Multiple Language Family Support for Programmable Network Systems

Michael Conrad, Marcus Schöller, Thomas Fuhrmann,
Gerhard Bocksch and Martina Zitterbart

Institut für Telematik
Universität Karlsruhe, Germany

Abstract. Various programmable networks have been designed and implemented during the last couple of years. Many of them are focused on a single programming language only. This limitation might—to a certain extend—hinder the productivity of service modules being programmed for such networks. Therefore, the concurrent support of service modules written in multiple programming languages was investigated within the FlexiNet project. Basically, support for three major programming paradigms was incorporated into FlexiNet: compiled programming languages like C, interpreted languages (e.g., Java), and hardware description languages such as VHDL. The key concept can be seen in an integral interface that is used by all three programming languages. This leads to a configuration scheme which is totally transparent to the programming languages used to develop the service. In order to get a better idea about the impact of the programming language used, some measurement experiments were conducted.

1 Introduction

Active and programmable networks are a vital area of research that aims to improve existing networks, for example, with respect to the number and flexibility of services provided to customers or to network operators themselves. In order to achieve this, programmable networks introduce programmable nodes—a sort of middleboxes—that are incorporated into the network. These nodes provide services that are not available in the network itself. For example, routers at the edge of the network do not only forward packets but also provide some additional services, such as multicast support. These services can be provided on demand, i.e., they need not to be pre-installed when the router is put into operation in the network. As a result the number and flexibility of services being provided to customers can be largly increased without demanding the overall network infrastructure to be updated. Multicast serves as a good example, since the introduction of IP multicast into the network would require all routers to talk IP multicast and to provide suited routing protocols.

Although research in active and programmable networks is on-going since several years, few working frameworks exist only. The FlexiNet framework based on AMnodes discussed in this paper is considered as one of those frameworks. On the other hand, an increased interest in these technologies can be observed recently, for example, from

mobile network operators. With programmable nodes, mobility support can be easily integrated into a network. Performance enhancing proxies (e.g. [10]) can be named as an example. These proxies boost TCP performance in wireless scenarios. Furthermore, a high flexibility with respect to the network services needed is considered essential for next generation networks in general.

Up to now, most research projects concentrated on how to realise the packet mangling and manipulation, how to do resource management and overcome according limitations, or how to deploy such a system. From the customers point of view, services form the most critical part of networks. However, in the context of active and programmable networks a lack of readily available implemented services that can provide the targeted flexibility can be observed. On one hand this is due to the concentration of technical issues related to the design and implementation of such programmable nodes. On the other hand, many projects are restricted to individual programming languages and, mostly, just to a single programming language. Some projects have even developed their own service programming languages to achieve restrictions in their programming model or to ease tasks like resource control and management. This further limits the number of programmers that are motivated to implement new services. As a result, currently available services for programmable networks were mostly developed within related projects and, thus, are limited to few available services only. However, in order to attract customers a variety of widely interesting services needs to be readily available. To achieve this, the hurdle to implement new services should be as low as possible.

Enabling the development of services in a variety of well-known programming languages should help overcoming this problem and, thus, make programmable networks more attractive to customers and providers. Typically, programmers are used to a couple of programming languages they always select for network or service programming and can not be easily convienced to learn a new one. Their refusal further grows if that new language follows another programming paradigm. Programmers used to an object oriented programming language often refuse to use systems which only offer support for classical languages like C. Moreover, different programming languages have different fortes and bring along different sets of libraries. Some services may be easier implemented in Java whereas for other services C may be favourable. Therefore, the AMnet framework [4], currently part of the larger FlexiNet project, was extended to provide support for various programming languages.

In this framework service developers are no longer restricted to a single programming language. Support for different programming languages was first presented in [7] with special focus on resource control and limitations. This mechanism forms the basis for flexible usage of the programming language as long as the execution environment is run as a user space process. Any additional mechanism like a Java sandbox are usable but not necessarily needed to safely run the programmable node.

The reminder of the paper is structured as follows. Section 2 provides a brief overview of the FlexiNet framework for those readers that are not yet familiar with it. The concept of integrating different programming languages is discussed in detail in Section 3. As a usage example, an RTSP/RTP implementation with modules implemented in different languages is presented in section 4. Some measurement results on

the integration of C and Java service modules complement the paper. A brief summary and outlook on future work concludes the paper.

1.1 Related Work

The basic concepts of programmable networks have been described in various publication, see, e.g., [2,3]. AMnet [5] which now provides the FlexiNet framework is an operational implementation of many of these fundamental ideas together with specific extensions and improvements [4]. Its concept is basically that of a Linux based software router. Similar approaches have been pursued by various other projects, e.g., the Click Modular Router project [8]. There, low-level extensions to the regular network protocol stack provide a router environment in which so-called *elements* perform the basic processing steps like packet classification and mangling. Compared to AMnet, which (mostly) runs in the user-space of an unmodified Linux installation, Click's direct hardware access trades security and programming ease against performance. Both projects' common objective, namely to benefit from existing operating system functionality, is also shared by SILK [1], in which a port of Scout [9] replaces the standard Linux protocol stack.

2 Brief Introduction into FlexiNet Framework

The FlexiNet framework provides a design and implementation for the paradigm of programmable networks. The design paradigm is based on the fact that programs (respecticely service modules) can be downloaded on demand in order to provide a service. This flexibility, however, is not expected to be provided on a per-packet bases but, in contrast, for dedicated end-to-end data flows. These service modules are available from so-called service module repositories. Installation and termination of services in the network takes place according to the demands of a service user. Thereby, a service can be constructed out of several so-called service modules. These service modules are arranged in a service module chain, i.e. different service modules are processed sequentially for a certain IP packet in order to provide it with the desired service.

The architecture of the programmable node (so-called AMnode) used in the FlexiNet framework basically consists of two major components: the framework and the execution environment (cf., Fig. 1). The framework itself provides all required system services, such as handling several execution environments as well as creation and setup of desired services. The execution environment acts as runtime system for a specified service, which consists of different service modules. For each service FlexiNet creates a separate execution environment and instantiates all modules that are needed to provide a certain service in this execution environment. Inside the framework several execution environments run separately and independent from each other. IP packets for a desired service will be forwarded to the corresponding execution environment and, then, can be processed accordingly by the chain of service modules.

Currently, FlexiNet provides support for the following programming languages: C, C++, Java and VHDL. They can be used concurrently, i.e., if a service is composed of
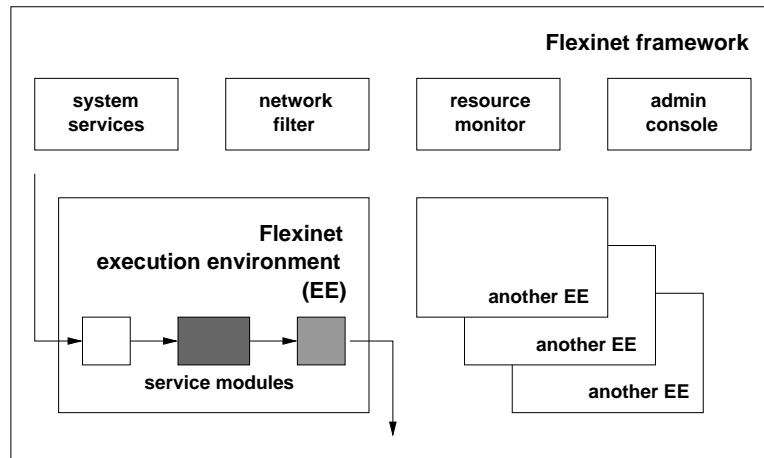
Fig. 1. Structure of FlexiNet framework and execution environment

different service modules, those modules can be written in different programming languages. C and C++ were chosen since they are popular languages for network programming. Java brings additional support that is particular valuable for higher level services. The support of VHDL allows us to outsource processing intensive service modules on dedicated programmable hardware platforms.

In FlexiNet service modules are natively programmed in C and compiled to shared libraries which are dynamically loaded into the execution environment on demand. In [6] we introduced the concept of Happlets and gave an extensive overview on performance and interaction with other service modules. Happlets are an abstraction for various hardware programming languages, such as VHDL. With Happlets a so-called virtual service module is loaded into the execution environment which sets up and configures hardware like DSPs or FPGAs and redirects incoming packets to that hardware. The concept of Happlets itself will not be discussed further within this paper. Interested readers are referred to [6].

Furthermore, Java is integrated into FlexiNet as an example of an interpreted programming language. An advantage of integrating support for Java is its platform independence in contrast to object code which is compiled for a predefined system. This eases the deployment of modules in heterogeneous environments. Moreover, Java brings along a vast number of classes and packages which alleviate the design and programming of new services. Besides C Java is a popular network programming language. Many software developers are familiar with its principals and can implement modules very fast. Both facts are expected to help increase the number of FlexiNet service programmers considerably.

The decision to integrate the different programming languages as special FlexiNet modules gave us the freedom to develop them independently as long as the interface stays the same. Further on an execution environment only has to instantiate the lan-

guage support for those languages which were used to program that service. In that way system resources can be saved but still any combination of service modules from various programming language is possible.

3 Integration of Multiple Language Support

This section focuses on the technical issues of integrating support for multiple languages in FlexiNet. It is our assumption that a service designer has broken up the problem in several subproblems. Each of these subproblems is realised in one service module. The module programmer can now freely choose from the programming languages supported by FlexiNet. The key concept can be seen in an integral interface that is used by all three programming languages. This leads to a configuration scheme which is totally transparent to the programming languages used to develop the service. An examples on how a service can be configured with service modules that are implemented in different programming languages is presented in section 3.

3.1 General integration concept

In the original design FlexiNet only supported shared object code. As a result, all service modules had to be implemented in C oder C++. For usage by the execution environment the modules must implement the following function prototypes (cf., Fig. 2).

```
1  #include "modules.h"
2
3  int module_init( struct module *module);
4  int module_shutdown( struct module *module);
5
6  int module_run( struct module *module, nl_packet_msg *nlPacketMsg,
7                  struct iphdr *ip);
% 8
% 9  int getstatus( struct module *module);
```

Fig. 2. FlexiNet C module interface

The function `module_init` is called once after instantiation of the specified module. Accordingly the function `module_shutdown` is called once at termination of the module. The function `module_run` is called for all IP packets that are processed by the corresponding service modules. The parameters of the function provide some information about the module itself as well as the desired IP packet. Inside this function the module can create, delete or modify IP packets. For example, a service module may receive a single IP packet and provide multiple replicated IP packets at its output interface (e.g., in case of a multicast service). By setting special return codes the further processing of the IP packet can be guided.

3.2 Integration of Java-based service modules

The interface for Java modules is analogous to the C interface:

```
 1  package de.flexinet.modules;
 2
 3  public interface FlexinetModule {
 4
 5    public void moduleInit( Adapter a);
 6    public void moduleShutdown();
 7
 8    public void moduleRun( IPPacket ip, NlPacketMsg nlPacketMsg)
 9      throws FlexinetException;
10
11    public String status();
12 }
```

Fig. 3. FlexiNet Java module interface

To integrate support for Java-based service modules a programming interface for Java and a wrapper module was designed and implemented. The wrapper module is written in C and implements the classical FlexiNet module interface already described above. During module initialisation the Java Virtual Machine (JVM) is started and configured using the Java Native Interface (JNI). In addition some special memory is allocated which is used to pass IP packets from the wrapper to Java. Furthermore, within the JVM an adapter class is started which functions as a counterpart to the wrapper module is started. As a result, service modules written in Java only must implement the interface `FlexinetModule` (cf., Fig. 3).

To process an IP packet the wrapper copies the IP packet into the allocated memory and calls the function `moduleRun` of the adapter which only calls the corresponding function within the Java module. After packet processing within the JVM the wrapper passes the packet to the next service module or back into the IP layer if that was the last module of the service module chain.

3.3 Startup Script for FlexiNet modules

The FlexiNet configuration scheme is transparent to the programming language used to implement service modules. All startup scripts contain two parts: the first one configures the modules used to realise the desired service, the second one defines filter rules to select packets to which the service is applied to.

The module configuration section contains all information, required for correct instantiation and setup of a specified module. The following example (cf. Fig. 4) shows the module configuration section of a C and a Java module.

```
1  loadmodule nop_c {              1  loadmodule nop_java {
2    file "libnop.so";            2    file "libjmod.so";
3                                  3    className = "Nop";
4    myparameter = "some string";  4    myparameter = "some string";
5  };                             5  };
```

Fig. 4. Module configuration section of FlexiNet execution environment script (C and Java)

The left side of figure 4) presents the configuration section for the NOP module written in C. The right side shows the configuration section for the equivalent module written in Java. Both start with the `loadmodule` command in line 1. In line 2 the C example loads the shared library `libnop.so`, which contains the NOP module written in C. In the same line the Java example loads the library `libjmod.so`, which contains the wrapper for Java modules. This wrapper creates a virtual machine and loads the desired service module. Information about that module is taken from the parameter `className` which is shown in line 3. In this case the class `Nop` should be loaded. Both module configuration sections contain the setting of a paramenter called `myparameter` in line 4.

In addition to the module configuration section each startup script must contain a network configuration section. This network configuration section is independent of the module implementation language. The section holds information about the network traffic that should be processed by the module. With the following configuration it is indicated, that the packets for all TCP connections from server `www.some-server.net` port 80 should be processed by the service module. The command `nfhook` in line 3 specifies the Netfilter hook of the Linux kernel, that applies to the service module. In the example the module selects hook `PREROUTING`. As a result, the processing of the IP traffic by this service module takes place before the routing lookup takes place.

```
1  listen {
2    protocol "tcp";
3    nfhook PREROUTING;
4
5    source {
6      name "www.some-server.net";
7      port 80;
8    };
9  };
```

Fig. 5. Network configuration section of FlexiNet execution environment script

3.4 Structuring the service modules

Only very few services can be implemented within one module. Most services are composed of several different modules each of those providing a single task. The next section shows an example service which consists of three modules.

The easiest way to connect modules is to build a single chain where packets get processed by each module consecutively. This is the standard behavior of the FlexiNet execution environment. However the service programmer can influence the order of modules. The FlexiNet execution environment supports the dynamic creation of multiple in-ports. One optional parameter of a filter entry is the module name of that module which should start the packet processing. This mechanismen was used in the following example as shown figure 6 where RTSP packets are handled diffrently from RTP packets. A second way to alter the order of modules is by building conditional branches or loops. Within each module the programmer can set the next module by a simple call to the execution environment. Furthermore it is possible to abort the packet processing any time by either deleting the packet or by returning it to the IP layer. All thoses possiblities can be used in C and Java.

Finally an effective way to pass parameters from one module to another must be provided to realise real cooperation of modules. The FlexiNet execution environment provides to mechanisms for that. The first way is to set configuration parameters of one module by a simple call the the execution environment by another module. This provides a easy interface of module interaction. The second way is to use a stack which is provided by the FlexiNet execution environment. Any module can push data onto that stack where it will remain until a module pops it from there. The access to the stack from within Java is realised by JNI functions.

4 Cooperation among heterogeneous service modules

To demonstrate the cooperation of service modules programmed with different programming languages (so-called heterogeneous service modules) we chose to realise an RTSP/RTP application level multicast service. The decision was driven by the goal to provide relevant services to emerging applications. RTSP/RTP has proven to be such a service. For example, web radio applications often use RTP to deliver audio data to the end system after the RTSP protocol negotiated several communication parameters. Many times, multiple end users will request the same audio stream from RTP servers. In such cases, programmable nodes, such as the once developed within the FlexiNet project, can help to reduce the server load and the bandwidth utilisation if multiple receivers of the same audio stream are located behind the same programmable network node. Therefore, the service placed at the programmable node acts as both, an audio stream client as well as an audio server. The client role is dedicated to the sender of the stream, i.e., the programmable nodes acts as one client receiving the audio stream. Locally, the programmable node replicates the audio stream and forwards it to all clients in its subnet. Thus it owns the server role with respect to these clients.

In order to implement the above scenario, various service modules are required, as depicted in Figure 6.
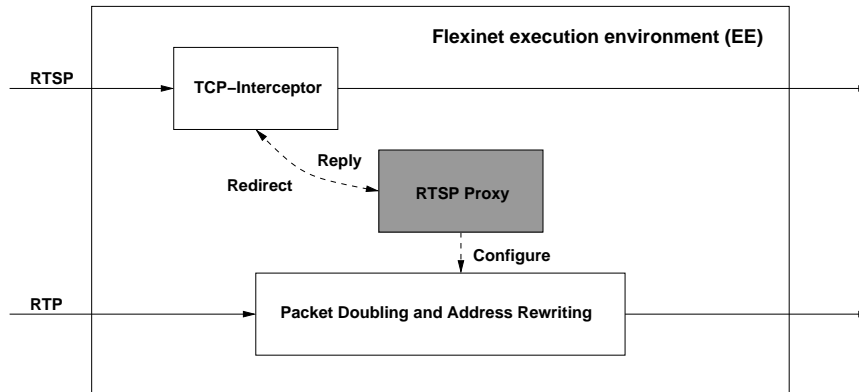
Fig. 6. A RTSP/RTP application level multicast service

The TCP interceptor module redirects incoming packets from any client to the local machine. As a result, the RTSP proxy module receives the data and parses the RTSP message accordingly. RTSP messages are text based similar to HTTP messages. If the received RTSP message is the first request for a stream, the RTSP proxy initiates a new RTSP client connection to the requested server and forwards the server information to the client. If already an earlier request for that stream was received, the RTSP proxy retransmits the corresponding cached RTSP messages to the client. Any client is added to a list of receivers for that particular audio stream. The Packet Doubling and Address Setting module reads this list of clients and replicates as required. Furthermore, the addresses are set appropriately.

The above described RTSP/RTP scenario is currently being implemented in the FlexiNet project. For the implementation it turned out to be very convenient to use different programming languages for the service modules. Since text parsing can be very easily implemented in Java—regular expression were added in version 1.4 of Java—we decided to implement that module in Java. The TCP interceptor was already implemented for other services in C. The packet duplication and address setting is a simple task that has to be applied to lots of packets. Therefore, we decided to implement this module in C. Due to the integration concept of service modules written in different languages, the described modules can interact seamlessly.

5 Some Performance Considerations

In order to get a feeling for the overhead associated with service modules written in different programming languages, we conducted a number of measurement experiments. For example, the overhead introduced by using Java was of some interest. The experiments were performed on a Pentium III 800MHz Linux (2.4.19) Router with 256MB memory. Client and server were both directly connected to this router via a 100Mbit Ethernet network.

5.1 Overhead of Java Service Modules

To measure the overhead introduced by Java several experiments were conducted. First measurements were taken on a router that was not running a FlexiNet execution environment at all. Then, first a C module and second a Java module was added to that router. Both service modules simply decremented the IP TTL field. We used the `ping` program to measure the round trip time of packets. Every test was run with 100000 packets. The results are summarized in table 1.

Module	Min/ms	Avg/ms	Max/ms
no module	0.083	0.085	0.177
C NOP module	0.101	0.106	2.259
Java NOP module	0.111	0.118	2.776
Java NOP module (cold)	0.111	0.120	14.801

Table 1. Overhead measurements

As table 1 shows, a small overhead can be noticed by the Java service module compared to the C module. Mainly, the additional copy operation of the IP packet from the wrapper to the Jave service module is responsible for that.

The last line of table 1 reflects the effect of just-in-time compilation. The Java service module is compiled at its initial execution. As a result, the maximum value is drastically higher than all other measured times for the Java service module were just-in-time compilation did not take place.

5.2 Initial Performance Tests

For the performance measurements reported here, the network tool `iperf` with packet size of 1470 byte and UDP as transport protocol was used. The test setup consisted of 3 machines. One machine acts as client, one as server and the other as a programmable node between both. The network traffic generated by the client was varied from 10Mbit/s up to the theoretical maximum of the network (100Mbit/s) and the incoming bandwidth at the server was measured.

The measurement results are summarized in figure 7. The practical achievable throughput was measured without any processing of IP packets inside the FlexiNet execution environment. Further measurements were conducted with simple service modules, that implement `UDP checksum` calculation in C and Java.

As long as the programmable node can keep up with the received load, no packet loss and, thus, no performance degradation can be observed. As soon as the node is overloaded packets must be dropped and, as a result, the bandwidth of the incoming data stream at the server degrades accordingly as shown in figure 7. Alltogether, the experiments show that the performance of the Java service modules is lower than the one of the service modules written in C. This, however, is not surprising. Furthermore, the Java modules consume more resources since they require JVM to be installed.

Relating the measurements to the above described example of an RTSP/RTP implementation with heterogeneous service modules, it can be stated that the application of Java for text parsing makes sense. This service module is part of the control plane and, thus, is not as performance critical as those modules of the data plane. Using Java to implement such a service module is perfectly suitable. On the other side, it is also advisable to write performance critical service modules, for example, in C.
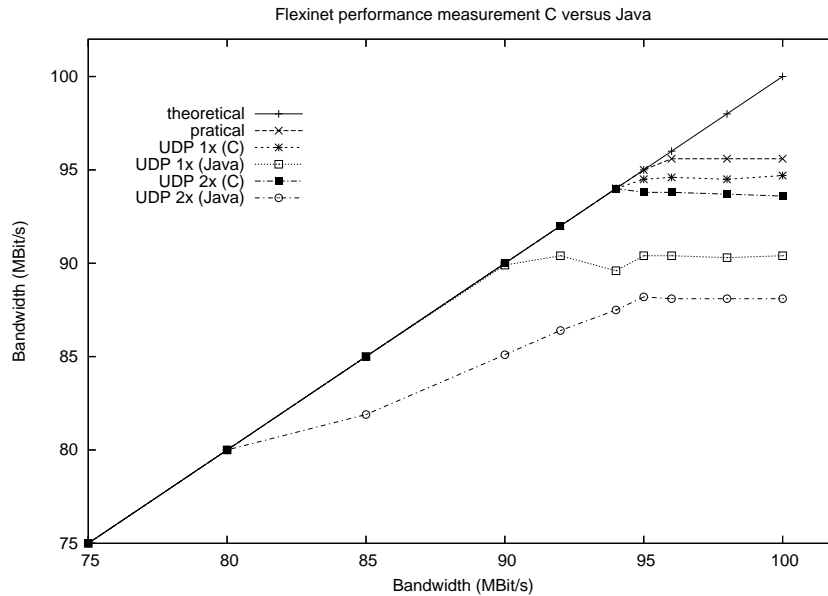

Fig. 7. Performance measurements

6 Summary

Programmable networks still lack readily available services. Among many issues that are responsible for that situation, an important aspect can be seen in the possibility of using service modules that are written in different implementation languages, dependent on the implementers preference. On the one hand service designers and programmers have varying experience in different programming paradigms or even programming languages. On the other hand programming languages have different fortes. The support of multiple programming languages eases the work of service module designing and programming and may very well raise the productivity with respect to service module implementations.

Motivated by this idea, FlexiNet was enhanced in order to simultaneously support multiple programming languages even for service modules being part of the same service module chain.

The integration of C, Java, and Happlets in FlexiNet is seamless and transparent in view to programming interfaces and configuration. In order to get a close idea on the overheads involved in the different programming languages, particularly C and Java, various measurement experiments were conducted. The higher overhead of Java could be clearly seen. However, the performance is by far good enough to apply Java service modules, for example, in the control plane as demonstrated in the RTSP/RTP example.

References

1. Andy Bavier, Thiemo Voigt, Mike Wawrzoniak, Larry Peterson, and Per Gunningberg. SILK: Scout paths in the Linux kernel. Technical Report 2002-009, Uppsala Universitet, February 2002.
2. Kenneth L. Calvert, Samrat Bhattacharjee, Ellen Zegura, and James Sterbenz. Directions in active networks. *IEEE Communications Magazine*, 36(10):72–78, October 1998.
3. Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vicente, and Daniel Villela. A survey of programmable networks. *ACM SIGCOMM Computer Communication Review*, 29(2), April 1999.
4. Thomas Fuhrmann, Till Harbaum, Panos Kassianidis, Marcus Schöller, and Martina Zitterbart. Results on the practical feasibility of programmable network services. In *2nd International Workshop on Active Network Technologies and Applications (ANTA 2003)*, 2003.
5. Thomas Fuhrmann, Till Harbaum, Marcus Schöller, and Martina Zitterbart. AMnet 3.0 source code distribution. Available from http://www.flexinet.de.
6. Till Harbaum, Anke Speer, Ralph Wittmann, and Martina Zitterbart. Providing Heterogeneous Multicast Services with AMnet. *Journal of Communications and Networks*, 3(1), March 2001.
7. A. Hess, M. Schöller, G. Schäfer, A. Wolisz, and M. Zitterbart. A dynamic and flexible access control and resource monitoring mechanism for active nodes. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH) (Short Paper Session)*, 2002.
8. Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.
9. David Mosberger. *Scout: A Path-based Operating System*. PhD thesis, Department of Computer Science, University of Arizona, July 1997.
10. M. Schlaeger, B. Rathke, S. Bodenstein, and A. Wolisz, editors. *Advocating a Remote Socket Architecture for Internet Access using Wireless LANs*, volume 6 no. 1 pp. 23-42. Mobile Networks and Applications (Special Issue on Wireless Internet and Intranet Access), January 2001.