving normal TCP/IP packets labeled "client-server data packets." The two circles on the lower-left container are packet processing application modules. The arrow labeled "forwarded data packets" shows how an incoming packet is routed from the kernel to a user space container process. The packet then visits the two packet processing modules in turn before being pushed back to the kernel.

We provide a detailed description of each part of Figure 3 in the following subsections.

4.1 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 [17], an IETF standard for signaling. NSIS consists of two layers: a generic *signaling transport* layer and an application-specific *signaling application* layer.

The two boxes in Figure 3, labeled "GIST" and "NetServ NSLP," represent the two NSIS signaling layers used in a NetServ node. GIST, the General Internet Signalling Transport protocol [27], 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.

```
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 4: A SETUP message.

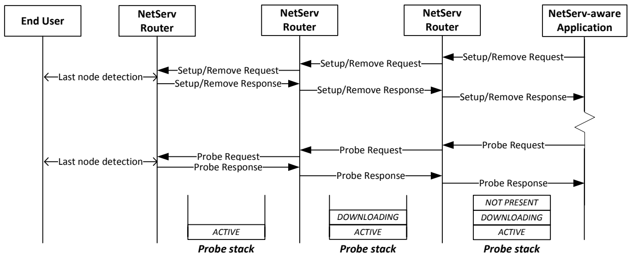


Figure 5: Request and response exchange.

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 [3].

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 4 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. Our companion technical report [23] contains a listing of the currently supported header fields in request messages.

Figure 5 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 alone 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 5 shows a response to a module status probe being collected in a response stack.

4.2 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 preforks 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.

4.3 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 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 [22] and the OpenFlow [24] switch. We are also porting NetServ to Juniper routers using the JUNOS SDK [21]. The Click implementation and our approach for Juniper port are described in [23]. The OpenFlow extension for NetServ is described in detail in Section 10.

4.4 Service Container and Modules

Service containers are user space processes that run modules written in Java. Figure 6 shows our current implementation. The service container process can optionally

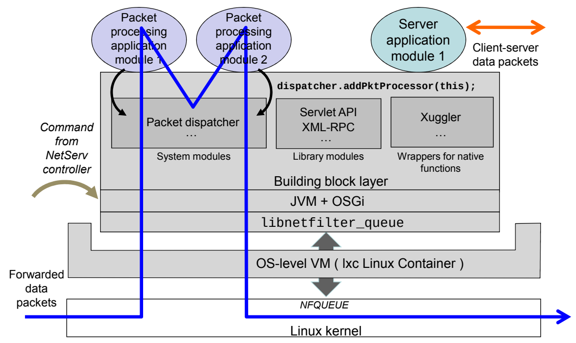

Packet processing application module 1
Packet processing application module 2
Server application module 1
Client-server data packets
dispatcher.addPktProcessor(this);
Packet dispatcher
...
System modules
Servlet API
XML-RPC
...
Library modules
Xuggler
...
Wrappers for native functions
Building block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

Figure 6: User space service container process.

be run within lxc [2], an OS-level virtualization technology included in the Linux kernel. We defer the discussion of lxc to Section 5.

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 6.1 requires Xuggler [7], a Java wrapper for the FFmpeg video 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.

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. 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.

5 Security

In this section, 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.

5.1 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 [14]. 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[3], 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.

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`

[3]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.

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.

5.2 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 [6]. 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.

6 NetServ Applications

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

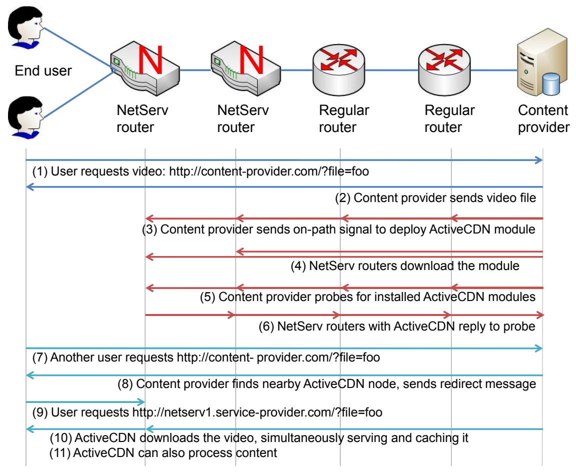

End user
NetServ router
NetServ router
Regular router
Regular router
Content provider
(1) User requests video: http://content-provider.com/?file=foo
(2) Content provider sends video file
(3) Content provider sends on-path signal to deploy ActiveCDN module
(4) NetServ routers download the module
(5) Content 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: How ActiveCDN works.

6.1 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 different 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 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. 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

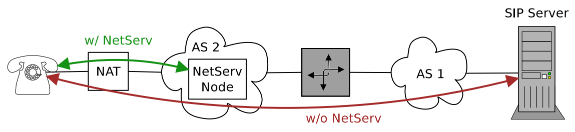


Figure 8: Operation of KeepAlive Responder.

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 a while. The process repeats when new requests are made from the same region.

6.2 KeepAlive Responder

The ubiquitous presence of Network Address Translators (NATs) poses a challenge to communication services based on Session Initiation Protocol (SIP) [25]. 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 [9] 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. A surprising fact is that the keep-alive traffic can be a bottleneck in scaling a SIP server to a large number of users [9].

Figure 8 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. 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.

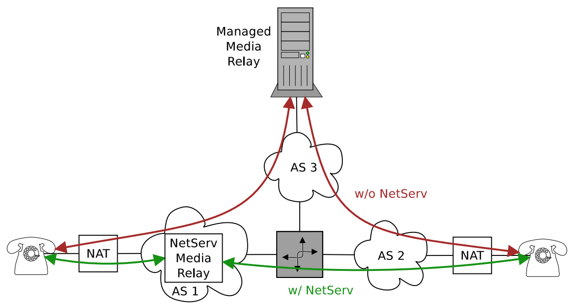


Figure 9: Operation of NetServ Media Relay.

6.3 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 9 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.

6.4 Overload Control

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 [15]. Figure 10 illustrates the scenario. The SIP server under load, referred to as the Receiving Entity (RE), periodically monitors its

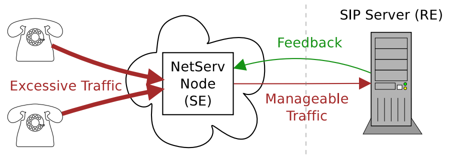


Figure 10: NetServ as SIP overload protection.

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. When the load on the 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 NetServ on GENI

GENI [1] 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.

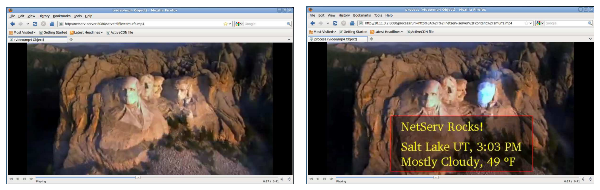


Figure 11: ActiveCDN demo.

We demonstrated NetServ, using two of our sample applications, at the plenary session of the 9th GENI Engineering Conference (GEC9).[4] Figure 11 shows the screenshots from the ActiveCDN demo. The ActiveCDN module performed custom processing on cached content, watermarking a video stream with local weather information of Salt Lake City where the module was installed. We hold NetServ tutorials at GECs since GEC11.

8 Discussion

Reverse data path

The descriptions of the NetServ applications in Section 6 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

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

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.

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 [13]. 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.

9 Evaluation

In this section, 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.

9.1 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) in the Linux kernel.

9.2 Results

We measured the MLFFRs of six different configurations of the router. 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 environment.

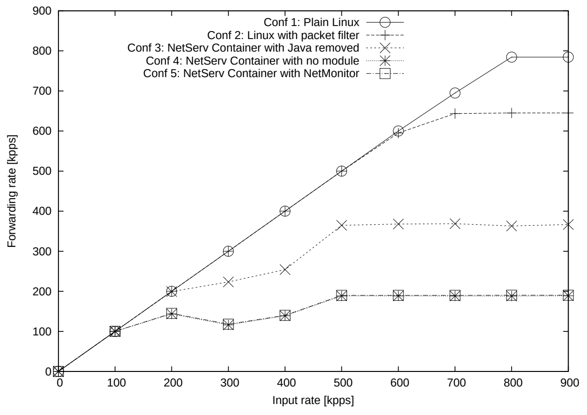
The packet path for configuration 4 includes the full NetServ container, which includes a Java execution environment. However, no application module is added to the NetServ container.

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.

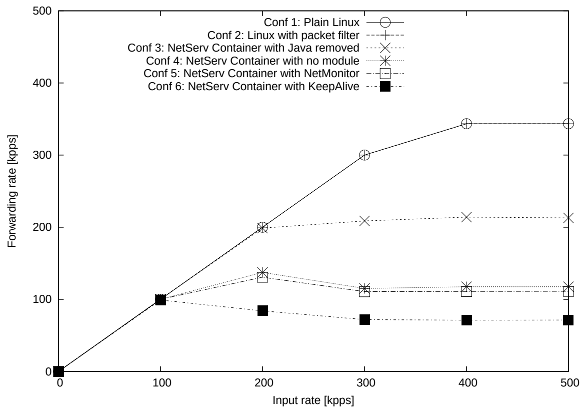
Configuration 6 replaces NetMonitor with the KeepAlive module described in Section 6.2. KeepAlive examines incoming packets for SIP NOTIFY requests with the keep-alive Event header and swaps the source and destination IP addresses. For the measurement, we disabled the address swapping so that packets can be forward to the receiver. This test represents a NetServ router running a real-world application module.

Figure 12(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.

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.



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



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

Figure 12: Forwarding rates of the router with different configurations.

The dips in the curves for configurations 3 through 5 are the result of switching between the interrupt and polling modes of the NAPI network driver in Linux. See our technical report [23] for details.

Figure 12(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.

Figure 13 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

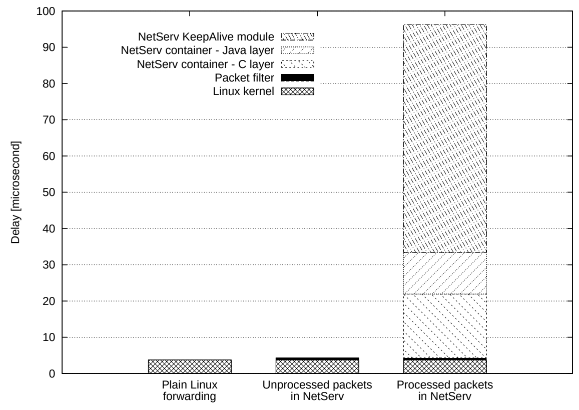


Figure 13: Microbenchmark.

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 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 [8]. 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. And, as we argued before, the OpenFlow extension in Section 10 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 14 shows the total throughput and memory consumption as we increase

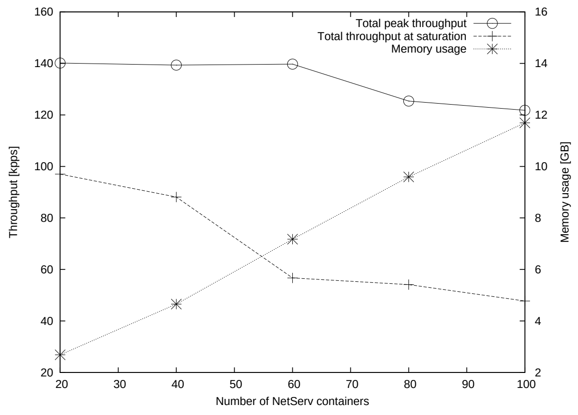


Figure 14: NetServ node with many containers.

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 14 shows that a NetServ node scales reasonably well as we increase the number of containers in it.

10 OpenFlow Extension

The Linux-based implementation that we described in the previous section 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 will show in Section 9. 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.

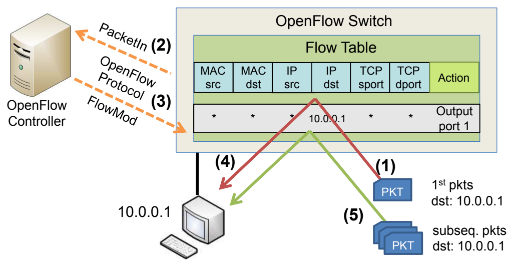


Figure 15: How OpenFlow works.

The OpenFlow programmable switch architecture provides exactly the capabilities that we need. In this section, we briefly explain the OpenFlow architecture, and describe our prototype implementation of the OpenFlow extension of NetServ. At the time of this writing, the prototype is at an early stage where we have proved that our approach works. We expect to have a beta release by the time this paper is published.

10.1 OpenFlow Overview

An OpenFlow switch [24] 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 15 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

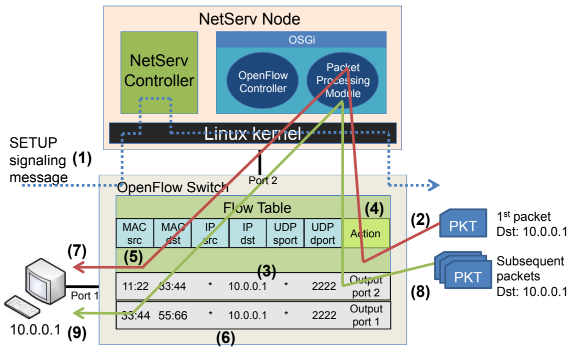


Figure 16: NetServ with OpenFlow extension.

switch at line rate without incurring the overhead of making a round trip to OFC.

10.2 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 [24] 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 16 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 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 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.

11 Related Work

Many earlier programmable routers focused on providing modularity without sacrificing forwarding performance, which meant installing modules in kernel space. Router Plugins [11], Click [22], PromethOS [20], and Pronto [18] followed this approach. As we noted before, NetServ runs modules in user space.

LARA++ [26] 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.

The Million Node GENI project [5], 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.

Google Global Cache (GGC) [16] 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) [19] 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.

12 Conclusion

We call for a revival of active networks. We present NetServ, 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.

We present a hybrid approach to active networking, which combines the best qualities from the two extreme approaches–integrated and discrete. We built a working system that strikes the right balance between security and performance by leveraging current technologies. We suggest an economic model based on NetServ between content providers and ISPs. We built four applications to illustrate the model.

Acknowledgements

The authors would like to thank Roxana Geambasu, Volker Hilt, Zoran Despotovic, Wolfgang Kellerer, Mauro Femminella and Gianluca Reali for feedback and discussion. We also thank Eric Liu, Raynald Seydoux, Abhishek Srivastava and Nathan Miller for their assistance with implementation. This work was supported in part by NSF grant NSF-CNS #0831912.

References

[1] GENI. `http://www.geni.net/`.

[2] lxc Linux Containers. `http://lxc.sourceforge.net/`.

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

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

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

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

[7] Xuggler. `http://www.xuggle.com/xuggler/`.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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`).

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

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

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

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

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

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

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

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

[22] 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.

[23] 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.

[24] 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.

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

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

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

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