PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing

Ralph Keller, Lukas Ruf, Amir Guindehi, Bernhard Plattner
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology, Switzerland
{keller | ruf | guindehi | plattner}@tik.ee.ethz.ch

Abstract. Commercially available routers typically have a monolithic operating system that cannot be easily tailored and upgraded and support new network protocols. PromethOS is a modular router architecture based on Linux 2.4 which can be dynamically extended by plugin modules that are installed in the networking kernel. To install and configure plugins we present a novel signaling protocol that establishes explicitly routed paths transiting selected nodes in a predefined order. Such paths can be non-simple, where a given node is being visited more than once.

Keywords: Active networking, extensible router architecture, explicit path routing, service deployment

1 Introduction

In the past, the functionality of routers was very limited, namely forwarding packets based on the destination address. Recently, new network protocols and extensions to existing protocols have been proposed and are being deployed, requiring new functionality in modern routers at an increasingly rapid pace. However, present day commercially available routers typically employ a monolithic architecture which is not easily upgradable and extensible to keep up with new innovations.

This paper presents the design and implementation of PromethOS[1], an innovative router architecture with a modular design that can be extended to support new and dynamically deployed protocols. The specific objectives of this architecture are as follows:

- *Modularity*. The router architecture is designed in a modular fashion with components coming in form of plugins which are modules that are dynamically loaded into the kernel and have full kernel access without crossing address spaces.

[1] PromethOS originates from Prometheus who was the wisest Titan according to the Greek mythology. His name means “forethought” and he was able to foretell the future. The project was initially codenamed COBRA [14].

- *Flexibility*. For each plugin class, multiple plugin instances can be created. Different configurations of the same plugin can co-exist simultaneously in the kernel, with plugin instances sharing the same code but operating on their own data.
- *Packet classification*. By defining filters, incoming data packets are classified to belong to a data flow and by binding a plugin instance to such a flow, all matching packets will be processed by the corresponding plugin instance.
- *Performance*. An efficient data path is guaranteed by implementing the complete data path in kernel, preventing costly context switches.
- *Code Deployment.* Efficient mechanisms exist to retrieve plugins from remote code servers, install and configure them, and to setup network wide paths such that traffic transits these plugins as desired by the application.
- *Integration in Linux.* The implementation needs only minimal changes to the existing Linux source code and can easily be integrated into newer releases.

We have implemented our framework based on the Linux 2.4 kernel. We have selected this platform because of its portability, freely available source code, extensive documentation, and wide-spread use as a state-of-the-art experimental platform by other research groups. Due to its modularity and extensibility, we are convinced that our proposed framework makes it a useful tool for researchers in the field of programmable router architectures and protocol design. All our code is released in the public domain and can be retrieved from our website [21].

The main contributions of this paper are as follows:

- Design and implementation of a modular and extensible *node architecture* that allows code modules to be dynamically loaded into the networking subsystem at runtime.
- Design and implementation of a novel *signaling protocol* to establish explicitly routed paths through the network and the installation and configuration of plugins along such paths.

In the remainder of this paper, we discuss the design and implementation of our framework. In Section 2, we first focus on a single node, describe the architecture, and consider how it can be extended by installing plugins into the networking subsystem. We demonstrate an example use of the PromethOS plugin framework to give the reader a feel of how the architecture can be used. Section 3 then focuses at the network scope, discusses how explicitly routed paths can be setup, how plugins are retrieved from remote code repositories and installed on selected nodes. Section 4 reviews related work and Section 5 concludes this paper.

2 PromethOS Node Architecture

2.1 Architectural Overview

The main objective of our proposed architecture is to build a modular and extensible networking subsystem that enables to deploy and configure packet processing components for specific flows. Figure 1 illustrates our dynamically extensible router architecture.

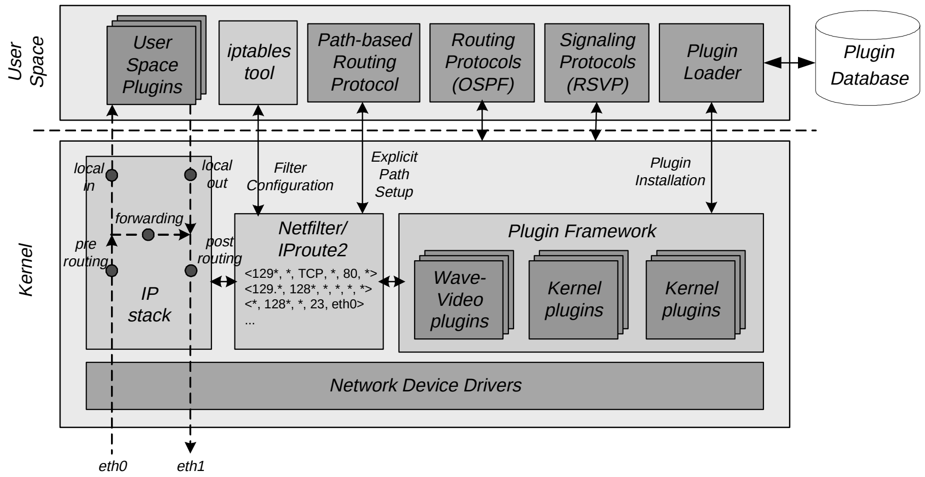

Figure 1 PromethOS modular and extensible router architecture

The most important components are as follows:

- *Network device drivers* implement hardware-specific send and receive functions. Packets correctly received from an interface enter the IP stack.
- *Netfilter* classifies packets according to filter rules at various hooks. Packets matching a filter are passed to registered kernel modules for further processing.
- The *plugin framework* provides an environment for the dynamic loading of plugin classes, the creation of plugin instances as well as their configuration and execution.
- The *plugin loader* is responsible for requesting plugins from remote code servers which store plugin classes in a distributed *plugin database*.
- The *path-based routing* protocol is used to setup explicitly routed paths and to install plugins on selected nodes.
- Other *routing and signaling* protocols compute the routing table and provide resource reservation mechanisms.

2.2 Netfilter Framework

The netfilter framework [23] provides flexible packet filtering mechanisms which are performed at various hooks inside the network stack. Kernel modules register callback functions that get invoked every time a packet passes the respective hook. The user space tool iptables allows to define rules that are evaluated at each hook. A packet that matches these rules is handed to the target kernel module for further processing. The netfilter framework together with the iptables tool provide the minimum mechanisms required to load modules into the kernel, specifying packet matching rules evaluated at hooks, and the invocation of the matching target module.

However, netfilter has a serious restriction since all loadable modules must be *known at compile time* to guarantee proper kernel symbol resolution for the linking process. Thus, only kernel modules that have been *statically configured* can be loaded into the networking subsystem. This is a significant limitation since we envision a router architecture that allows to load arbitrary *new* components at *runtime*.

2.3 Plugin Framework and Execution Environment

To overcome this limitation, we are extending the netfilter framework with a *plugin framework*. The plugin framework manages all loadable plugins and dispatches incoming packets to plugins according to matching filters. When a plugin initially gets loaded into the kernel, it registers its virtual functions with the plugin framework. Once a packet arrives and needs to be processed by a plugin, the framework invokes the previously registered plugin-specific callback function. Since plugins register their entry-points, the entry functions do not need to be known at compile time, and for this reason the plugin framework can load and link any plugin into the kernel.

Every PromethOS plugin offers an input and output channel (in accordance with [7]) representing a *control* and *reporting* port. The control port is used for managing the PromethOS plugin (such as configuration); the reporting port is read-only to collect status information from the plugin.

2.4 Plugin Classes And Instances

For the design of plugins, we follow an object-oriented approach. A *plugin class* is a dynamically loadable Linux kernel module that specifies the general behavior by defining how it is initialized, configured, and how packets need to be processed. A *plugin instance* is a runtime configuration of a plugin class bound to a specific flow. An instance is identified by a node unique instance identifier. In general, it is desirable to have multiple configurations of a plugin, each having its own data segment for internal state. Multiple plugin instances can be bound to one flow, and multiple flows can be bound to a single instance. Through a virtual function table, each plugin class responds to a standardized set of methods to initialize, configure, reconfigure itself, and for processing packets. All code is encapsulated in the plugin itself, thus the plugin framework is not required to know anything about a plugin's internal details. Once a packet is associated with a plugin, the plugin framework invokes the processing method of the corresponding plugin, passing it the current instance (data segment) and a pointer to the kernel structure representing the packet (struct sk_buff).

2.5 Control from User Space

PromethOS and its plugins are managed at load-time by providing configuration parameters and at run-time through the control interfaces via the procfs. When the PromethOS plugin framework initially gets loaded, it creates the entry /proc/promethos. Below this entry, the control and reporting ports of individual plugins are registered. PromethOS plugins are loaded by iptables which we extended with semantics required for the PromethOS plugin framework. The communication to control plugins and report messages between user space and plugins follows a request-reply approach. A control message is addressed to the appropriate plugin by passing the plugin instance identifier as a parameter and the plugin then responds with a reply.

2.6 Example Use of PromethOS Plugin Framework

To give the reader a feel for the simplicity and elegance with which plugins can be put into operation, we illustrate the commands necessary to load and configure a WaveVideo [17] plugin performing video scaling. Note that these commands can be executed at any time, even when network traffic is transiting through the system. As mentioned above, we use a PromethOS-enhanced iptables program that interacts with the iptables framework. In the extension of iptables, we implement calls to the insmod program, which serves as the primary tool to install Linux kernel modules.

- Loading and registering plugin:

  ```
  # iptables -t promethos -A PREROUTING -p UDP -s 129.132.66.115 -dport 6060
  -j PROMETHOS --plugin WV --autoinstance --config "65536"
  ```

  This command adds a filter specification to the PromethOS plugin framework, requesting to install the WV plugin at the PREROUTING hook, and creating an instance of this plugin to perform video scaling at 65536 Byte/s. If the plugin framework is not yet loaded, the module dependency resolution of Linux installs it on demand.
- Upon successful completion of the plugin loading and instantiation, the plugin framework reports the plugin instance number:

  ```
  Router plugin instance is 1
  ```
- By this instance number, the plugin control port can be accessed:

  ```
  # echo '#1' 131072 > /proc/promethos/net/management
  ```

  This reconfigures the WV plugin to scale the video to a maximum output of 131072 Byte/s.
- The configuration of the PromethOS table can be retrieved with iptables:

  ```
  # iptables -t promethos -L
  Chain PREROUTING (policy ACCEPT)
  target    prot opt source         destination
  PROMETHOS udp -- 129.132.66.115 anywhere udp dpt:6060 WV#1
  ```
- The plugin and the framework may be removed from the kernel by the standard mechanisms provided by iptables and the Linux kernel module framework.

This example demonstrates the seamless integration of the PromethOS plugin framework in Linux, allowing to load arbitrary code at runtime.

3 Code Deployment on Explicitly Routed Paths

In the previous section, we have presented our active node architecture that can be dynamically extended with components coming in form of loadable kernel modules. The mechanisms illustrated for installing and configuring plugins require *local access* to the router which is a feasible approach for setting up routers with a static configuration. However, the active networks paradigm envisions an infrastructure that can be programmed by network administrators and end users in a more flexible fashion. For active networks we need new routing mechanisms that take into account that end-to-end paths include processing sites.

In this section we present a novel signaling protocol that allows to *deploy plugins on selected nodes* and to *establish paths transiting these nodes* in a given order. In the context of active networks conventional destination-based routing schemes cannot satisfy the requirements demanded by active applications since traffic needs to transit processing sites generally *not located* on the IP default path. In our opinion, the introduction of new code into routers should be performed in a *structured way*, where network service providers or end users *explicitly configure* the network with the required functionality, enabling efficient allocation of network resources among competing applications.

Finding an optimal routing *and* processing path can be seen in the context of constraint-based routing, where processing constraints define requirements on the order and location of processing functions. A suitable algorithm that finds an optimal route through a sequence of processing sites has been proposed in [10].

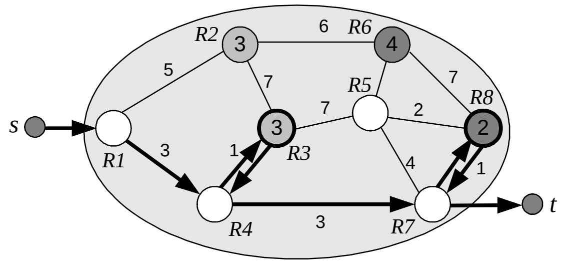

Figure 2 Optimal solution with intermediate processing can produce non-simple path

Figure 2 depicts a sample network with various processing sites. Each site has an associated cost (shown as the number in the node) that needs to be taken into account if processing on that site occurs. In the example, we are looking for an end-to-end path that includes two intermediate computations, with the constraints that the first computation should be placed on one of the light grey nodes (*R2* or *R3*), and the second on one of the darker nodes (*R6* or *R8*). An optimal solution for this constraint-based routing problem (taking into account *both* link and processing costs) can produce a *non-simple path*, also known as a *walk*, which is a sequence of consecutive edges where a given vertex is being visited more than once. Since such solutions are now possible when considering active processing, the signaling protocol must also support such paths. In the following we assume that such explicitly routed paths can be computed according to [10] and focus on the signaling protocol required for configuring such combined routing and processing paths.

3.1 Explicit Path Establishment

Our proposed *Path-Based Routing* (PBR) [15] protocol supports per-flow based explicit path establishment for one-way, unicast flows routed through a predefined list of hops and the installation and configuration of plugin modules along such paths.

As illustrated in Figure 3, the path establishment is based on a *two-phase* scheme: In the first phase, the protocol verifies whether sufficient resources are available along the *downstream path*. Beginning at the source, each node checks whether the required resources are available locally and if granted, reserves (but does not allocate yet) resourc-

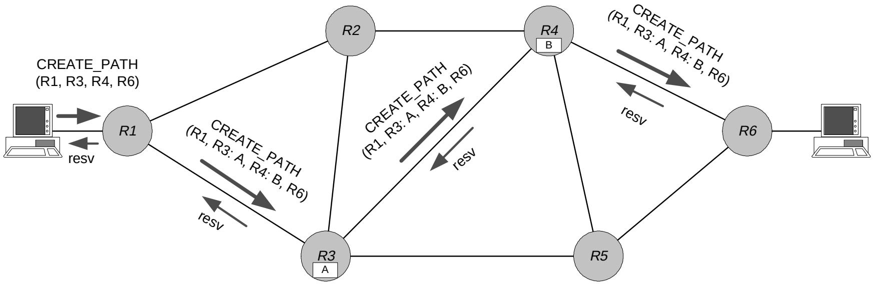

Figure 3 Explicit path setup using PBR protocol

es, and forwards the reservation request to the next node along the downstream path. This process is repeated until the destination node is reached. Once the first phase of the setup process has been completed, it is assured that sufficient resources are available. In the second phase, the actual allocation of network resources takes place. This happens along the *reverse path*, that is, on all routers from the destination towards the source node. This includes the installation of flow-specific filters such that packets matching the filter are forwarded on the corresponding outgoing interface, and the installation and configuration of plugins and binding them to the filter. Once all state has been established along the path, the application is informed and can transmit traffic.

If during the first phase a request is refused due to limited resources, the path setup process cannot continue and is aborted. The node then sends a reservation release message along the reverse path so that nodes that have already reserved resources can free them. Finally, the application is notified that the path could not be established.

The PBR protocol uses TCP as the transport mechanism between PBR peers (hop-by-hop) to send control messages for path establishment, plugin deployment, and release of resources. This guarantees reliable distribution of control messages. PBR uses *soft-state* for both path and plugin information being stored on nodes to take into account that network nodes and links are inherently unreliable and can fail. An application that sets up a path is required to refresh the path (by sending the setup request periodically), otherwise nodes will purge path and plugin state once the time-out expires. Path tear down works analogous to the path setup process, with a release request used instead.

3.2 Plugin Deployment

In addition of setting up flow-specific routes, the PBR protocol allows to install and configure plugin modules on selected nodes. To support this feature, the path establishment message includes a list of nodes where plugins need to be installed. If the requested target address for a plugin matches a node's own address, the node first checks whether the referred plugin class has already been loaded into the kernel. If it is not present, the plugin loader retrieves it from a remote code server and verifies the consistency by checking the module's digital signature [9]. Then the module is loaded into the kernel and linked against the current kernel image. Subsequently, the PBR daemon creates a new instance of the plugin, invokes the configuration method, and binds the plugin instance with the filter describing the flow. Once the path has been established and the required plugins deployed, the application can begin transmitting data which will be forwarded along the path and processed by intermediate plugins.

3.3 Message Details

A *CREATE_PATH message* is transmitted by the path-initiating router toward the destination to establish an explicitly routed path. The message contains the following different subobjects:

- *FLOWSPEC object*
  The flow specification describes the format of packets that follow the explicit path, described using the tuple <source addr/mask, dest addr/mask, source port, dest port, protocol>. Any field can be wildcarded, network addresses can be partially wildcarded with a prefix mask.
- *EXPLICIT_ROUTE object*
  An explicit route object is encoded as a series of nodes to be traversed. In the current implementation, the hops must form a strict explicit route.
- *PLUGIN object*
  The plugin object describes one or multiple plugins that need to be deployed on a node. It contains the address of the target node, followed by the plugin name and an initial configuration parameters.

The *RELEASE_PATH* message removes a previously established path. The PBR protocol also supports the *STATUS* message for the retrieval of path state from remote nodes.

3.4 Forwarding Mechanisms for Explicit Path

In the following we describe how we can override Linux's conventional destination-based forwarding and perform our own explicit path routing, that is, move packets along a predefined set of nodes. For each flow that requires flow-specific forwarding, we add a filter entry into netfilter. Incoming packets are classified by the netfilter framework and if a flow-specific filter matches, they are marked with a *next-hop neighbor tag*. For each adjacent neighbor, there is a special routing table containing a single default route entry pointing to the corresponding neighbor. Packets that have been marked with the next-hop tag then use the corresponding routing table and are sent to the appropriate neighbor. To establish a complete path, filter entries are added to all nodes along the path.

As discussed above, when considering processing sites the path from the source to the destination does *not need to be a straight IP path* anymore, where all of the nodes are distinct and no duplicated nodes exist. To support such paths, the forwarding mechanism must consider the *incoming port* from where the packet has been received. For that reason, flow filters consist of a six tuple, including the *incoming interface* for the forwarding decision as well.

Note however that the current implementation is restricted to paths that enter a node via the same interface only once, since a node cannot distinguish if a packet has previously traversed the node. This limitation could be overturned if incoming packets would be marked with a tag to be used for subsequent forwarding decisions.

3.5 Example Use of Path-Based Routing

In the scenario as illustrated in Figure 2 we want to install an encryption and decryption plugin on selected nodes. When considering processing costs, the optimal solution for such a path is *non-simple*. The following command establishes a path which routes all traffic matching the filter along the path:

```
# pbr create R1:R4:R3:R4:R7:R8:R7 --dport 6060 --plugin R3:ENCRYPT:init
--plugin R8:DECRYPT:init
```

4 Related Work

In this section we look at related work, both at active network architectures and mechanisms for explicit routing and remote code deployment.

4.1 Programmable Network Architectures

In the context of programmable networks, several node architectures have been proposed allowing to dynamically extend the networking subsystem of a router with additional functionality.

The *Active Network Node* [13] is a NetBSD-based architecture that allows code modules called router plugins to be dynamically downloaded and installed into the OS kernel and binding plugin instances to flows. PromethOS's plugin concept has been inspired mostly from this project. While ANN provides many of the concepts implemented, PromethOS requires less modification of the original network stack.

Scout [19] proposes a path-based approach where the functionality of a standard IP-compliant router is decomposed into a sequence of a interconnected components forming a path. Recently, Scout has been ported to Linux [4], however requiring to replace most of the Linux network stack with the Scout implementation.

NetGraph [11] is a the network stack filtering mechanisms for FreeBSD similar to netfilter. The concept is based on hooks offering bidirectional communication for components attached to the network stack. These hooks are freely interconnectable to form a network graph.

Click [18] is an architecture for assembling a router from packet processing modules called elements. Individual elements implement router functions like packet classification, queuing, scheduling, and interfacing with network devices. A router configuration is a directed graph describing the required components and how packets flow through the router. Click configurations are later compiled to machine code, providing very good performance. Once defined configurations are static and cannot be tailored at runtime (unlike PromethOS plugins). The static approach of Click is overcome by the *Open Kernel Environment Corral* [5] which makes use of the type-safe C programming language *Cyclone* [12] that is extended by authentication mechanisms for accessing resources.

4.2 Explicit Path Routing and Service Deployment

Several resource allocation protocols capable of supporting applications that request per-flow routing and allow functions to be deployed in the network core have been developed. This section briefly describes a few of these protocols.

Table 1: Comparison of signaling protocols supporting explicit paths

	Source Routing	*ATM/ PNNI*	*MPLS*	*Beagle*	*PBR*
Plugin deployment	no	no	no	yes	yes
Explicit routing	strict or loose	strict or loose[a]	strict or loose	strict or loose	strict[b]
Looping paths	no[c]	no	no	no	yes
Router state	none[d]	VCI/VPI entry	MPLS tag entry	RSVP filter entry	netfilter filter

[a] PNNI supports hierarchical routing where the source can address the logical group leader, representing an aggregation of nodes.

[b] PBR currently supports only strict routes but loose routes could be easily implemented.

[c] May be possible but not intended by IP protocol.

[d] Hop addresses are stored directly in the IP option header. However, due to the option header length limit, the number of hops is restricted to eight.

The *IP source routing option* [20] provides a means for the source of an IP datagram to supply routing information to be used by intermediate routers. The route data are composed of a series of Internet addresses present in the IP option header. Since there is an upper limit of the option header length, only 8 hosts can be explicitly routed.

The *Private Network-to-Network Interface* [1] is a signaling and routing protocol between ATM switches with the purpose of setting up Virtual Connections (VCs). PNNI determines an optimal path satisfying QoS constraints and reroutes connections (crankback) when VC establishment fails. PNNI performs *explicit source routing*, in which the ingress switch determines the *entire path* to the destination. Setting up explicit paths is seen as an attractive feature of ATM since each application can have its own specific path requirements. Nevertheless, ATM does not support the concept of processing resources as introduced by active networks.

The *Multiprotocol Label Switching* [22] approach is based on a label-swapping paradigm implemented in the networking layer. MPLS defines two label distribution protocols that support *explicitly routed* paths. *CR-LDP* [16], which is an extension of LDP [3], is peer-to-peer protocol where messages are reliably delivered using TCP and state information associated with explicitly routed LSPs does not require periodic refresh. An explicit route can be *strict*, where all nodes are explicitly listed, or *loose*, allowing to define paths through portions of the network with partial knowledge of the topology. *RSVP-TE* [2] extends the original RSVP [6] protocol by setting up explicit label switched paths and to allocate network resources (e.g., bandwidth). The *explicit route object* encapsulated in a Path message includes a concatenation of hops, describing a strict or loose route. RSVP-TE is based on soft state, where the state of each LSP must periodically be refreshed (typically every 30 seconds). CR-LDP and RSVP-TE are signaling protocols that perform similar functions but currently no consensus exists on which protocol is technically superior.

Beagle [8] is a signaling protocol for the setup of structured multi-party, multi-flow applications described by an application mesh. The mesh formulates the resources to be allocated as a network graph. The Beagle protocol is based on RSVP and introduces a new *route constraint object* carrying explicit routing information. In contrast to signaling protocols like MPLS and PNNI, Beagle allows applications to allocate computation and storage resources required for delegates, which are application-specific code segments that execute on routers.

The PBR protocol has specifically been designed for active networks. It allows the deployment of new code on routers and the setup of explicitly routed paths, supporting also looping paths such that the same processing site can be visited multiple times.

5 Conclusions

In this paper we have presented PromethOS, an extensible and modular architecture for integrated services routers. PromethOS allows to dynamically load plugins at runtime into the kernel, to create instances of plugins, and to bind plugin instances to individual flows. The path-based routing protocol establishes explicitly routed paths and installs plugins on selected nodes. We freely distribute our source code with the intent of providing the research community with a services platform to build upon.

Currently, PromethOS is being extended to provide resource control mechanisms for plugins in kernel space. We focus on aspects of memory consumption, processor cycles and bandwidth on both general purpose and network processors.

References

[1] ATM Forum Technical Committee, “Private Network-Network Interface Specification Version 1.0,” March 1996.

[2] Awduche, D., Berger, L., Gan, D., Li, T., Swallow, G. and V. Srinivasan, “RSVP-TE: Extensions to RSVP for LSP Tunnels,” *RFC 3209*, December 2001.

[3] Andersson, L., Doolan, P., Feldman N., Fredette, A., Thomas, B., “LDP Specification,” *RFC 3036*, January 2001.

[4] Andy Bavier, Thiemo Voigt, Mike Wawrzoniak, Larry Peterson, Per Gunningberg, “SILK: Scout Paths in the Linux Kernel,” Department of Information Technology, Uppsala University, 2002.

[5] Herbert Bos and Bart Samwel, “The OKE Corral: Code Organisation and Reconfiguration at Runtime using Active Linking,” *IWAN 2002*, December 2002.

[6] Braden, R., Zhang, L., Berson, S., Herzog, S. and S. Jamin, “Resource ReSerVation Protocol (RSVP) – Version 1, Functional Specification,” *RFC 2205*, September 1997.

[7] K. L. Calvert et al., “Architectural Framework for Active Networks Version 1.0,” *DARPA Active Network Working Group Draft*, July 1999.

[8] Prashant Chandra, Allan Fisher, Peter Steenkiste, “Beagle: A Resource Allocation Protocol for Advanced Services Internet,” Technical Report *CMU-CS-98-150*, August 1998.

[9] Sumi Choi, “Plugin Management,” Washington University in St. Louis, *Technical Report WUCS-00-04*.

[10] Sumi Choi, Jonathan Turner, Tilman Wolf, “Configuring Sessions in Programmable Networks,” In Proceedings of Infocom 2001, March 2001.

[11] A. Cobbs, “All About NetGraph,” http://www.daemonnews.org/200003/netgraph.html, 2001.

[12] Cyclone, AT&T Research Labs and Cornell University, http://www.research.att.com/projects/cyclone, 2001.

[13] Decasper, D., Dittia, Z., Parulkar, G., Plattner, B., “Router Plugins – A Modular and Extensible Software Framework for Modern High Performance Integrated Services Routers,” *Proceedings of ACM SIGCOMM'98*, September 1998.

[14] Amir Guindehi, “COBRA – Component Based Routing Architecture,” *Semester Thesis SA-2001.30*, Swiss Federal Institute of Technology Zurich, In German.

[15] Amir Guindehi, “PBRD – Path Based Routing Daemon,” *Semester Thesis SA-2002.17*, Swiss Federal Institute of Technology Zurich, In German.

[16] Jamoussi, B. et al, “Constraint-Based LSP Setup using LDP,” *RFC 3212*, January 2002.

[17] Ralph Keller, Sumi Choi, Dan Decasper, Marcel Dasen, George Fankhauser and Bernhard Plattner, “An Active Router Architecture for Multicast Video Distribution,” *Infocom 2000*, Tel Aviv, March 2000.

[18] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, M. Frans Kaashoek, “The Click modular router,” *ACM Transactions on Computer Systems 18(3)*, August 2000, pages 263-297.

[19] David Mosberger, Larry Peterson, “Making Paths Explicit in the Scout Operating System,” *Operating Systems Design and Implementation*, pages 153-167, 1996.

[20] J. Postel. “Internet Protocol,” *RFC 791*, 1981.

[21] PromethOS website, http://www.promethos.org/

[22] Rosen, E., Viswanathan, A., Callon, R., “Multiprotocol Label Switching Architecture,” *RFC 3031*, January 2001.

[23] Rusty Russell, “Linux NetFilter Hacking HOWTO,” http://www.netfilter.org/