NetServ: Activating the Network Edge

Jae Woo Lee[1], Roberto Francescangeli[2], Wonsang Song[1], Jan Janak[1],
Suman Srinivasan[1], Michael S. Kester[3], Salman A. Baset[4], Eric Liu[3],
Henning Schulzrinne[1], Volker Hilt[5], Zoran Despotovic[6], Wolfgang Kellerer[6]

[1]Columbia University {jae,wonsang,janakj,sumans,hgs}@cs.columbia.edu
[2]Universita degli Studi di Perugia roberto.francescangeli@diei.unipg.it
[3]Columbia University {msk2117,ewl2113}@columbia.edu
[4]IBM Research sabaset@us.ibm.com
[5]Bell Labs/Alcatel-Lucent volkerh@alcatel-lucent.com
[6]DoCoMo Communications Laboratories Europe {despotovic,kellerer}@docomolab-euro.com

Paper #1569469535, 12 pages

ABSTRACT

Eyeball ISPs today are under-utilizing an important asset: edge routers. We present NetServ, a programmable node architecture aimed at turning edge routers into service hosting platforms. This allows ISPs to allocate router resources to content publishers and application service providers motivated to deploy content and services at the network edge.

Unlike previous programmable router proposals which focused on customizing features of a router, NetServ focuses on deploying content and services across ownership boundaries. All our design decisions reflect this change in focus. We set three main design goals: a wide-area deployment, a multi-user execution environment, and a clear economic incentive. Towards these goals, our prototype uses NSIS signaling for deployment, runs application modules in isolated user space containers, and includes four sample applications demonstrating economic benefits.

1. INTRODUCTION

There are two types of Internet Service Providers (ISPs): content and eyeball. Content ISPs provide hosting and connectivity for content publishers, and 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 [13]. Eyeball ISPs have another unique asset, edge routers, which they are currently under-utilizing. This missed opportunity motivates our work.

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]. 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 the paper).

Programmable routers [12, 18, 21, 23], traditionally software routers based on commodity operating systems or more recently commercial routers with an SDK [22], 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 usu-

[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 chose “publishers” to clearly distinguish them from Internet Service Providers.

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

ally 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.

We propose NetServ, a programmable node architecture aimed at 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 design goals:

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.

In this paper, we describe our current prototype and how it reflects the design goals. The prototype is being actively developed, and does not yet fully implement the envisioned NetServ architecture. Nevertheless, we believe that describing the prototype is the best way to communicate our design. Therefore, in Section 2, we will interleave the descriptions of the prototype and the architectural design. We will make it clear, however, which part is in running code and which part is design only.

In Section 3, we discuss security issues. We describe four sample applications in Section 4. In Section 5, we show our evaluation results. Sections 6 and 7 discuss related work and future work, respectively. Lastly, we conclude in Section 8.

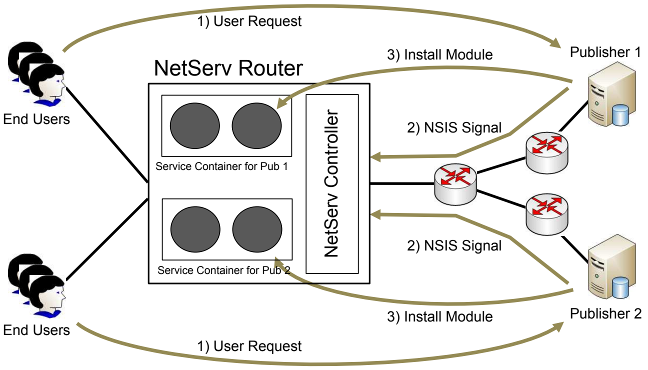


Figure 1: Deploying modules on a NetServ node.

2. NETSERV ARCHITECTURE

Figure 1 gives an overview of how content publishers can deploy application modules to a NetServ router. End user requests received by a content publisher's server will trigger signaling from the server. As a signaling message travels towards an end user, it passes through routers between the publisher and the user. A signaling message is contained in a UDP packet, so regular IP routers simply forward it towards the destination. When the message passes through a NetServ router, however, it causes the NetServ router to download and install an application module from the content publisher. The exact condition to trigger signaling and what the module does once installed will depend on the application. For example, a content publisher 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 see specific application examples in Section 4.

Figure 2 describes the architecture of our current prototype which is based on Linux. The arrow at the bottom labeled "signaling packets" indicates the path a signaling packet takes through this router. It is intercepted by the signaling daemons, which unpack the signaling packet and pass the contained NetServ Control 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 Java Virtual Machines (JVMs). Each container holds one or more application modules created by a single publisher. The JVMs run the OSGi module framework [6]. 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.

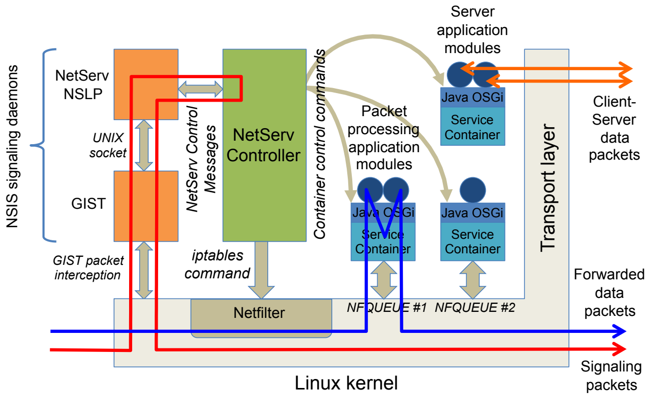


Figure 2: NetServ node prototype.

Our choice of Java for publisher-created applications reflects our design goals. Java's binary-level portability, extensive libraries, and popularity support our goal of wide-area deployment. The resource control and isolation features of Java 2 Security and the OSGi framework support our goal of a multi-user execution environment. Placing Java in a router is unconventional and may raise concerns about performance. However, our evaluation results in Section 5 mitigate this concern.

There are two types of application modules shown in Figure 2. *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. As we will see in Section 4, the applications deployed by content publishers typically include both functionalities.

We provide a detailed description of each part of Figure 2 in the following subsections.

2.1 Signaling

In order to satisfy our goal of wide-area deployment 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 2, labeled "GIST" and "NetServ NSLP," represent the two signaling layers used in a NetServ node. GIST, the General Internet Signalling Transport protocol [30], is a widely used implementation of NTLP, the transport layer of NSIS. NetServ NSLP is the NetServ-specific implementation of NSLP, the application layer of NSIS. The NetServ NSLP daemon receives signaling messages from the GIST daemon through a UNIX domain socket. The NetServ NSLP daemon then passes the NetServ Control Message (NCM) contained in the signal to the NetServ controller. The current implementation of the NetServ signaling daemons is based on NSIS-ka [5].

GIST is a soft state protocol that discovers peers and maintains associations in the background, transparently providing this service to the NSLP layer. 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) [20]. 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.

Publishers 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 backoff up to a predefined threshold. Depending on the threshold 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.

An NCM, in binary format, is included in the payload of an NSIS signaling message. An NCM is converted into an HTTP-like text format when it is delivered from the NetServ NSLP daemon to the NetServ controller. This decouples the rest of the NetServ node from the signaling daemons which allows debugging and testing of the NetServ core without signaling.

There are two kinds of NetServ signaling messages: requests and responses. Typically, a publisher's server sends a request, an on-path signaling message containing an NCM, 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 poli-

```
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-publisher.com/modules/netmonitor.jar
signature:4Z+HvDEm2WhHJrg9UKovwmMutxGibsA71FTMFykVa0Y\xGclG8o=
<blank line>
```

Figure 3: A SETUP message.

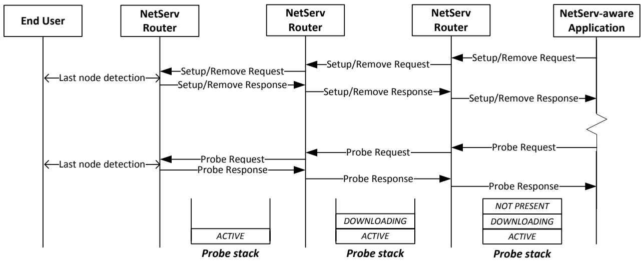


Figure 4: Request and response exchange.

cies. Figure 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. Our companion technical report [24] contains a listing of the currently supported header fields in request messages.

Figure 4 shows how response messages are generated at the last node and are 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 PROBE requester. Figure 4 shows a response to a module status probe being collected in a response stack.

2.2 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.

2.3 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 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 implemented an alternate forwarding plane for NetServ using a Click router [23], are currently developing one based on OpenFlow [26], and plan to port NetServ to Juniper routers using the JUNOS SDK [22]. The Click implementation and the plan for JUNOS are described in [24].

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.

2.4 Service Container and Modules

Service containers are user space processes that run modules written in Java. Figure 5 shows our current implementation. The service container process can optionally be run within lxc [4], an OS-level virtualization technology included in the Linux kernel. We defer the discussion of lxc to Section 3.

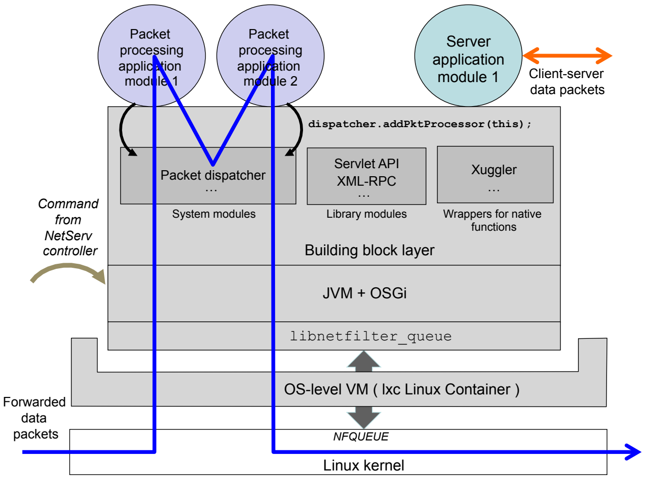

Packet processing application module 1
Packet processing application module 2
Server application module 1
Client-server data packets
dispatcher.addPktProcessor(this);
Packet dispatcher
...
Servlet API
XML-RPC
...
Xuggler
...
System modules
Library modules
Wrappers for native functions
Building block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

Figure 5: User space service container process.

When the container process starts, the container creates a Java Virtual Machine (JVM) using the invocation API, which is a part of the Java Native Interface (JNI), 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 4.1 requires Xuggler [9], a Java wrapper for the FFmpeg [2] 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 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.

3. SECURITY

3.1 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, the standard Linux resource control and isolation mechanisms apply: the scheduling priority, the *nice* value, can be lowered; the usage of system resources like memory and file descriptors can be limited using `setrlimit()`; disk quota can be set; and file system access can be restricted using `chroot`.

We control the application modules further using Java 2 Security [14]. Java 2 Security 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.

OSGi provides namespace isolation between bundles using a custom class loader. The only method of interbundle 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 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`

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

3.2 Authentication

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 publisher. When a publisher 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 publisher's public key. A clearinghouse will manage user credentials and settle payments between publishers 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 publisher 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 [8]. 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 publisher 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 publisher 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.

4. NETSERV APPLICATIONS

We advocate NetServ as a platform that enables publishers 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 a clear economic benefit for both parties.

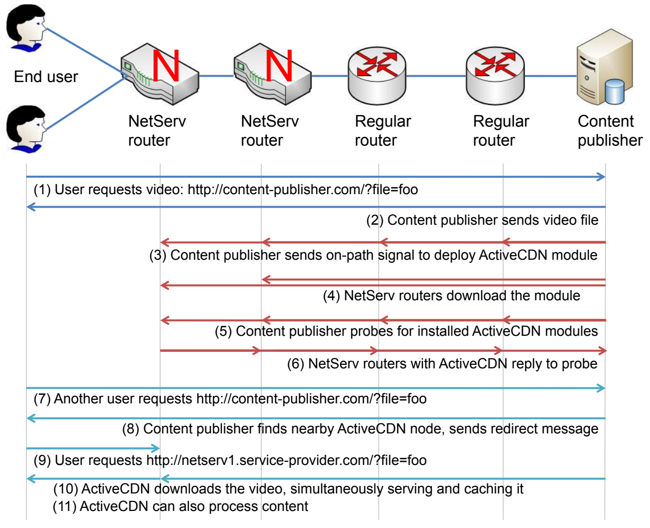

End user
N
N
NetServ router
NetServ router
Regular router
Regular router
Content publisher
(1) User requests video: http://content-publisher.com/?file=foo
(2) Content publisher sends video file
(3) Content publisher sends on-path signal to deploy ActiveCDN module
(4) NetServ routers download the module
(5) Content publisher probes for installed ActiveCDN modules
(6) NetServ routers with ActiveCDN reply to probe
(7) Another user requests http://content-publisher.com/?file=foo
(8) Content publisher 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 6: How ActiveCDN works.

We ran our applications on a topology of four sites across the United States. The testbed was provided by the GENI [3] project. We demonstrated our work at the 9th GENI Engineering Conference (GEC9).

4.1 ActiveCDN

Content publishers currently use CDNs to offload multimedia content. CDNs run on a largely preconfigured topology, which may make it difficult for publishers to place content dynamically according to changing traffic patterns, for example, flash crowds.

We developed ActiveCDN, an 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 publisher. Thus, the publisher has control of what the module does and where it resides. The module's functionality can be updated freely and the module can be redeployed to different parts of the Internet as needed.

Figure 6 offers an example of how ActiveCDN works. When an end user requests video content from a publisher'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 publisher'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 publisher'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.

The module can also perform custom processing. We demonstrated this capability at our GEC9 demonstration. We wrote a custom ActiveCDN module that watermarks a video with local weather information.

4.2 KeepAlive Responder

The ubiquitous presence of Network Address Translators (NATs) poses a challenge to communication services based on Session Initiation Protocol (SIP) [28]. NAT boxes maintain an ephemeral state between internal and external IP addresses and ports, which is referred to as a binding. 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.

Because the timeout interval for expiring NAT bindings has not been standardized, different implementations use different timeouts. The timeout for UDP bindings appears to be rather short in most implementations. As a result, SIP UAs typically send keep-alive messages every 15 seconds [11] 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 (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 [11].

Figure 7 shows how NetServ could help offload NAT keep-alive traffic from the ITSP's infrastructure. 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. Such a module would typically be provided by the ITSP running the SIP server.

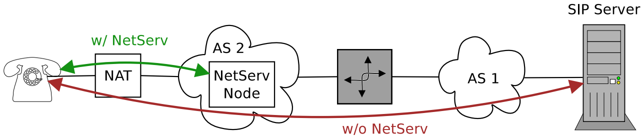


Figure 7: Operation of KeepAlive Responder.

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 in this case because the NetServ router is on the path between the spoofed IP address and the UA.

4.3 Media Relay

NAT boxes may also prevent SIP UAs from directly exchanging media packets, like voice or video. This means that ITSPs must 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. One way to address the drawbacks is to deploy the media relay functionality at the edge of the network closer to UAs.

Figure 8 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.

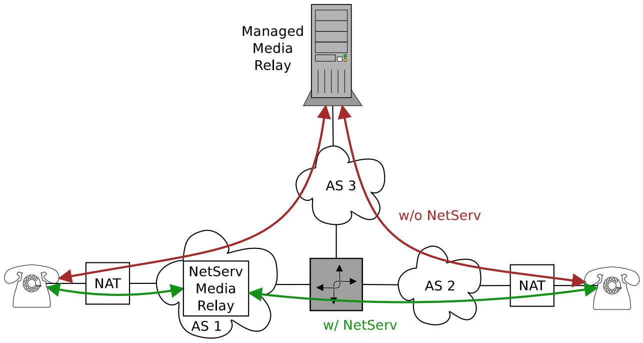


Figure 8: Operation of NetServ Media Relay.

NetServ media relay servers that are deployed at the network edge nicely fit into the Internet Connectivity Establishment (ICE) [27] framework and can be used as TURN servers [25] 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 TURN servers across the Internet. The capability of NSIS signaling to select a TURN server close to one of the communicating UAs helps select TURN 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.

4.4 Overload Control

SIP primarily uses UDP as the transport protocol. This makes SIP servers 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]. The framework proposes to implement the missing control loop (otherwise implemented in TCP) in SIP. Figure 9 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. 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 Sending Entity (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.

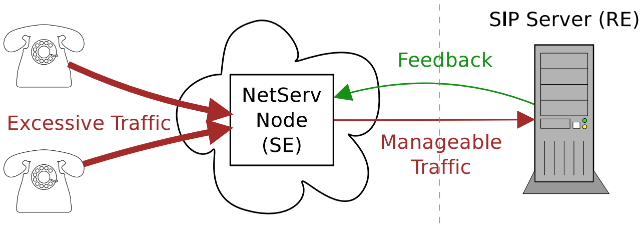


Figure 9: NetServ as SIP overload protection.

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.

4.5 Reverse Data Path

The previous descriptions of the applications 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.

5. EVALUATION

We provide measurement results on what may be the most controversial part of our system: using Java for packet processing. Our results suggest that while there is certainly significant overhead, it is not prohibitive. We measured the Maximum Loss Free Forwarding Rate (MLFFR) of a NetServ router, and compared it with that of a plain Linux host used as a router. This comparison demonstrates the performance overhead introduced by the service layer of NetServ. Our evaluation results are shown without the use of lxc which does not support packet processing yet.

5.1 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 turned off 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.

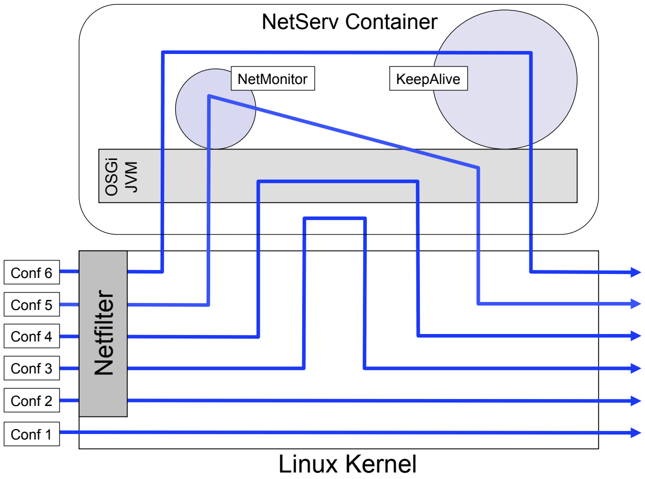


Figure 10: Test configurations 1 to 6.

5.2 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 10 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.

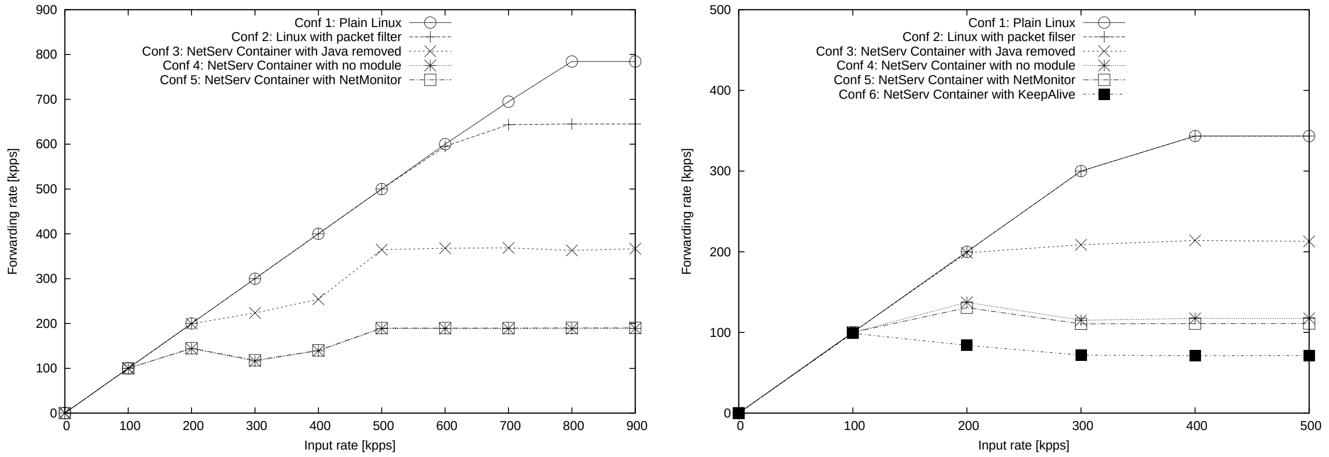
Configuration 1 is the plain Linux router we described above. This represents the maximum attainable rate of our hardware using a Linux kernel as a router.

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 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. NetMonitor sends the counts to a preconfigured server every half-second using a separate thread. This module was part of the network traffic visualization system that we used in the GEC9 demonstration.



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

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

Figure 11: Forwarding rates of the router with different configurations.

Configuration 6 replaces NetMonitor with the KeepAlive module described in Section 4.2. 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 11(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.

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 11(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. The difference between configurations 4 and 5 is explained as the minimal overhead created by the NetMonitor module. Finally, 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 of the router in configuration 1 and 2 reached the theoretical MLFFR of 343 kpps for the 1 Gb/s link.

The MLFFR of the KeepAlive test shows that a NetServ router performs reasonably when compared to the typical traffic volume seen by an edge router today. Real-time router traffic statistics from Princeton University [10] show that the average traffic over the course of a year on an edge router is approximately 32.8 kpps inbound and 31.2 kpps outbound. The NetServ router running KeepAlive was able to achieve 71 kpps in a single direction. We also note that our tests were performed on modest hardware, and more importantly, the packet processing module would only be expected to handle a fraction of the total traffic.

6. RELATED WORK

Many earlier programmable routers focused on providing modularity without sacrificing forwarding performance, which meant installing modules in kernel space. Router Plugins [12], Click [23], PromethOS [21], and Pronto [18] followed this approach. NetServ runs modules in user space, since multi-user execution environment takes priority over raw forwarding performance.

These kernel-level programmable routers, in fact, can be used as NetServ's forwarding plane. We mentioned our on-going and future work with Click, OpenFlow, and Juniper routers in Section 2.3.

LARA++ [29] 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 wide-area deployment.

Active networks [31] proposed two approaches to in-network functionality. In the *integrated* approach, every packet carries code which gets executed in the network nodes. Many researchers attribute the ultimate demise of active networks to the security risk associated with the approach–or at least to the perception of that risk. In the more conservative *discrete* approach, code is installed as modules in the network nodes, and packet headers trigger the execution of the code. All programmable routers, including NetServ, can be viewed as a discrete active network element. Indeed, 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.

GENI is a federation of many existing network testbeds under a common management framework. GENI is important to NetServ for two reasons. First, GENI provides a large-scale infrastructure on which to test NetServ's wide-area deployment mechanisms. Second, GENI is comprised of a diverse set of platform resources, which are shared among many experimenters. NetServ provides a hardware-independent multi-user execution environment where experimenters can run network servers and packet processors written in Java. This can provide an easier development platform for certain experiments and for educational use. We are working on making NetServ a resident feature of the GENI infrastructure.

The Million Node GENI project [7], which is a part of GENI, provides a peer-to-peer hosting platform where an end user can contribute resources from his own computer in exchange for the use of the overlay network. We are particularly interested in its use of Python sandbox, which can offer an alternative to our Java execution environment.

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 publishers, 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 publishers. 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.

7. FUTURE WORK

We plan to extend our framework to support a network monitoring module. Network monitoring modules work similarly to packet processing modules, with the exception that packets are not modified; they simply gather statistics. The current packet processing framework in NetServ is inadequate for this purpose. Clearly it is inefficient and unnecessary to push every packet to user space simply to gather statistics. The forwarding plane needs to provide an interface through which a monitoring module can request to sample packets at a certain rate.

We are exploring the possibility of using NetServ to implement wide-area multicast as a hybrid between IP multicast and application-layer multicast. The NetServ nodes in the network can also utilize storage to provide delayed streaming to save further bandwidth. We are also interested in investigating if NetServ's publisher-specific nature can provide proper economic incentive, which IP multicast failed to provide.

8. CONCLUSION

We present a programmable router architecture intended for edge routers. Unlike previous programmable router proposals which focused on customizing features of a router, NetServ focuses on deploying content and services across ownership boundaries. All our design decisions reflect this change in focus.

We set three main design goals: a wide-area deployment, a multi-user execution environment, and a clear economic benefit. We address these goals by building a prototype using NSIS signaling and user space Java containers, and presenting compelling use cases using example applications.

Our choice of Java and user space module execution has a performance penalty. Our evaluation of the most worrisome case, packet processing in Java, shows that the penalty is significant, but not prohibitive.

Acknowledgements

The authors would like to thank Raynald Seydoux, Abhishek Srivastava and Nathan Miller for their assistance with implementation. This work was supported in part by NSF grant NSF-CNS #0831912.

9. REFERENCES

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

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

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

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

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

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

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

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

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

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

[11] 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.

[12] 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.

[13] 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.

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

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

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

[23] 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.

[24] 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 (available at http://www.cs.columbia.edu/~jae/papers/netserv-tech-report-1.0.pdf), Columbia University, May 2011.

[25] 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.

[26] 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.

[27] 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.

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

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

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

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