An Active Node Architecture for Proactive Services

Hoa-Binh Nguyen, Andrzej Duda

LSR-IMAG Laboratory

Grenoble, France

e-mail: {Hoa-Binh.Nguyen, Andrzej.Duda}@imag.fr

phone: +33 476 827 268

Abstract

We propose an active node architecture that supports *proactive services* able to take into account *adaptability* to varying conditions and flexible *supervision* by the application or the user. Proactive services are software modules that can be deployed in the network at chosen nodes. Each service processes flows of packets forwarded by a node. A packet filter recognizes some packets according to the information in the packet header and passes them to the right proactive service. Intercepting packets can be activated and disabled dynamically, so that there is no overhead for forwarding packets that do not require active processing. Association of a proactive service with the packet filter is also dynamic. Services are adaptable with respect to the environment of the active node by means of monitors, which gather information from probes and notify services about the current state of the active node, links, or other network resources.

We have prototyped the active node for proactive services on Linux. Each proactive service runs as a user space process. We have modified the Netfilter module in the Linux kernel so that it delivers matching packets directly to the right user process without passing by a multiplexer daemon. Measures of packet forwarding show that deviating packets to the user space for processing incurs a slight overhead, however it only concerns data flows that require specific processing by proactive services.

Keywords: Proactive services, adaptation, active node on Linux

1 Introduction

In a recent work, David Tennenhouse has introduced the concept of *proactive computing* [19]: as the number of computer devices grows, our traditional office-centered computing based on close human-computer interaction will need to evolve to a new mode of operation in which networked systems composed of many processors, sensors and actuators work in an independent way. The role of humans is reduced to supervision. Such proactive systems will be connected to the world around them, using sensors and actuators to both monitor and shape their physical surroundings. We think that these ideas can be successfully applied to active networking.

The principal idea of *active networking* is the introduction of programmability into the heart of the network [13, 4, 9, 11, 22]. An intermediate system in an active network provides an execution environment for mobile code that can be loaded and linked on demand depending on application requirements. This approach allows to build new services or protocols in a flexible manner and deploy them at network nodes dynamically. Its main advantage is the ability to express the behavior of a network node by means of a programming language.

Even if programmability may be added to any network node such as a router, a gateway, or a proxy, probably the most suitable place for introducing active nodes is the edge of the network infrastructure. Many applications require new services to be physically situated at an appropriate strategic location within the network. For example, a video gateway that adjusts the bit rate of video stream to accommodate the constrained capacity of communication links should be located near such critical links. The approach based on *active services* [2, 11] proposes sufficient functionalities to deal with such examples. However, pervasive environments such as those described above present new requirements and challenges. Two most important of them are *adaptability* to varying conditions and flexible *supervision* by the application or the user. We propose an active node architecture that supports *proactive services* able to take into account these requirements.

Proactive services are software modules that can be deployed in the network at chosen nodes. Each service processes flows of packets forwarded by a node. A packet filter recognizes some packets according to the information in the packet header and passes them to the right proactive service. Intercepting

packets can be activated and disabled dynamically, so that there is no overhead for forwarding packets that do not require active processing. Association of a proactive service with the packet filter is also done dynamically by a service itself. Proactive services may use for example a generic parser that can be instantiated to parse the contents of packets, for example the contents of a HTML page, an MPEG frame, or a SIP segment. Delegating such processing to services makes simpler and more efficient the operation of the packet filter.

Services are adaptable with respect to the environment of the active node by means of monitors, which gather information from probes and notify services about the current state of the active node, links, or other network resources. The user can also supervise the execution of services by setting their parameters or policies. Service deployment and supervision may be only done by authentified trusted third parties.

We have prototyped the active node for proactive services on Linux. Each proactive service runs as a user space process. We have modified the Netfilter module in the Linux kernel so that it delivers matching packets directly to the right user process without passing by a multiplexer daemon. To enforce security, every communication with proactive services (activation, event notification) passes through a control module that takes care of authentification. Proactive services can be written in any language, currently we use C, C++, and Java.

The paper is organized as follows. Section 2 presents related work. Section 3 presents the architecture of an active node supporting proactive services. In Section 4, we describe a prototype implementation on Linux. Section 5 reports on the measured performance of our active node. Finally, we present conclusion and future work (Section 6).

2 Related work

We can classify active networks into two main categories. The first one is based on active packets (capsules) [12, 18, 22] that carry executable code executed on some data at network nodes. A packet can only change the state or the behavior of a node. Nodes provides execution environments for packets. In some cases, the executable code can be reduced to some identifiers or references to predefined functions

that reside in network nodes [3, 5]. Packets decide which functions to execute on their contents and they provide parameters for these functions.

The second category is based on active nodes that process passive packets [1, 10, 7]. A packet classifier invokes a right function on a chosen packet.

In some cases, the two approaches are mixed together [8]: an active node supports both the execution of active packets and processing of passive packets.

Some authors claim that active networks should only be provided at nodes located at the network border and all processing should be done at the application level. AS1 [2] and ALAN [11] are two platforms that allow users or applications to download active services at some service servers. These active services are in fact some kinds of application level proxies. The media gateway (MeGa) service in AS1 is an application-level media gateway to transcode different formats of video and audio on behalf of users. The compression proxylet in ALAN is an active service to compress a data flow.

The active services proposed by AS1 or ALAN fail to provide two important features: adaptability and trusted supervision. Once deployed and activated, active services do not take into account the state of the node on which they are running or the network: CPU load, network congestion or other events that can happen during the execution. These platforms do not provide support for notifying services about some events. Only users can configure an active service by sending new parameters. In this case, it is up to the active service to authenticate users. This means that authentication should be done twice, the first time when the user activates an active service and the second time when the user wants to change some parameters. So each active service has to implement the authentication module.

Moreover, ALAN uses Java Virtual Machine (JVM) as the execution environment for active services and each active service (proxylet) runs in a separate JVM. This limits the number of active services that can be activated at the same time (ALAN limits this number to 4).

Chameleon is a component-based service model for high performance active networks [6]. The model takes care of heterogeneous active nodes and environments to deploy appropriate service instances at the right place. The authors recognize the necessity of adaptive services in active networks, services in Chameleon, once deployed and activated, cannot adapt their behavior to the state changes in routers or

in the network.

We follow the approach initiated by AS1 and ALAN, however we enhance their functionalities. We would like to provide the possibility to process packets not only at the application level, but also at the network or transport layers without significant performance penalty. We also want to provide a generic parsing module so that the development of new services that process the contents of packets is easy.

Our architecture tries to address the issues of adaptability and flexible supervision by providing support for modifying the behavior of active services based on the state of the node, the network or the user. Communication between users and proactive services is done through a common control module that implements authentication. In this way, proactive services do not need any integrated authentication.

Adaptive active services have been already proposed in the context of link error control. Kulkarni [14] proposed an adaptive Forward Error Correction (FEC) service to take into account the varying bit-error rate (BER) in wireless networks. Boulis et al. [7] have implemented a service placed in an active base station that fragments long audio packets into several smaller ones. In the case of a noisy environment, many long packets are lost. If they are fragmented, only a small fraction of packets are lost and the quality of audio becomes much better. However, if the channel has good quality, the fragmentation deteriorates the audio quality, because of the increased jitter. This example calls for an adaptive service that only intercepts and fragments audio packets when needed. Our architecture extends this approach by making active services dependent on their environment.

3 Active node architecture for proactive services

We have design the architecture of an active node that supports proactive services. Such a node, besides providing standard active node functionalities, should offer:

- the association of proactive services with chosen data flows, done dynamically by the services themselves,
- monitors able to detect varying conditions in the environment (network, active node, services,

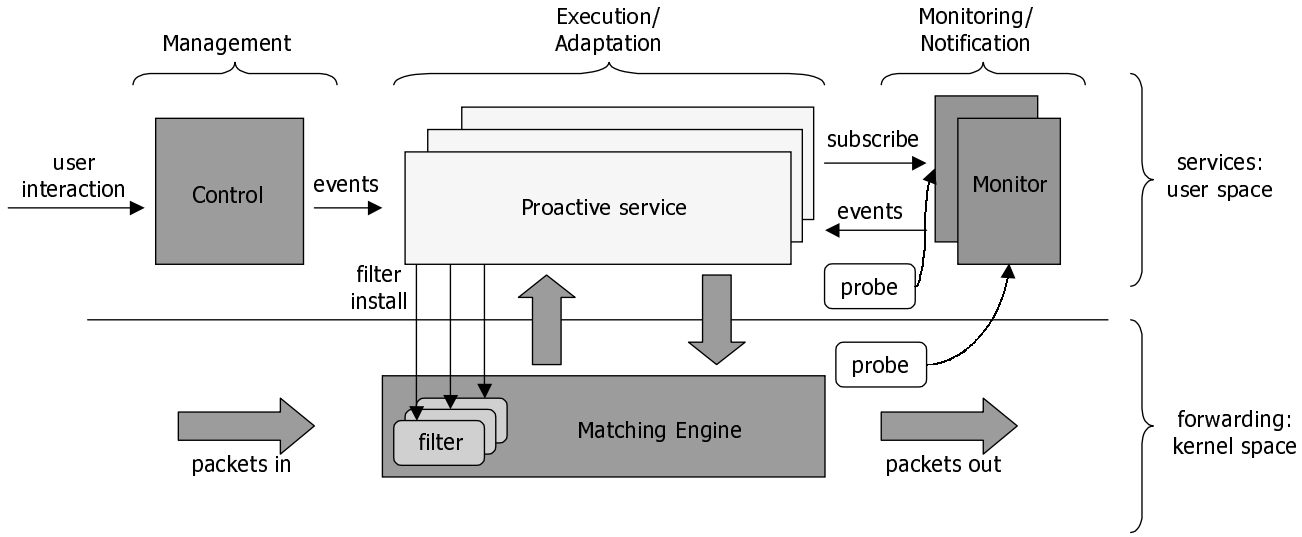

Figure 1: Active node architecture for proactive services

users),

- an asynchronous notification mechanism that allows monitors to notify proactive services.

Figure 1 presents a general view of the architecture. The node provides two basic levels of abstraction. The lower layer takes care of packet forwarding. The upper layer provides support for proactive services, adaptation, and supervision. The architecture includes the following elements:

- *Matching engine*. Input packets go through a *matching engine* that allows to dynamically install and uninstall *packet filters* responsible for intercepting packets and passing them to *proactive services*. Packets that do not match any filter are forwarded in a standard way to output links.

- *Proactive services*. Proactive services are code modules dynamically instantiated to enhance network functionalities on behalf of a given data flow. They are developed by a third party trusted users, deployed by node administrators, and activated by users after an authentication procedure via a *control module*. An activated service dynamically installs a filter to receive chosen packets based on a simple criteria involving IP addresses, port numbers, and protocol types. The service may further analyze a packet (for example, parse its contents), process, and re-inject it into the kernel. When the service does not longer need to receive packets, it can uninstall the filter, so that packets go through the node without any additional overhead.

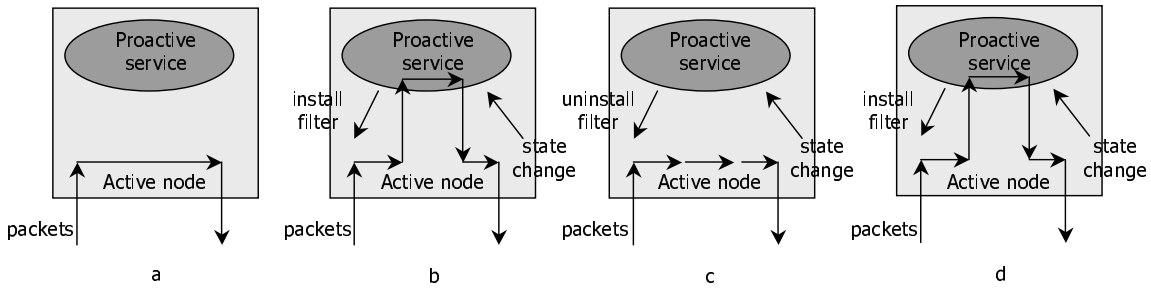

Figure 2: Dynamic behavior of proactive services

- *Control module.* The control module takes care of authentifying administrators and users. It activates or suspends services according to users requests. Users can pass parameters to the activated services and specify rules of supervision to the control module that will in turn notify the services by means of events. The control module also manages resources of the node: CPU, memory, storage.

- *Monitors.* To become reactive to the state of the environment, a service subscribes to a set of *monitors* able to gather events interesting to the service from *probes*. The monitor signals a service by sending an event so that the service can react to the changed state. Probes are able to detect for example congestion on a specific link, degraded conditions of a wireless link, limited buffer space, increased jitter for some time critical streams, disconnected mobile device, and many others related to the state of the network, the active node, or the users. In this way, a service can install a packet filter and process packets when it is needed.

Figure 2 illustrates dynamic behavior of proactive services. We assume that when a service is activated, the state of the environment is such that no processing is required. The service sleeps and the forwarding layer processes packets without any performance penalty (Figure 2.a). Packets are not passed to the service. When the monitor detects a change in the state, for example, congestion on a link, an event is sent to the service. The service wakes up and installs an appropriate packet filter in the matching engine that passes matching packets to the service for required processing (Figure 2.b). When the state changes again (congestion disappears), another event is sent to the service which uninstalls the packet filter so that packets are no longer intercepted. The service returns to sleep (Figure 2.c). The whole

process may repeat (Figure 2.d).

This way of operation avoids passing packets to the service when it is not necessary. This approach differs from other architectures in which a control module installs a packet filter immediately upon the activation of an active service so that the service always receives packets even if it does not perform any useful processing of packets. We obtain two benefits from this mode of operation: the active node can spare resources and data flows do not incur performance penalty when there is no need for packet processing. Moreover, services can still receive packets immediately upon activation as in other architectures.

3.1 Events

Each monitor has to specify a resource that should be monitored. In some cases, a service can directly provide such a resource to a monitor. For example, a monitor that keeps track of the presence of a host needs to know its name. When a service wants to survey its clients, it should provide their names to the presence monitor.

A resource is specified by a set of parameters. Each parameter has a set of values and a finite number of states. Each state corresponds to a set of parameter values. The change from one state to another will trigger an event. A service can specify these states by providing the parameter values to the monitor.

Events are typed, their definition includes the event name and the event data, which is a list of event attribute values that can be numerals or strings.

To cooperate with monitors, a proactive service need to specify subscription requests and define rules for reacting to events. The rules describes which action execute when a given event occurs.

3.2 Application scenarios

To illustrate how proactive services can be used, we describe an application scenario oriented towards wireless mobile networks. A mobile user receives a MPEG video flow from a streaming server controlled by RTSP. The network provider has deployed on an active node close to the wireless link a *caching service* that stores packets when the mobile is temporarily disconnected. The service subscribes to a *presence*

monitor which surveys the presence of the mobile by periodically sending a probe packet. When the mobile is connected, the active node forwards the video flow packets as in a regular router, the service being active, but packets are not intercepted. When the monitor detects the disconnection of the mobile, it sends a notification to the service that installs a packet filter. Packets are then forwarded to the service and stored. The service issues a PAUSE command to the RTSP server to suspend streaming. When the mobile reconnects, the monitor notifies the service that in turn sends the cached packets and resumes streaming by using the PLAY command.

A similar service can adapt the video flow to the available bandwidth on a wireless link. In the case of 802.11 wireless LANs, the effective throughput depends on the quality of the radio channel: if the quality degrades, the nominal bandwidth is reduced from 11 Mb/s to 5.5, 2, and even to 1 Mb/s. The video adaptation service dynamically changes the resolution or the type of encoding (H.263 instead of MPEG) of the video to adapt to the available bandwidth.

4 Implementation of the active node on Linux

We have implemented the active node supporting proactive services on Linux. Linux is a good candidate for such a node because of its interesting properties: packet forwarding support, loadable kernel modules, and the ease of modifying the kernel behavior. Forwarding packet part of our architecture is implemented in the Linux kernel and all other components are implemented as user space processes. In particular, each proactive service is executed as a separate process.

4.1 Matching engine

The matching engine allows to dynamically install and uninstall packet filters in the forwarding path. From the Linux kernel version 2.4, there is a support for custom processing of packets in the kernel: Netfilter [20, 15]. It allows users to hook extended modules in the packet forwarding path and to pass packets of a flow to a process in the user space for further processing. After processing packets are re-injected into the kernel . The architecture of Netfilter is presented in Figure 3.

Packets entering the kernel pass through hook 1 (`NF_IP_PRE_ROUTING`). After hook 1, packets go to

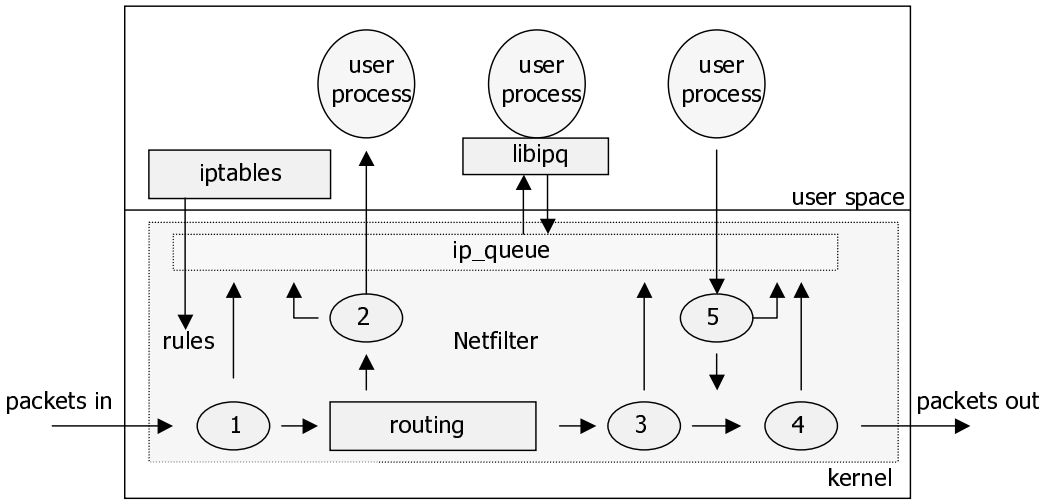

Figure 3: Netfilter Architecture

the routing module that decides weather they should be further forwarded or delivered to the local host. If packets are local, they are passed to hook 2 (`NF_IP_LOCAL_IN`). If packets must be forwarded, they go to hook 3 (`NF_IP_FORWARD`) and then to hook 4 (`NF_IP_POST_ROUTING`). When packets are sent out from a user process, they go to hook 5 (`NF_IP_LOCAL_OUT`) before being passed to hook 4.

At each hook, we can add a kernel module that will be called for each packet going through the hook. The module is free to alter the packet and it must return to Netfilter one of the following verdict values that determine a corresponding action on the packet:

- `NF_ACCEPT`: the packet can continue as a regular packet
- `NF_DROP`: the packet must be dropped
- `NF_STOLEN`: the packet is stolen by the module; do not continue
- `NF_QUEUE`: the packet should go to `ip_queue` module for further processing by a user space process
- `NF_REPEAT`: the module must be called again.

Netfilter can pass packets to a process in the user space through the `ip_queue` module from any of five hooks. The user process must return one of the five verdict values to `ip_queue` module which pass

it to Netfilter. The process uses `libipq` library to communicate (read a packet, set mode etc.) with `ip_queue` module through `netlink` socket.

`iptables` [16] is a tool in the user space that allows the user to configure (add, delete, change rules) Netfilter. A rule includes a packet flow specification and an action that indicates what to do with the packet flow. The packet flow specification makes use of the information in the packet header such as source, destination addresses and ports, and types of protocols.

Netfilter is limited because only one process in the user space can receive packets from the kernel. `ipqmpd` [21] adds the possibility of passing packets from different flows to different user processes by using a daemon in the user space called `ip_queue` multiplexer daemon. It communicates with different user processes using sockets or other IPC mechanisms. That is inefficient, because packets must re-enter the kernel before arriving in the user process.

To obtain good performance of our active node, we have modified Netfilter to pass different packet flows directly to the right user process without going through the multiplexer daemon. Each rule in our version of Netfilter has an extra field called PID (process ID) to identify the user process that wants to receive packets matching the rule. Netfilter passes this information to `ip_queue` module which supports multiple queues, one queue per user space process.

Table 1: Modes in `ip_queue` module

Mode	Behavior	Version
`IPQ_COPY_NONE`	Initial mode, packets are dropped	Original
`IPQ_COPY_META`	Copy the metadata to the user process; keep another copy in the kernel	Original
`IPQ_COPY_PACKET`	Copy the metadata and the packet payload to the user process and keep another copy in the kernel	Original
`IPQ_PASS_PACKET`	Pass the metadata and the packet payload to process in the user space; do not keep any copy in the kernel	New

When installing a packet filter, a user process provides a packet flow specification and its PID. As this

rule passes through `iptables` module in the user space, we have modified it to take into account the process PID. When Netfilter passes packets to `ip_queue`, it presents the PID of the process interested in these packets. `ip_queue` forwards the packets to the right process.

We have also modified `ip_queue` module to support more than three modes of operation—standard `ip_queue` module always keeps a copy of a packet passed to a user space process and we needed different behavior. Table 4.1 summarizes the original and extended modes. Proactive services often require `IPQ_PASS_PACKET` mode. When a packet is passed to a user space process in this mode, it uses a new verdict value called `NF_INJECT` to inject the packet into the kernel after processing.

Our version of the modified Netfilter for Linux 2.4.16 currently supports 40 queues in `ip_queue` module. This version of Netfilter and `iptables` module are available for downloading at `http://drakkar.imag.fr/~nguyenb/Netfilter`.

4.2 Proactive services

A proactive service is an executable program written in a general purpose language such as C, C++, or Java. In case of Java, we provide a JNI (Java Native Interface) API that allows Java programs to use `libipq` library. This version of the Java API for `libipq` is also available for downloading at the already mentioned URL.

A service instance is a child process of the control module and it is activated on behalf of a user authentified by the control module. Upon activation, the service will pass a packet flow specification to Netfilter via the modified `iptables` module when appropriate.

A proactive service may be interested in events generated by a monitor when the state of the node changes. It can discover what are monitors available at the node and register with a given monitor. As a service need to continuously receive packets from the kernel and wait for events from monitors, services are implemented as multithread processes having at least two threads. One thread is responsible for communication and the others are for processing.

A service can be controlled by the user who can send new parameters to its service via the control module that authenticates user requests and passes the parameters to the right service by means of

events.

Each service has a unique name which is stored in a service database along with some information about management and security policies. We use the process PID of the service to form the name needed by the `AF_UNIX` socket for communication between the service and the monitor. Thus, each service process has a unique communication endpoint, so that the control module and monitors can distinguish between different service instances of the same service activated by different users.

We have implemented two services described in Section 3.2. Table 4.2 presents the event subscription command and the event data structure used by the caching service.

The video adaptation service uses the Dalí [17] MPEG C library to dynamically adapt a video flow to the available bandwidth of the radio channel in a 802.11 wireless LAN.

Table 2: Event description for the caching service

`Command = SUBSCRIBE`	`Command = EVENT`	`Command = UNSUBSCRIBE`
`From=1470` *service process PID*	`From=presence-monitor` *monitor name*	`From=1470` *service process PID*
`HostName=marie.imag.fr`	`HostName=marie.imag.fr`	`HostName=marie.imag.fr`
	`State=disconnected`	

4.3 Control module

This module is implemented as a daemon process that handles requests for service activation from the user. It is responsible for user authentication via a user database that stores information about user accounts. Upon user request for service activation, it looks up the service in the database, verifies the access rights of the user, and activates the service: it forks a child process. It changes the effective UID of the child process, which is 0 (the control module run as the root), to the user ID. The user ID is used for accounting resource usage: activated services, storage quota, CPU usage etc.

The child process replaces then its process image with the proactive service code with the `execpv` command and waits for `CAP_NET_ADMIN` capability that is sent from the control module. The service requires such capability for reading packets from the kernel by means of `netlink` socket. When the

service process dies abnormally, a signal is sent to the control module that takes over to release any resources and clean the packet filter installed in Netfilter.

4.4 Monitors

A monitor is responsible for monitoring the state of the environment, for example CPU load, available storage space, link congestion, queue size etc. Each resource has its own data and measurement units. A service can choose a resource to monitor and subscribe to an appropriate monitor. A monitor usually manages a set of services that are interested in a given resource. Communication between services and monitors is asynchronous and is implemented using `AF_UNIX` sockets. When a monitor detects a change in the state, it sends a notification to the service that decides what to do with the event. Monitors are identified by means of unique names. They use them to form the names needed by their `AF_UNIX` sockets. Monitors have also at least two threads: one for communication (receiving subscription commands and sending events) and others for processing.

5 Performance

To evaluate the impact of our Linux implementation of the active node architecture for dynamic association of proactive services with data flows, we have measured the performance of packet forwarding and passing them to the user space on a 800 MHz Pentium III PC with 128 MB RAM running Red Hat 7.2. Figure 4 presents the delay of packet forwarding in function of packet size for two cases: in the first one, packets enters the kernel and they are just forwarded to the destination (no proactive service installed); in the second case, a packet filter is installed to intercept packets and pass them to a proactive service (active flow - proactive service installed). It does not perform any processing and just re-injects packets into the kernel for further forwarding. The difference between the two curves represents the overhead of passing a packet to the user space. These results show that when a service does not install a packet filter, data flows do not incur any performance penalty.

We can also see from the figure that in the second case the overhead has only impact on data flows on which proactive services need to perform useful processing: the delay for a passive flow (the flow for

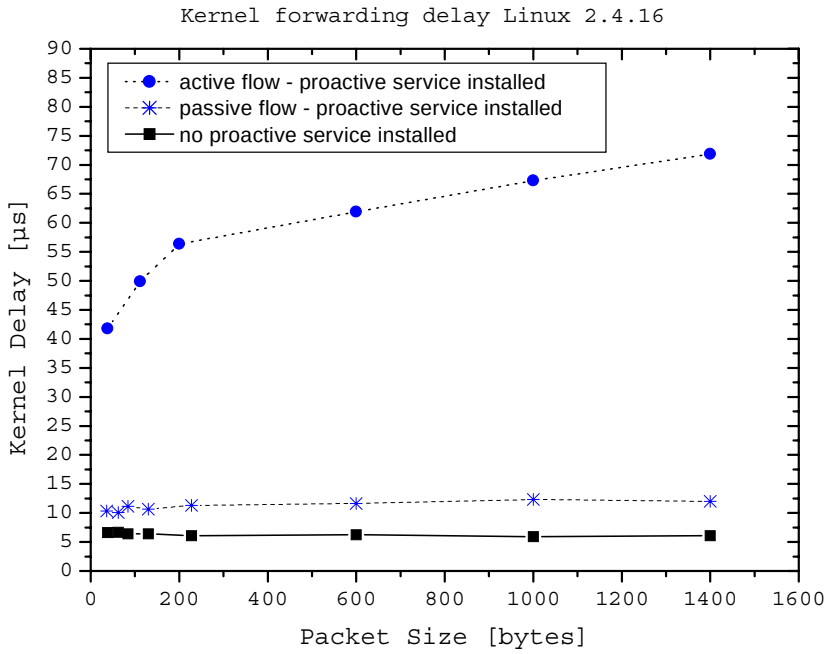

Figure 4: Performance of passing packets to the user space

which packets are not intercepted by its packet filter) stays small even if the packets of an active flow are processed by the associated proactive service.

6 Conclusion and Future Work

We have presented an active node architecture for supporting proactive services. The main idea of proactive services is to provide a means for taking into account adaptability to varying conditions and flexible supervision by the application or the user. Proactive services intercept packets of chosen data flows. Installing and uninstalling a filter for intercepting packets are done dynamically, so that other flows, which do not need any additional processing, are not subject to any performance penalty.

We have prototyped the active node on Linux. A modified Netfilter module delivers matching packets directly to the right user process implementing a proactive service. Measures of packet forwarding show that deviating packets to the user space for processing incurs a slight overhead, however it only concerns data flows that require specific processing by proactive services.

Our architecture and its prototype open several interesting research directions. First, we want to provide a high level language for specifying proactive services. It should allow to reason in terms of monitors, events, and actions. The specification would be compiled into the implementation language like C, C++, or Java.

Second, we are interested in experimenting with proactive services for different applications such as Content Distribution, Custom Data Flow Adaptation, and Sensor Networks. These kinds of applications call for a generic tool for specifying and parsing the contents of application level protocols. We can integrate it as a library to be used by proactive services to analyze the contents of packets and process them in a right way.

Last, we would like to integrate the other active model (based on active capsules) for custom deployment of proactive services on several nodes and for flexible management of our active nodes. We work on a specialized scripting language for programming such management capsules.

Acknowledgments

We would like to thank Paul Starzetz for his help in implementing the active node on Linux. This work has been partially supported by France Telecom R&D.

References

[1] D. Alexander et al., "Active Bridging", In Proceedings of SIGCOMM '97, September 1997.

[2] E. Amir, S. McCanne, and R. Katz,"An Active Service Framework and its Application to Real-time Multimedia Transcoding,", ACM Communication Review, vol.28, no. 4, pp.178-189, Sep. 1998.

[3] S. Bhattacharjee, "On Active Networking and Congestion", Tech. Rep. GIT-CC-96-02, Georgia Institute of Technology, Atlanta, Georgia, 1996.

[4] S. Bhattacharjee, K. Calvert, and E. W. Zegura. "An Architecture for Active Networking". Proc. High Performance Networking (HPN'97), White Plains, NY, April 1997.

[5] S. Bhattacharjee, and M. W. McKinnon, "Performance of Application-Specific Buffering Schemes for Active Networks", Georgia Institute of Technology, Atlanta, Georgia, 1998.

[6] M. Bossardt et al., "Service Deployment on High Performance Active Network Nodes". IEEE Network Operations and Management Symposium (NOMS 2002), Florence, Italy, April 2002.

[7] A. Boulis, P. Lettieri, and M.B. Srivastava, "Active Base Stations and Nodes for a Mobile Environment", ACM 1998.

[8] K. Calvert. "Architectural Framework for Active Networks", July 1999. http://protocols.netlab.uky.edu/~calvert/arch-docs.html

[9] A.T Campbell et al., "A Survey of Programmable Networks", *ACM Computer Communications Review*, April 1999.

[10] D. Decasper et al., "Router Plugins : A Software Architecture for Next Generation Routers", Proceedings of the ACM SIGCOMM '98, Vancouver Canada, September 1998.

[11] M. Fry and A. Ghosh, "Application Level Active Networking", Computer Networks 1999.

[12] M. Hicks et al. "PLAN: A Packet Language for Active Networks", Proceedings of the International Conference on Functional Programming (ICFP)'98.

[13] K. Ksounis, "Active Networks: Applications, Security, Safety, and Architecture" , IEEE Communication Surveys Magazine, 1st quarter, '99.

[14] A. B. Kulkarni and G. J. Minden, "Active Networking Services for Wired/Wireless Networks", Eighteenth Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM'99), New York, March 1999.

[15] The netfilter/iptables project : http://netfilter.samba.org/

[16] R. Russell, "Linux iptables HOWTO", http://www.linuxguruz.org/iptables/howto/iptables-HOWTO.html

[17] W-T. Ooi et al., "The Dalí Multimedia Software Library", 1999 SPIE Multimedia Computing and Networking, 25 - 27 Jan 99, San Jose, CA.

[18] B. Schwartz et al., "Smart Packets for Active Networks", The Second IEEE Conference on Open Architectures and Network Programming (OPENARCH'99),New York, N.Y. USA, March 26-27, 1999.

[19] D. Tennenhouse "Proactive Computing", *CACM*, May 2000.

[20] H. Welte, "The Netfilter Framework in Linux 2.4", http://www.gnumonks.org/papers/netfilter-lk2000/presentation.html

[21] H. Welte, "ipqmpd - ip queue multiplexer daemon", http://www.gnumonks.org/projects

[22] D. Wetherall, U. Legedza, and J. Guttag, "Introducing New Internet Services : Why and How", *IEEE Network Magazine*, July 1998.