Design and Implementation of a Python-Based Active Network Platform for Network Management and Control

Florian Baumgartner[1], Torsten Braun[2], and Bharat Bhargava[1]

[1] Department of Computer Sciences and Center for Education and Research in Information Assurance and Security (CERIAS), Purdue University, West Lafayette, IN 47907, USA
baumgart|bb@cs.purdue.edu,
[2] Institute of Computer Science and Applied Mathematics
University of Berne, Bern, Switzerland
braun@iam.unibe.ch

Abstract. Active networks can provide lightweight solutions for network management-related tasks. Specific requirements for these tasks have to be met, while at the same time several issues crucial for active networks can be solved rather easily. A system addressing especially network management was developed and implemented. It provides a flexible environment for rapid development using the platform-independent programming language Python, and also supports platform dependent native code. By allowing to add new functions to network devices it improves the performance of Internet routers, and simplifies the introduction and maintenance of new services.
To show the capabilities of the approach, two different quality of service related applications, that is a simple multicast algorithm and an approach to automatically set up tunnels, have been implemented. The evaluation of these services shows the advantages of the architecture, and its benefits for the task of network and quality of service management.

1 Introduction

The limitations of current management mechanisms to provide an adaptable system to configure and control services within the network of Internet Service Providers (ISP) enforce the development of new strategies for network management.

In addition to a distributed resource management, mechanisms are needed to add new services dynamically to the network. For competing ISPs the time needed to provide a new service to their customers is essential. This is why an architecture allowing the dynamic and quick establishment of new Internet services is important.

The most obvious way to provide such flexible systems is the exchange of program code between network nodes. The program code is executed by a network device, reconfiguring the router, collecting information or sending data and program code to other devices. Networks providing such mechanisms are called active networks. Such an approach is even more powerful since the differences between classical network management and signaling are becoming smaller. Active networks can not only be used to provide a high degree of control over the network but can easily be used to implement

lightweight signaling mechanisms, needed for the control of large networks. There are several types of active networks.

The capsule approach, as presented by Tennenhouse [19], uses packets consisting of a short piece of code and of additional payload. An active router receives the capsule and executes the code. This code may simply be used to route the capsule to the destination or to configure the active router on the path. Another approach focuses more on a programmable network [7] and proposes a more moderate active networking model. Active network mechanisms are used for signaling, configuration and monitoring purposes [11].

The execution of code on network devices causes several problems. Especially with the capsule approach security and performance are still open issues, but even within moderate active networks, the security issue is not solved yet. Murphy [14] shows the security problems and proposes an architecture for active networks. A different approach to improve security was proposed by Brunner [6], who suggests the creation of Virtual Active Networks. Similar to virtual private networks (VPNs), a customer can ”rent” such a virtual active network. Within its active network, the customer (e.g. a company) can exploit AN technology but has no access to the other virtual networks.

The goal of the approach presented within this paper is to provide an easy to use, lightweight mechanism with a focus on management related tasks. In contrast to systems with a broader approach, this reduces the complexity of several typical active networking problems, like security and performance. On the other hand, such a system requires a more direct interaction between the IP router and the active components. Instead of providing a distributed application platform, which mainly uses the IP network as a transport medium, the system has to change and modify router functions, add new components, or has to be able to influence packet processing within the router core. This requires lightweight but powerful and flexible interfaces to native code and functions as well as a high level programming language to support the rapid development of configuration scripts.

2 The Python-Based Active Router

Various languages have been discussed and used for the implementation of active networking systems [18]. While PLAN [13] is based on a functional language like Caml [8], others, like the Active Network Transport System (ANTS) [22], are based on Java. In contrast to those, the active networking architecture proposed here was developed especially for the purpose of network management and therefore uses the Python [17] language, which provides certain advantages for that kind of application.

Python as a Language for Active Network Systems

Python meets the properties of most modern, interpreted languages, used for active networks. It provides portable bytecode, which allows an execution on different platforms; restricted execution environments to keep active packets from damaging the environment; and threads.

One of the strengths of Python, which makes it especially useful for configuration related tasks, is its extensibility. In contrast to Java with its native interface (JNI) [7] or other languages, the integration of native code modules has been a central aspect of the Python programming language from the beginning. This is also the reason why many applications use Python as a configuration and control front-end and rely on native code for time consuming tasks. The functional separation used by these programs matches perfectly the situation we face on an active IP router, with high speed packet processing in the kernel and an interpreted language on the control plane.

Another argument, which led to the use of Python instead of Java or Caml, was the type of programs we expect to be sent over the network. Similar to system administrators, automating certain tasks, network administrators have to be able to write short configuration scripts and send them through the network. Since these scripts will highly depend on the current network state and on a specific task, the capacity of a language to support rapid development is important. Python provides high-level data structures and dynamic types, which support the rapid development of configuration scripts. Since more generic systems face security and performance problems, Alexander [2] recommends different, or even contrary properties for active networking languages. As will be shown in the next section, the specialization of the PyBAR system reduces this problems, and therefore allows to benefit from Python's prototyping capabilities.

Due to its high level data structures and dynamic types, Python programs are short. Their source code is usually three to five times shorter than comparable Java sources [20] and also the bytecode used by the PyBAR consumes significantly less memory. Therefore it is possible, similar to special purpose languages like Sprocket or Spanner [18], to transmit reasonable programs within a single Ethernet packet. This simplifies the transmission of active packets crucially, since no fragmentation/reassembly and reliability mechanisms are necessary.

A general comparison between Python and other languages is presented in [20]. For a detailed performance evaluation of Python and Java see [12]. The Python based active router (PyBAR) system has been implemented for Linux and for Virtual Routers [3]. A Virtual Router is an IP router emulator, which can be combined with real networks. This allows real routers and hosts to be integrated into an emulated network.

2.1 The PyBAR Architecture

The design of the active network platform tries to separate the active components as far as possible from the conventional router functionalities, but provides access to routers internal components, like traffic conditioning systems, packet filters and routing. Such a separation ensures portability and also allows an easier integration of existing devices. Figure 1 shows the general architecture of the system.

The PyBAR architecture is based on the standard Python virtual machine and can be connected to several network nodes (e.g. routers). In addition to a rather thin NodeOS (platform adaptor) layer written in C++, the system consists of a set of native or interpreted library and extension modules and a central core written in Python to execute received code. Received packets are forwarded either to a specific service handler, provided by an extension module, or are processed by the core.

The NodeOS provides communication facilities for the PyBAR core and the extension modules. Instead of running merely on top of the IP router, the NodeOS provides several interfaces to the routers, using Python's capability to use native code. This allows the addition of new functions like traffic conditioning, encapsulation or monitoring components directly to the IP routers kernel.

Security Since the system is only used by network administrators and their management tools, security issues are less complex to solve than in more generic environments [14][11]. The question of whether a platform is trustworthy or whether a secure bootstrap mode [2] is supported, is of less importance, since the devices are owned and controlled by the same network provider.

The basic mechanism used by the PyBAR platform is a public key infrastructure, which can be used to sign and encrypt active packets. For that purpose the system provides a set of encryption mechanisms, which are provided by the PyRSA extension module allowing the flexible use of high speed cryptographic algorithms. Cryptographic mechanisms require high processing speed, and the module therefore was implemented in C. This solution provides high usability with good performance.

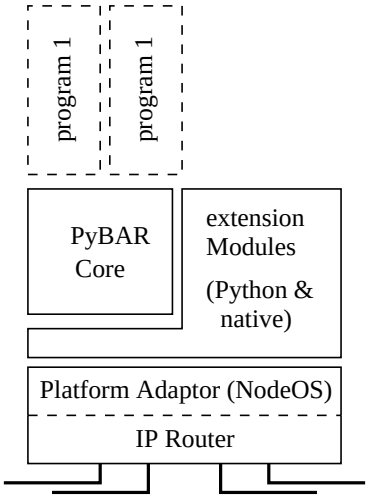

Fig. 1. Basic architecture of the PyBAR system.

Since active packets can be modified during transmission, authentication for active packets is complicated [14]. A PyBAR active packet consists usually of several parts (code blocks), which allows sensitive (e.g. executable) code blocks to be static and uses other code blocks for the volatile data [11]. Additionally, within the PyBAR system active packets can be forwarded within a specific Differentiated Service class. Access to this Differentiated Services class can be limited to devices operated by the network provider itself. By setting up filter mechanisms at border routers, a modification of active packets can be prevented. This approach is similar to the Virtual Active Networks [6], but uses a Differentiated Services class instead of a VPN-like mechanism to control the access to the active network.

Code Transport The advantage of active networking regarding network management is that code can be transported along the path with normal data traffic. Therefore, programs can become active at nodes requiring configuration without keeping central structures like topology databases.

– A packet has one of the router addresses and is forwarded to the PyBAR for further processing. This mechanism is especially useful to address specific end systems or to send an active packet directly to a special active router.

– The Router Alert option is an IP option, indicating that a router should treat the packet in a special way and can be used to trigger active packet processing [18]. The Router Alert option was introduced in conjunction with the RSVP protocol and has the purpose to send a packet to a specific destination and trigger certain functionalities in the routers along a certain path.
– Some IP routers process packets containing IP options like Router Alert more slowly than normal packets. To prevent a delay of active packets, a special Differentiated Services Code Point [4] value can be used instead of the Router Alert option. Active routers forward these packets to the active components while conventional devices will simply ignore this DSCP value. Besides improved performance this has other advantage. Since packets with such a DSCP can be handled preferentially by Differentiated Services routers, the loss of active packets can be ignored, and limiting the use of that DSCP to devices of the network provider reduces security risks. Of course multiple DSCPs may be used as well. As an example, there might be a DSCP value for active signaling packets and another for active packets carrying data.

Extension/Library Modules Since the hard coded commands provided by the PyBAR cover only very fundamental tasks, and the availability of functions may depend on the platform, more complex issues have to be covered by extension modules.

Extension modules can be provided by native code or by Python. Since the integration of native code and libraries is a fundamental mechanisms in Python, already existing libraries can be added without much overhead, similar to the PyRSA module. Of course, an appropriate module for the current platform has to be available, but since there is no difference between calling functions provided by a native module, or using an interpreted Python module, a platform-independent Python version of a module may be provided as a default.

Any of these modules can be added, replaced, or removed dynamically by active network mechanisms. A high-level, interpreted extension module to provide a uniform interface to set up resources will be presented in Section 2.2.

Packet Processing and Code Execution The PyBAR core is responsible for the treatment of received packets containing code. The active service id field in the packet header signals the platform adaptor which active packet type is received. While packets containing executable code are forwarded to the core, the processing of other packets is left to active service handlers. Comparable to router plug-ins [9], modules can be installed to process certain packets, identified by an active service id.

The processing of executable packets is much slower and time consuming than the simple forwarding of a packet's payload. Furthermore, the treatment of such streams can be accomplished completely by the NodeOS and specialized extension modules. If the service handler is provided by native code, the complete processing of such a packet is "Python-free" and therefore reasonably fast.

If a packet contains a program to be executed, the entire packet has to pass consistency and security checks. While in the first step basic properties of the packet are controlled, the second one covers digital signatures and encryption related issues. After

the packet has passed the consistency and authentication tests, an execution environment is initialized and the code within the packet is executed.

The core is Python-based and mainly a kind of framework, which can be adapted to different needs and to provide very different mechanisms. Typical functions provided by the core are:

- Monitoring of a running code. This monitoring provides no absolute security, but merely limits the damage caused by program errors.
- A small central database or stack is provided to allow capsules to store and exchange data.
- Functions to install and replace modules within the library are provided.

In the description of the Python language the ”high level” of the Python language was mentioned. Even if some of the tasks, like cryptography, require rather complex algorithms, the core itself is very simple and uses only some high-level data structures and functions provided by those modules.

Packet Format Packets, which contain executable code, can use the same basic header as packets, which only have to be processed by some active component. To avoid as much overhead as possible, this header is much simpler and shorter than an ANEP header [1]. This increases the performance of packet processing by service handlers, and leaves the processing of more sophisticated header fields, which are required for executable packets, to the Python-based core.

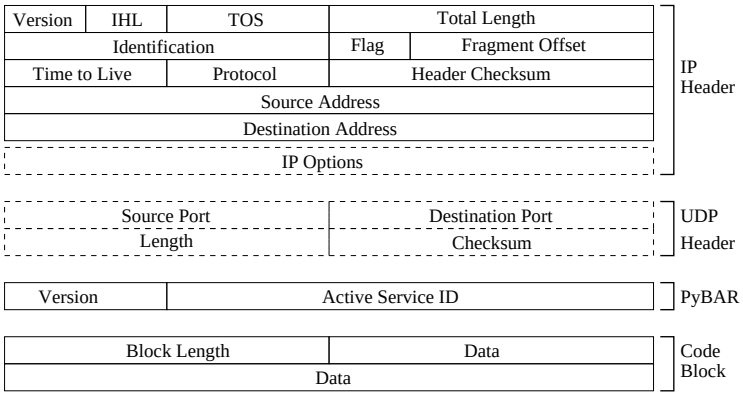

Fig. 2. PyBAR IP/UDP payload and a PyBAR code block. Several code blocks might be attached to a packet.

The minimum required PyBAR header is exactly four bytes long (see Figure 2) and can be encapsulated either in raw IP or in UDP packets. It only contains an eight-bit wide field with version information and a 24-bit wide active service identifier. The latter

allows to notify which module of the PyBAR system shall process the packet's payload. Since this header is very small and simple, a stream packet has to be checked only for the active service id and can be forwarded to the corresponding module responsible for this type of service.

The special active service identifier is used if the rest of the packet contains executable code. If the active service identifier signals an "active" payload the rest of the payload has to consist of at least one code block. A code block consists of a 16 bit wide length field and the code block's content only. Multiple code blocks may be contained within a packet. The PyBAR system expects the first code block to contain executable code. Besides the program, this code block also contains a signature and other information. However the payload format of the first code block is completely handled by the PyBAR system itself and can therefore be adjusted easily. The other code blocks are not processed and it is left to the code in the first code block to handle them. The platform adaptor provides mechanisms allowing to access and manipulate the other code blocks in the packet. This includes the execution of code blocks and their installation as PyBAR library modules.

2.2 Differentiated Services Support

A central problem within the Internet is the lack of a homogeneous configuration interface to network devices. Dependent on the vendor, very different interfaces are provided. Even more problematic than the interfaces are the fundamental design differences of devices [21].

A good example for such different design concepts are traffic conditioning components, which are rather different on various platforms. Therefore, a simple mapping of configuration commands might not be feasible. The Linux traffic control system uses a concept of nested boxes. Each box is a traffic conditioning component, which can contain other components. Other implementations are more list oriented or use a graph-like layout of their traffic conditioning systems like the Virtual Router does.

These different concepts prohibit the use of low-level configuration commands within a network. In contrast to providing a general application programming interface [15] the PyBAR simply offers different commands for different types of platforms. Since a capsule might be executed on multiple hosts and a distinction among the different command sets is not feasible, those low-level commands are usually not used but library modules are installed on the system in advance. A module providing methods to set up and maintain Differentiated Services on a router can map a command set to the low-level configuration scripts. Several high-level functions can be provided by such a module:

- Functions to initialize and configure the basic system have to be provided. Some kind of init() function may set up the complete set of queues, schedulers and classifiers needed to provide Differentiated Services. Parameters of the init function may define whether an ingress, egress or intermediate router has to be set up.
- The marker mechanisms have to be configured to mark packets with different DSCPs. Functions to add or remove flow descriptions at the ingress routers have to be provided. The mechanisms to apply such flow descriptions to the router may vary significantly. Therefore, a high level interface would be beneficial.

– Since the Differentiated Services concept requires a separate handling of the different traffic classes, each traffic class has to be configured for a certain share of the link bandwidth. Therefore, also these parameters are important for a general interface.

Obviously, such a description can never meet all aspects of Differentiated Services or of any other Quality of Service- providing mechanism. Therefore the PyBAR does not even try to provide such a general interface or even a multi-platform module providing such a functionality.

Therefore, a module integrating DiffServ configurations for Virtual Routers and for Linux was implemented, providing a convenient small set of commands as listed in Table 1. The module is written in Python and can rather easily be extended to provide more control and advanced features. This is important, because the set of commands provided by an appropriate Differentiated Services module depends on the services an Internet Service Provider wants to provide and control.

init()	sets up the complete traffic conditioning components required for DiffServ, with a an appropriate scheduler, EF and AF queues, token bucket filters
setClassShares(...)	configures the bandwidth shares for the different traffic types
mark(...)	configures the Differentiated Services marker to mark specific flows with certain DSCPs
unmark(..)	removes a marker rule

Table 1. Commands provided by the DiffServ module for the configuration of Differentiated Services resources.

This explains why a lightweight mechanism to install modules on different platforms and to adapt extension modules for new purposes is much more important than the attempt to provide a really generic interface. The mechanisms to install such modules are provided by the PyBAR platform. An active packet may transport and install code within each suitable node of a network, on a single device only, or on any machine along a certain path. Even the attachment of multiple code objects is possible, whereas each code object is to be installed on the matching router hardware.

3 Adding Active Services to a Network

The active service id field in the PyBAR header provides an easy mechanism to add various packet treatments. Since the processing can completely be provided by native code, the required processing power stays reasonably low, even for more complex algorithms. A possible application for such active services can be the support for video applications or the support for management related tasks, like a framework to establish tunnels within a network, or simple multicast mechanism for small groups.

3.1 Active Tunnel Establishment

IP tunnels are an enabling technology for the application of new services and network management. Even if the basic mechanism – an encapsulation of a packet into another packet (IPIP) as proposed by [16] – is simple, more complex functions may be realized:

- The encryption and decryption of packets at the tunnel start and end points can provide end-to-end security. This way flows can be transmitted securely without the danger that an untrustworthy service provider might eavesdrop.
- Since all packets transmitted through a tunnel get a new header at the tunnel start and the tunnel end point addresses, traffic conditioning mechanisms can be applied quite easily for all packets in the tunnel, even if the encapsulated packets have different source and destination addresses.
- Tunnels allow transport of signals transparently through a network and keep them from triggering mechanisms within an ISP (e.g. an RSVP reservation setup).

For the establishment of tunnels, appropriate start and end points are required, if certain special services like encryption have to be provided. But even for a simple IP in IP encapsulation, an end point needs to be capable of handling the decapsulation of packets.

Since the configuration of a tunnel is usually sender-driven, the general problem is to detect an appropriate end point. When establishing a tunnel between border routers of an ISP, the ingress border router does not usually know the address of the egress border router. In contrast to other approaches using active networks to establish tunnels [13], the focus here is not to provide tunneling mechanisms by active code itself, but on finding appropriate end points and setting up encapsulation mechanisms within the IP router. The active components are only involved for the instantiation of the tunnel, and leave the encapsulation of packets to the IP router.

An active network allows triggering the establishment of tunnels automatically without the need for of any additional protocol. Since an active packet allows the definition of a kind of ”search pattern” for an end point and can simply be sent downstream towards the destination, the establishment of a tunnel can be simplified and also the exchange of keys might be accomplished.

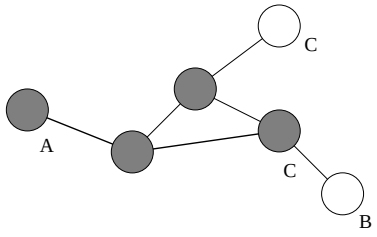

Fig. 3. A network with routers capable to tunnel packets (grey) and routers without that mechanisms (white).

Figure 3 shows a small network with a set of grey routers capable of handling tunnels and white end systems without this support. Node A wants a tunnel to be established as close as possible to point B. To find out an appropriate end node, an active packet addressed to node B can be injected into the network. The packet will pass the nodes along the path and check whether the node is an appropriate tunnel end point or not. Similar to the traceroute program reporting each passed router, the active packet can report each possible candidate for a tunnel end point to node A. If simple IPIP tunnels, not requiring a specific setup of the tunnel end point,

```
class DiscoverEP(ARpacket):
    def __init__(self,acpkt):
        # get a list of router properties/services
        c=pad.getCaps()
        # if IPIP available, extract information from
        # code block and send feedback packet
        if c.count('IPIP'):
            src_info=cPickle.loads(acpkt.cb(1))
            # generate and send feedback packet
            p=pad.UDPPacket()
            p.source=pad.hostip
            p.dest=src_info['tunnel_start']
            p.destport=src_info['portnumber']
            p.payload=cPickle.dumps({'service':'IPIP',
                'tunnel_end':pad.hostip, 'time':pad.time})
            p.send()
        # forward original active packet
        acpkt.send()
        return
```

Table 2. Active Packet code to discover a device, able to handle IPIP tunnel endpoints.

are used, node A simply uses the most appropriate candidate as the end point and sends the encapsulated packets to this address.

Table 2 shows a the active packet's Python code to check whether a device can handle an IPIP tunnel. The whole packet consists of the first code block containing this executable program and a second code block containing information about the tunnel start point (src_info) like the address and the port number to which the feedback packet has to be sent.

To ensure that such a configuration packet was sent by an authorized network node, this exchange of active packets takes place within a Differentiated Services class, which is accessible only for specific nodes and prevents a transmission of active configuration packets from outside that network. Additionally, such a packet carries a signature, which can be checked periodically.

This search for an end point is not only determined by the simple capability to provide the decapsulation of tunneled packets, but may also be used to find nodes capable of certain encryption techniques.

3.2 A Simple Active Multicast Service

Another example for an application using active services is the simple active multicast mechanism, which is useful to support multicast for small groups. The active service id signals the existence of additional IP addresses in the packet's payload.

A similar but not active multicast service for small groups (xcast) has been proposed by Boivie et al. [5]. An active approach to distribute multicast packets using functional language elements of the PLAN system is described by Hicks [10]. In contrast to the approach using PLAN, the algorithm presented here conforms better to the current xcast approach. The main difference between the approach presented here and xcast, is the

way an xcast packet is signaled. While xcast uses a dedicated destination address, this algorithm uses the active service id to trigger the special packet treatment. Especially if a native service handler is provided, an active router running the PyBAR system can easily be updated to provide a xcast mechanism at reasonable speed.

Figure 4 illustrates the general concept. A server has to send identical data to multiple addresses. Instead of transmitting one packet via unicast to each destination, one packet is sent, containing multiple addresses. An active router on the path detects the packet due to a special DSCP values, checks the active service id, and extracts the list of addresses. If all addresses are routed over the same interface, the original packet is forwarded, otherwise the packet is converted into several packets, each carrying the addresses of the clients routed over the same interface. The resources saved in comparison to unicast are evident.

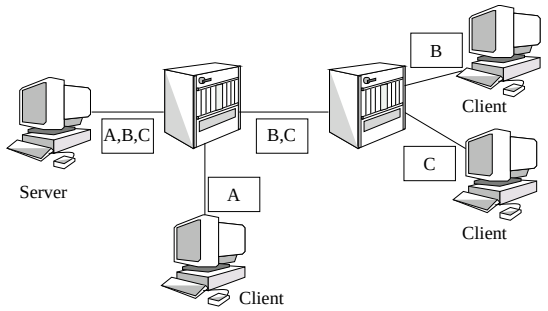

Fig. 4. Processing of a packet with multiple addresses.

Table 3. The complete code of the service handler providing a simple multicast service.

```
class MiniMulticast(ARservicehandler):
    def forward(self,pkt):
        # extract address list from first code block
        alist=cPickle.loads(pkt.cb(0))
        ifcs={}
        # scan address list and create interface/address-l ist structure
        for i in alist:
            interface=pad.queryRoute(i)['if']
            if not ifcs.has_key(interface):
                ifcs[interface]=[]
            ifcs[interface].append(i)
        # scan interface/address-list structure and send packets
        for i in ifcs.keys():
            p=pad.Packet()
            p.cb(0)=cPickle.dumps(ifcs[i])
            p.cb(1)=pkt.cb(1)
            p.send({'dest':ifcs[i][0], 'ptype':pkt})
        return
```

Two extension modules providing the appropriate service handler have been implemented. While the first one uses Python only, the second one is platform dependent using C++. Table 3 lists the complete Python code required for the service handler. An active packet for installing this service handler will usually consist of a short script to register the service and the service handler code itself.

The example uses the PyBAR packet format. A multicast packet consists of two code blocks *cb(0)* and *cb(1)* (see Table 3). The first code block contains the address list, the second one the payload of the packet. As a simplification, in the example the addresses are realized by a Python list, but also the more generic xcast format could be used. The addresses within this list are scanned and for each address it is checked, which interface would be used to reach it. As intermediate results, a data structure is created and filled with addresses for each outgoing interface. In a final step new packets are generated and transmitted. The *pad.** functions are provided by the platform adapter. Once a packet for a specific service handler arrives, the *forward()* function of this service handler class is called with the packet as a parameter. The *forward()* function extracts the address list from the first code block of the packet (*cPickle*), scans the addresses, sets up the dictionary and finally creates and sends the new packets. The destination address of a new packet for a certain interface is chosen from the list of addresses routed over that interface.

Table 4 shows the performance[3] of the Python and the C++ based service handlers for different numbers of addresses. The bandwidth calculation was based on the execution speed of the Python code and an average packet size of 1000 bytes. For the output bandwidth it was assumed that each packet is split up completely, and each address has to be routed over a different interface.

nr.of addresses	time [ms]	packet rate (in)	packet rate (out)
Python module			
4	1	1000	4000
8	1.7	580	4640
16	2.2	454	7264
Native extension module (C++)			
4	0.014	> 10000	-
8	0.026	> 10000	-
16	0.05	> 10000	-

Table 4. Performance of the simple multicast service provided by service handlers written in Python and in C++.

The packet rates reached by the native extension module could not have been measured exactly, but they were more than sufficient to process incoming packets at a full link bandwidth of 100 Mbps. The performance of the Python example can not compete with the implementation using the native code but could being improved by using a more sophisticated format to store the destination IP addresses.

[3] Measured on a Linux 400 MHz Pentium II

4 Summary

The Python-based Active Router (PyBAR) system is a lightweight active networking platform focusing on the tasks of network management and configuration. The system provides flexible access to router resources and forwarding mechanisms, which is necessary to provide quality of service or to configure devices. Both platform-independent and native extension/library modules are supported, and allow, if a native module is available, to perform packet processing without the limitations of interpreted code.

The system provides basic security by using standard encryption and authentication mechanisms. Additionally active packets can be transported within a specific Differentiated Services class, which prevents unauthorized injection of active packets, and also supports a preferential treatment of active packets.

The support for rapid development, provided by Python, is pursued by ongoing work to combine Python and Java with the goal to be capable to use prototypical Python programs from within Java applications, and directly compile Python programs to Java bytecode. These prototyping capabilities allow a rapid development of configuration scripts, which is important for the usability of the system. Additionally, the high-level character of the language reduces the program size and improves the system performance.

The capabilities of the approach were demonstrated by two classical active network applications. Using the system for the establishment of tunnels, illustrates how to automate and distribute configuration tasks with PyBAR. Active packets can be used to set up tunnel endpoints according to certain specifications. The second example is a multicast mechanism for small groups, which shows the advantages of installing new services on the network, and serves for a short performance comparison of a platform independent Python approach and an implementation using native code.

5 Conclusions

Seamless integration of native code modules within the PyBAR provides great advantages for the task of network and quality of service management. Resources within the network devices can be accessed and native code can be used more easily than using languages like Java. This allows the easy installation of new traffic conditioning components. The capacity to provide packet processing by the native code only allows to implement services requiring high performance. This encourages the implementation of native mechanisms for the PyBAR, instead of the underlying platform itself, increasing the flexibility and portability of new services.

Acknowledgment

The work described in this paper is a part of the work done at the University of Bern in the project 'Quality of Service Support for the Internet Based on Intelligent Network Elements' funded by the Swiss National Science Foundation (project no 2100-055789.98/1) and the SNF R'Equip project no. 2160-53299.98/1. This work was also supported in part by the NSF grant EIA 0103676.

References

1. D.S. Alexander. Active network encapsulation protocol. CIS, University of Pennsylvania, http://www.cis.upenn.edu/ switchware/ANEP/, August 2002.
2. D.S. Alexander, W.A. Arbaugh, A.D. Keromytis, and J.M. Smith. A secure active network environment realization in switchware. *IEEE Network*, 12(3):37–45, 1998.
3. F. Baumgartner and T. Braun. Virtual routers: A novel approach for qos performance evaluation. In Crowcroft e. al., editor, *Quality of Future Internet Services*, LNCS, pages 336–347. Springer, 2000. ISBN 3-540-41076-7.
4. S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weis. An architecture for differentiated services. Internet Standard RFC 2475, December 1998.
5. R. Boivie, N. Feldman, Y. Imai, W. Livens, D. Ooms, and O. Paridans. Explicit multicast (xcast) basic specification. Internet Draft draft-ooms-xcast-basic-spec-03.txt, June 2002. work in progress.
6. M. Brunner and R. Stadler. Virtual active networks - safe and flexible environments for customer-managed services. In R. Stadler and B. Stiller, editors, *Active Technologies for Network and Service Management*. Springer, 1999. ISBN 3-540-66598-6.
7. Ken Calvert, Samrat Bhatacharjes, Ellen Zegua, and J.P.G. Sterbenz. Directions in active networks. *IEEE Communications*, 36(10):72–78, October 1998.
8. The Caml language, INRIA, France, http://caml.inria.fr, June 2002.
9. D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: A software architecture for next generation routers. In *Proceedings of the SIGCOMM Conference*, 1998.
10. M. Hicks, P. Kakkar, T. Moore, C.A. Gunter, and Scott Nettles. Network programming using plan. In B. Belkhouche and L. Cardelli, editors, *Proceedings of the ICCL Workshop, Chicago*, LNCS. Springer, 1998. ISBN 3-540-66673-7.
11. A.W. Jackson, J.P.G. Sterbenz, and R.R. Condell, M.N. Hain. Active monitoring and control: The sencomm architecture and implementation. In *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE)*, pages 379–393. DARPA, 2002.
12. G. Lefkowitz. A subjective analysis of two high level, object oriented languages. http://-www.python.org/doc/Comparisons.html, April 2000.
13. J.T. Moore, M. Hicks, and S. Nettles. Chunks in plan: Language support for programs as packets. In *Proceedings of 37th Annual Allerton Conference on Communication, Control, and Computing*, 1999.
14. Sandy Murphy. Security architecture for active nets. http://www.dcs.uky.edu/∼calvert/arch-docs.html, May 2001. AN Security Working Group.
15. Proposed ieee standard for application programming interfaces for networks. http://www-.ieee-pin.org.
16. C. Perkins. IP encapsulation within IP. Internet Standard RFC 2003, October 1996.
17. Python language website. http://www.python.org, August 2002.
18. B. Schwartz, A.W. Jackson, W.T. Strayer, W. Zhou, R.D. Rockwell, and C. Partbridge. Smart packets: applying active networks to network management. *ACM Transactions on Computer Systems*, 18(1):67–88, 2000.
19. D. Tennenhouse et al. A survey of active network research. *IEEE Communications Magazine*, January 1997.
20. G. van Rossum. Comparing python to other languages. http://www.python.org/doc/essays/-comparisons.html, August 2002.
21. W. Wang and J. Biswas. Standardizing programming interfaces for tomorrow's telecommunications network. *IEEE Standard Bearer*, 12(2), April 1998.
22. D. Wetherall, J. Guttag, and D. Tenenhouse. Ants: A toolkit for building and dynamically deploying network protocol. In *IEEE Openarch*, April 1998.