# Programming Network Architectures

Michael E. Kounavis

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

in the Graduate School of Arts and Sciences

Columbia University

2004

ABSTRACT

# Programming Network Architectures

Michael E. Kounavis

In the thesis we address the problem of programming network architectures. We broadly define a network architecture as a distributed communication system having the following attributes: *(i)* network services, which the network architecture realizes as a set of distributed network algorithms and offers to the end systems, *(ii)* network algorithms, which include transport, signaling/control and management mechanisms, *(iii)* multiple time scales, which impact and influence the design of the network algorithms; and *(iv)* network state management, which includes the state that the network algorithms operate on (e.g., switching, routing, QOS state) to support consistent services. Programmability allows network designers to add remove, or modify network service components on-demand. By adding, removing or modifying network service components, designers can architect their networks so that they behave more optimally according to some system-wide performance objective. Architecting networks can be accomplished in many different ways such as by programming the service disciplines that are supported by the intermediate nodes of networks or by programming the routing, signaling and flow control algorithms that affect the services offered to the end-systems.

Programming network architectures is a challenging problem. The difficulty stems from the fact that it is hard to define a unifying programmable networking model and a set of programming interfaces that encompass services as diverse as routing, signaling, and access control/forwarding. Another challenging issue is related to the computational efficiency and performance of programmable network architectures. Programmable networks require more computational resources than existing networks in order to support the introduction of new services in software. In addition, today's router systems are generally configured with only a small amount of memory with limited access bandwidth. Hence, a key challenge is to design programming systems and network algorithms that can operate efficiently under stringent space-time constraints.

This thesis makes a number of contributions. First, a programmable networking model that provides a common framework for understanding the state-of-the-art in programmable networks is presented. A number of projects are reviewed and discussed against a set of programmable network characteristics. We present a simple qualitative comparison of the surveyed work and make a number of observations about the direction of the field.

Next, we present the design, implementation and evaluation of a programming system that automates a life cycle process for the creation, deployment, management, and architecting of network architectures. We discuss our experiences in building a “spawning” network testbed that is capable of creating distinct network architectures on-demand. Network architectures are created as programmable virtual networks. Our programming system is based on a methodology that allows a “child” network to operate on top of a subset of its “parent’s” network resources and in isolation from other spawned virtual networks. We show through experimentation how a number of diverse network architectures can be spawned and architecturally refined.

Third, we discuss how we can support end-system connectivity with programmable network architectures. We focus on wireless network architectures, where the connectivity problem is more challenging due to host mobility. We describe a ‘reflective handoff’ service that allows access networks to dynamically inject signaling systems into mobile devices before handoff. Thus, mobile devices can seamlessly roam between wireless access networks that support different mobility management systems. We also show how a ‘multi-handoff’ access network service can be introduced that simultaneously supports different styles of handoff control over the same wireless access network. This programmable approach can benefit service providers who need to be able to satisfy the mobility management needs of a range of mobile devices from cellular phones to PDAs and wireless laptop computers.

Next, we study the performance of network programming systems. In particular, we focus on the problem of efficiently programming the data path. Programming the data path is challenging because data path algorithms operate on the fastest time scale and are associated with small time budgets. We focus on the implementation of a network programming system in network processor-based routers. Network processors comprise multiple processing units for parallel packet processing and constitute suitable building blocks for software-based routers. We propose the design of a binding tool called NetBind that

balances the flexibility of network programmability against the need to process and forward packets at line speeds. To support dynamic binding of components with minimum addition of instructions in the critical path, NetBind modifies the machine language code of components at run time. To support fast data path composition, NetBind reduces the number of binding operations required for constructing data paths to a minimum set so that binding latencies are comparable to packet forwarding times.

Finally, we study the realization of performance critical algorithms in programmable networks. Performance critical algorithms include packet classification, packet forwarding and traffic management. While packet forwarding and traffic management problems have been well studied in the literature there are still a number of open issues associated with packet classification. We conjecture that the design of classification algorithms will need to exploit the structure and characteristics of packet classification rules. We study the properties of several classification data bases and, based on these findings, we suggest a classification architecture that can be implemented efficiently in programmable networks.

# Table of Contents

# List of Figures

# List of Tables

## ACKNOWLEDGEMENTS

I would like to thank my Ph.D thesis supervisor Prof. Andrew Campbell who influenced me most during my years at Columbia. Andrew taught me how to look for new areas of research, how to understand the state of the art quickly, how to write good technical papers and how to present my ideas effectively. Andrew gave me the necessary amount of freedom to pursue my own directions and was always there to listen even if my ideas were not all the time substantiated. I consider myself fortunate to have had the opportunity to work with Andrew.

I would also like to thank the members of my defense committee, Professor Aurel Lazar, Professor Dan Rubenstein, Professor Angelos Keromytis and John Vicente for devoting their time to read and comment on my thesis.

During the course of this dissertation I had two rewarding experiences as a summer Intern at the Network Processor Division, Intel Corporation where I had the opportunity to meet and interact with extraordinary researchers and engineers. I am especially indebted to my supervisors Dr. Raj Yavatkar and Prof. Harrick Vin for giving me the opportunity to understand new problems in the area of network processors and for sharing their time and expertise with me.

Special thanks to all the members of the COMET lab who made my graduate studies a very pleasant experience and contributed in one way or another to the outcome of this thesis. I would like to thank my colleagues Professor Giuseppe Bianchi, Stephen Chou, Gen Ito, Professor Herman De Meer, Kazuho Miki, Fabien Modoux, Daniel Villela and Hao Zhuang, for the fruitful discussions I had with them or for helping me with building the prototype systems described in this thesis.

Pursuing a Ph.D requires not only technical skill but also tremendous amount of stamina and courage. I would like to thank my parents Evangelos and Eleni and my family for sharing their unconditional love with me and giving me the necessary amount of courage required for pursuing my goals at Columbia.

*to my parents and my family*

# Chapter 1

# Introduction

## 1.1 Overview

Network architectures are complex systems that offer communication services implemented as distributed network algorithms and deployed over infrastructures of switches, routers, and base stations. We broadly define a network architecture as a distributed communication system having the following attributes: *(i)* network services, which the network architecture realizes as a set of distributed network algorithms and offers to the end systems, *(ii)* network algorithms, which include transport, signaling/control and management mechanisms, *(iii)* multiple time scales, which impact and influence the design of the network algorithms; and *(iv)* network state management, which includes the state that the network algorithms operate on (e.g., switching, routing, QOS state) to support consistent services. Two network architectures have been successfully deployed on a worldwide scale. These architectures are the Public Switched Telephone Network (PSTN) and the Internet. PSTN and the Internet have gradually grown out of smaller networks and their services and algorithms have in most cases been designed following an empirical ‘try and error’ approach. Today, the Internet connects more than a hundred and seventy million hosts (January 2003 account) while its backbone traffic doubles every year. Despite Internet's large size and growth rate, we still lack a systematic methodology for analyzing and modifying the Internet architecture. The same can be said about designing and deploying new network architectures that potentially support disruptive applications and technologies, such as, architectures for distributed games, file sharing, augmented reality, wearable computers, and so on.

The need for a methodology for designing, deploying, analyzing and modifying network architectures is apparent. First, simulation tools [112] cannot always capture the diversity of link properties, traffic patterns or protocol and service implementation constraints. Mathematical models underpinning simulation tools capture some of the analytical properties of network services (e.g., Poisson arrivals of TCP connections [111], heavy-tailed distributions of file sizes [19] or self similarity of network traffic [92, 143]) but in most cases fail to offer a complete view of the interactions that occur between the various algorithms that run in the network. This happens because network algorithms operate over multiple time scales, which impact the performance, cost and efficiency of network architectures, and because network algorithms run over complex and large topologies [52, 89]. Quantifying the behavior of network architectures, let alone understanding the interaction between the distributed algorithms characterizing architectures is not an easy task.

Second, Internet Service Providers (ISPs) cannot always deploy new network architectures at low cost. Until recently, there has been a limited need for new network architectures because the Internet infrastructure has proven sufficient to meet the demands of conventional data applications such as the web, electronic mail and ftp. However, as new applications emerge it is likely that new network architectures or even new physical networks will be needed. Third, there are no means for network managers to calculate or estimate the cost of a network architecture. By cost we mean the amount of resources that remain idle (e.g., unused bandwidth in lightly utilized networks) or used for control and management (e.g., CPU and memory for storing, accessing and modifying network state). Knowing the cost of an architecture before its final deployment is important since cost can capture an architecture's scalability, performance and commercial success. Monitoring the cost of an architecture while the architecture is operating is also is an important challenge because observations on cost can be used for real-time refinement of the architecture.

Because of the reasons discussed above, we believe that there is a need to investigate the development of experimental systems for solving problems related to the design, analysis, and performance of network architectures. Problems related to the design, analysis and performance of network architectures include the analysis and synthesis of network architecture. The analysis problem can be stated as follows. Given a network architecture, network topology and set of resources such as link and processing capacity, what is the cost and performance of a given architecture when deployed over a given topology and set of resources? The synthesis problem can be stated as follows. Given the cost and performance requirements of a network architecture, a network topology, and a set of network resources what is the minimum cost network architecture that can meet the specified cost and performance requirements when operating over the given topology and resources. To investigate solutions to these problems we need to have a good understanding of how network algorithms affect the performance and cost of network architectures. We lack such understanding today.

In this thesis, we contribute toward solving the problems of analyzing and synthesizing network architectures by investigating how to enable network designers to add, remove or modify network service components on demand. By adding, removing or modifying network service components, designers can architect their networks so that they behave more optimally according to some system-wide performance objective. Architecting networks can be accomplished in many different ways such as by programming the service disciplines that are supported by the intermediate nodes of networks or by programming the routing, signaling traffic engineering or flow control algorithms that affect the services offered to the end-systems. We propose a programmable networking model and describe the design, implementation and evaluation of the *Genesis Kernel*, a programming system that automates the creation, deployment, management and refinement of network architectures, based on our programmable networking model. The Genesis Kernel automates a virtual network life cycle process, which comprises profiling, spawning, management and architecting phases. The profiling phase captures the blueprint of a network architecture in terms of a comprehensive profiling script. The spawning phase systematically sets up the topology and address space, allocates resources and binds transport, control and management objects to the physical network infrastructure. The management phase supports virtual network resource management [28, 49] while the architecting phase allows network designers to add, remove or replace distributed network algorithms on-demand, analyzing the pros and cons of the network design space.

This thesis addresses specific challenges on how to program network architectures. These challenges are related to *(i)* the need for specifying a comprehensive yet simple set of programming interfaces that allow network designers to easily create architectures from primitives; *(ii)* the engineering of a distributed programming system that allows the introduction of new services into the network with the least possible computational overhead; *(iii)* the need to support end-system connectivity with programmable internetworking environments, and *(iv)* the design of network algorithms and services that can be efficiently implemented in software routers.

## 1.2 Technical Barriers

### 1.2.1 Defining Network Programming Interfaces

The definition of network programming interfaces concerns the specification of a set of programming primitives that allow a network designer to construct a network architecture from a set of algorithmic components and services. Ideally, we would like a network programmer to construct network architectures as easily as an end-system programmer constructs applications from libraries of classes or functions. Defining the "right" set of programming interfaces for network architectures is not an easy task. On the one hand, programming interfaces should be at sufficiently low level so as to offer the flexibility for programming a wide range of network services and algorithms. On the other hand, programming interfaces should be at a sufficiently high enough level so as to hide the complexity of network algorithm and service implementations in the same way that class interfaces hide the complexity of object implementations. In this thesis, we attempt to answer the question what is a suitable set of programming interfaces for network architectures. To answer that question, we study the design and implementation of existing network architectures. We find a set of attributes that are common to network architectures and identify the role of each attribute in the process of delivering communication services to the end-systems. Understanding the common attributes of network architectures helps us introduce a binding model that decomposes network architectures into their fundamental building bocks. We use these building blocks to define programmable objects for network architectures and use these objects to construct well known, as well as, new architectures.

### 1.2.2 Engineering Network Programming Systems

Network architecture modularity and extensibility requires the dynamic binding between independently developed components. The main challenge in engineering a network programming system is to allow the introduction of new services with the least possible amount of computational overhead. By computational overhead we mean the amount of resources (i.e., link and processing capacity resources) and state used for allowing algorithmic and service components to create associations with each other. We call the process of creating associations between algorithmic and service components *binding*. Creating a network architecture requires the realization of bindings between control plane (i.e., routing or signaling) objects and between data path (i.e., classifiers, forwarders, shapers or schedulers) objects. While code modularity and extensibility is supported by programming environments running in host processors (e.g., high level programming language compilers and linkers), such capability cannot be easily offered in the network. For example, traditional techniques for realizing code binding, (e.g., insertion of code stubs or indirection through function tables), cannot be applied to programmable routers because these techniques introduce considerable overhead in terms of additional instructions in the critical path.

To support dynamic binding between control plane objects, we study the use of conventional distributed computing platforms for network programming. We investigate the necessary modifications that need to be made using off-the-shelf distributed object technologies (e.g., CORBA, DCOM, Java RMI) so that they can be used for binding control plane objects (e.g., routing daemons or bandwidth brokers) into the network. To support dynamic binding between data path objects, we develop a binding tool called *NetBind* that modifies the machine language code of data path components at run time. In this way, NetBind can construct data paths without introducing significant overhead in terms of additional instructions in the critical path. To support fast data path composition, NetBind reduces the number of binding operations required for constructing data paths to a minimum set so that binding latencies are comparable to packet forwarding times.

### 1.2.3 Supporting End-System Connectivity

The problem of supporting end-system connectivity with programmable network architectures concerns the design of appropriate end-system and network software support so that hosts can be dynamically connected to programmable networks with diverse sets of protocols for transport control and management. While the problem has a straightforward solution for static hosts (i.e., static hosts can install the software support needed for connecting to programmable network architectures off-line), the problem is more demanding in wireless, mobile networks. This is because incompatibility of signaling systems prevents mobile devices from roaming between access networks with different mobility management architectures.

In this thesis, we propose a solution to the connectivity problem, where the implementation details of mobility management algorithms are hidden from handoff control systems, allowing the handoff detection state (e.g., the best candidate access point for a mobile device) to be managed separately from handoff execution state (e.g., mobile registration information). This software approach can be used to enable inter-system handoffs between different types of access networks. The basic idea behind realizing inter-system handoffs is that the same detection algorithms operating in mobile devices, or access networks can interface with multiple types of mobility management architectures, operating in heterogeneous access networks. Handoff control systems issue a number of generic service requests, which mobility management systems execute according to their own programmable implementation. In one case where the location of the handoff control system is at the mobile device, different mobility management protocols can be dynamically loaded into mobile devices allowing them to roam between heterogeneous access networks in a seamless manner.

### 1.2.4 Designing Efficient Network Algorithms

Network programmability requires that network algorithms are implemented in software. Software-based network algorithms usually need to maintain and to navigate through search data structures. Unfortunately, the overhead of navigating through search data structures can often exceed the time and space budget enforced by router system constraints. Thus, a key challenge is to design network algorithms that impose low memory space and time overhead. In this thesis, we focus on the design of data path algorithms because these algorithms operate on the fastest time scale and, as a result, they are associated with smaller time budgets than control and management plane algorithms. Typically, a packet data path comprises algorithms for classification, forwarding, and traffic management. While forwarding and traffic management have been investigated in the literature [14, 47, 60, 66, 119, 132, 153], we still lack a good solution for packet classification. The classification problem is challenging, particularly in IP networks because forwarding decisions are made based on the values of several different header fields (i.e., source and destination IP addresses, port number of protocol fields) and because classification rules are associated with arbitrary priority levels.

In this thesis, we conjecture that the design of efficient classification algorithms will need to exploit the structure and characteristics of packet classification rules. We study the properties of several classification data bases and, based on these findings, we suggest a classification architecture that can be implemented efficiently in programmable networks. We justify our findings based on several standard practices employed by network administrators, and thereby argue that although our findings are for specific databases, the properties are likely to hold for most databases.

## 1.3 Thesis outline

This thesis makes a number of contributions towards programming network architectures. First, a programmable networking model that provides a common framework for understanding the state-of-the-art in programmable networks is presented. Next, we present the design, implementation and evaluation of a programming system that automates a life cycle process for the creation, deployment, management, and architecting of network architectures based on a methodology of spawning network architectures. Third, we discuss how end-system connectivity can be supported in diverse programmable internetworking environments. Next, we study the performance of programmable network architectures. In particular, we focus on problem of efficiently programming the data path. Finally, we investigate the realization of performance critical algorithms such as packet classification in programmable networks. The outline of our study is as follows:

### 1.3.1 Programmable Networking Model

In Chapter 2, we present a generalized model for programmable networks. The introduction of new services is a challenging task and calls for advances in methodologies and toolkits for service creation and enabling network technologies. Before we can meet this challenge, we need to understand the fundamentals for making networks programmable. There is growing consensus that these network fundamentals are strongly associated with the deployment of new network programming environments, possibly based on “network-wide operating system support”, that explicitly recognize service creation, deployment and management in the network infrastructure.

We examine the state of the art in programmable networks and present two schools of thought on programmable networks advocated by the Active Networks (AN) [43] and Open Signalling (Opensig) [108] communities. By reviewing each contribution in turn, we arrive at a common set of features that govern the construction of programmable networks. We present a generalized model for programmable networks and a common set of characteristics that govern their construction and show how our generalized model can help with defining programming interfaces for networks.

The main results of our analysis are: *(i)* a programmable network is distinguished from any other networking environment by the fact that it can be programmed from a minimal set of APIs from which one can ideally compose an infinite spectrum of higher level services; *(ii)* we view the generalized model for programmable networks as comprising conventional communication, encompassing the transport, control and management planes, and computation. Collectively computation and communication models make up a programmable network; and *(iii)* a programmable network consists of a collection of “node kernels”, a “network programming environment” and a “programmable network architecture”.

The node kernel represents the lowest level of network programmability, providing a small set of node interfaces. These interfaces support the manipulation of the node state (e.g., accessing and controlling the node resources) and the invocation of communication services (e.g. communication abstractions and security). The network programming environment supports the construction of networks, enabling the dynamic deployment of distributed network services and protocols. The network programming environment does not offer core network algorithms (e.g., routing, signaling) that define and differentiate network architectures in the same way that an operating system does not embed application specific algorithms in the kernel. The programmable network architecture is a set of network algorithms (i.e., routing signaling, or flow control) that take into account network state and reflect the time scales over which these algorithms operate. Network algorithms are potentially as diverse as the application base that exists in the end-systems today.

### 1.3.2 A Programming System for Spawning Network Architectures

In Chapter 3, we address the problems of defining programming interfaces for network architectures and engineering a network programming system for PC-based routers. In particular, we present the design implementation and evaluation of the Genesis Kernel, a programming system for dynamically creating, deploying and managing network architectures. The Genesis Kernel is a programming system that offers programming primitives for constructing network architectures as spawned virtual networks. The design of the Genesis Kernel is based on the generalized programmable networking model discussed in Chapter 2. Three distinct levels of the Genesis Kernel support the creation of network services and algorithms:

- At the lowest level, a “transport environment” delivers packets from source to destination end-systems through a set of open programmable virtual router nodes called routelets (i.e., node kernels). A virtual network is characterized by a set of routelets interconnected by a set of virtual links, where a set of routelets and virtual links collectively form a virtual network topology. Routelets process packets along a programmable data path at the internetworking layer, while control algorithms (e.g., routing and resource reservation) are made programmable using a “programming environment”.
- Each instance of the Genesis Kernel can create a distinct programming environment that enables the interaction between distributed objects that characterize a spawned network architecture (e.g., routing daemons, bandwidth brokers, etc). The programming environment comprises a “metabus”, which is a per-virtual network software bus for object interaction (akin to CORBA, DCOM and Java RMI software buses) and a “binding interface base” which supports a set of open programmable interfaces on top of the metabus, providing open access to the routelets and virtual links associated with a spawned virtual network.
- A key capability of the Genesis Kernel is its ability to support a virtual network life cycle process that supports the dynamic creation, deployment and management of network architectures. The life cycle process comprises four phases: *(i)* a profiling phase, which captures the blueprint of a network architecture in terms of a comprehensive profiling script. Profiling captures addressing, routing, signaling, security, control and management requirements in an executable profiling script that is used to automate the deployment of networks architectures; *(ii)* a spawning phase, which systematically sets up the topology and address space, allocates resources and binds transport, control and network

management objects to the physical network infrastructure; *(iii)* a management phase, which supports virtual network resource management; and *(iv)* an architecting phase, which allows network designers to analyze the pros and cons of the architectural design space and to dynamically modify a spawned architecture by changing transport, signaling, control and management mechanisms.

### 1.3.3 End-System Connectivity

In Chapter 4, we address the problem of supporting end-system connectivity focusing on programmable mobile networks. Programmable mobile networks can be spawned using programming systems such as the Genesis Kernel discussed in Chapter 3. Supporting end-system connectivity in programmable mobile networks is not an easy task because mobile devices may roam across access networks with heterogeneous mobility management architectures. While a variety of handoff algorithms have been proposed and investigated in the past [123, 138, 145] these algorithms are mostly tailored toward the needs of some specific type of mobile device or access network. The diversity in signaling systems that characterize wireless access network architectures poses a challenge in realizing inter-system handoff.

In this chapter, we propose a solution to the intersystem handoff problem where the implementation details of mobility management algorithms are hidden from handoff control systems, allowing the handoff detection state (e.g., the best candidate access point for a mobile device) to be managed separately from handoff execution state (e.g., mobile registration information). The same detection algorithms operating in mobile devices, or access networks can interface with multiple types of mobility management architectures, operating in heterogeneous access networks.

The main results of our work are: *(i)* we present the design, implementation and evaluation of a 'reflective handoff' service that allows access networks to dynamically inject signaling systems into mobile devices before handoff. Thus, mobile devices can seamlessly roam between wireless access networks that support radically different mobility management systems; and *(ii)* we show how a 'multi-handoff' access network service can simultaneously support different styles of handoff control over the same wireless access network. This programmable approach can benefit service providers who need to be able to satisfy the mobility management needs of a wide range of mobile devices from cellular phones to more sophisticated palmtop and laptop computers. To allow a range of mobile devices to connect to programmable mobile networks we further decompose the handoff control process into programmable objects, separating the transmission of beacons, from the collection of wireless channel quality measurements and from the handoff detection algorithm.

### 1.3.4 Programming the Data Path

In Chapter 5, we study the performance of network programming systems such as the Genesis Kernel discussed in Chapter 3. In particular, we focus on the problem of efficiently programming the data path. We focus our study on a network processor-based implementation of the Genesis Kernel because network processors are suitable building blocks for software-base routers, comprising multiple processing units for parallel packet processing. Data path modularity and extensibility requires the dynamic binding between independently developed packet processing components. While code modularity and extensibility is supported by programming environments running in host processors (e.g., high level programming language compilers and linkers), such capability cannot be easily offered in the network. Traditional techniques for realizing code binding, (e.g., insertion of code stubs or indirection through function tables), cannot be applied to network processors because these techniques introduce considerable overhead in terms of additional instructions in the critical path.

One solution to this problem is to optimize the code produced by a binding tool, once data path composition has taken place. Code optimization algorithms can be complex and time-consuming, however. For example, code optimization algorithms may require to process each instruction in the data path code several times resulting in O($n$) or higher complexity as a function of the number of instructions in the critical path. Such algorithms may not be suitable for fast data path composition (i.e., when a rapid change in the structure and protocols of the network architecture is required once the traffic demand or topology changes). We believe that a binding tool for network processor-based routers needs to balance the flexibility of network programmability against the need to process and forward packets at line rates. This poses significant challenges.

In this thesis we present the design, implementation and evaluation of *NetBind*, a binding tool that balances the flexibility of network programmability against the need to process and forward packets at line speeds. The main results of our work are the following: *(i)* NetBind can produce data paths that forward minimum size packets at line rates without introducing significant overhead in the critical path. NetBind modifies the machine language code of components at run time, directing the program flow from one component to another. In this manner, NetBind avoids the addition of code stubs in the critical path; *(ii)* NetBind allows data paths to be composed at a fine granularity from components supporting simple operations on packet headers and payloads. NetBind can create packet-processing pipelines through the dynamic binding of small pieces of machine language code. A binder modifies the machine language code of executable components at run-time. As a result, components can be seamlessly merged into a single code piece, and *(iii)* NetBind supports fast data path composition reducing the number of binding operations required for constructing data paths to a minimum set so that binding latencies are comparable to packet forwarding times. In NetBind, data path components export symbols, which are used during the binding process. The NetBind binding algorithm does not inspect every instruction in the data path code but only the symbols exported by data path components. In this manner, the NetBind binding algorithm can compose packet processing pipelines very fast, in the order of microseconds.

### 1.3.5 Packet Classification in Programmable Routers

Chapters 2-5 address the problems of defining network programming interfaces and engineering network programming systems. In Chapter 6, we focus on the design of efficient network algorithms for programmable network architectures. We focus on packet classification in IP networks. Packet classification is a performance critical network algorithm that typically executes under stringent space-time constraints. Packet classification is often the first packet processing step in routers. It requires routers to maintain and to navigate through search data structures. Since flows can be identified only after the classification step, to prevent performance interference across flows, network systems must ensure that classification operates at line speeds. Unfortunately, the overhead of navigating through search data structures can often exceed the time budget enforced by the line-speed processing requirement. Thus, a key challenge is to design packet classification algorithms that impose low memory space and access overhead and hence can scale to high bandwidth networks and large databases of classification rules.

Our analysis leads us to the following main results: *(i)* the fields contained in each rule in classification data bases can be partitioned into two logical entities: (1) source and destination network address pairs that characterize distinct network paths, and (2) a set of transport level fields (e.g., port numbers, protocol identifier, etc.) that characterize network applications. In most cases, the number of distinct network paths far exceeds the number of network applications; *(ii)* the network address filters (i.e., prefix pairs) create only a few partial overlaps with each other. Thus, the total number of overlaps is significantly smaller than the theoretical bound; and *(iii)* many source-destination network address pairs share the same set of transport-level fields. Hence, only a small number of transport-level fields are sufficient to characterize databases of different sizes.

Based on these findings, we provide the following guidelines for designing efficient classification algorithms. First, the multi-dimensional classification problem should be split into two sub-problems (or two stages): (1) finding a 2-dimensional match based on source and destination network addresses contained in the packet, and (2) finding a (*n-2*) dimensional match based on transport-level fields. Whereas the first stage only involves prefix matching, the second stage involves the more general range matching. Second, because of the overlap between network address filters maintained in each classification data base, each packet may match multiple filters. Identifying all the matching filters is complex. Since the total number of overlaps is significantly smaller than the theoretical upper-bound, a design that maintains all of the intersection filters and returns exactly a single match is both feasible an desirable. Since each network address filter is associated with multiple transport-level fields, identifying the highest priority rule that matches a packet requires searching through all the transport-level fields associated with the matching network address filter. Since the number of transport-level fields associated with most classification data bases is rather small, it is possible to perform the (*n-2*) dimensional search quickly and in parallel.

## 1.4 Thesis Contributions

The major contributions of this thesis can be summarized as follows:

- We propose a generalized programmable networking model and a set of network programming interfaces for constructing programmable network architectures.
- We present the design, implementation, and evaluation of programming system for dynamically creating network architectures as spawned virtual networks and investigate the performance of such a system when running on PC-based and network processor-based routers.

- We investigate end-system connectivity issues, especially in dynamic systems such as programmable mobile networks, and provide a solution to the inter-system handoff problem.
- We investigate the design of efficient network algorithms in programmable networks focusing on packet classification, and provide a solution that can meet the stringent space-time constraints associated with modern router systems.
- We have built a number of experimental testbeds for programmable mobile networks and spawning networks, as well as developed a number of software tools such as NetBind.
- We have released the open source code for these experimental systems on the Web. The source code release includes, Mobiware [101] ; the Genesis Kernel [57], and the NetBind [105] tool.

# Chapter 2

# Programmable Networking Model

## 2.1 Introduction

The ability to rapidly create, deploy and manage novel services in response to user demands is a key factor driving the networking industry and research community. Results from this field of research are likely to have a broad impact on customers, service providers and equipment vendors across a range of telecommunication sectors, including broadband, mobile and IP networking. Competition between existing and future Internet Service Providers (ISPs) could solely hinge on the speed at which one service provider can respond to new market demands over another. The introduction of new services is a challenging task and calls for major advances in methodologies and toolkits for service creation and enabling network technologies. A vast amount of service-specific computation, processing and switching must be handled and new network programming environments have to be engineered to enable future networking infrastructures to be open, extensible and programmable.

Before we can meet this challenge, we need to better understand the limitations of existing networks and the fundamentals for making networks more programmable. There is growing consensus that these network fundamentals are strongly associated with the deployment of new network programming environments, possibly based on “network-wide operating system support”, that explicitly recognize service creation, deployment and management in the network infrastructure.

This chapter examines the state of the art in programmable networks and presents a generalized programmable networking model. In Section 2.2, we present and discuss two schools of thought on programmable networks advocated by the Active Networks (AN) [43] and Open Signalling (Opensig) [108] communities. A number of programmable network toolkits have been implemented in the past. By reviewing each contribution in turn, we arrive at a common set of features that govern the construction of these programmable networks. In Section 2.3, we present a generalized model and common set of characteristics to better understand the contributions found in the literature. Following this, in Section 2.4, we discuss a number of specific projects and characterize them in terms of a simple set of characteristics. In Section 2.5, we present a simple qualitative comparison of the surveyed work and make a number of observations about the direction of the field. We believe that a number of important innovations are creating a paradigm shift in networking leading to higher levels of network programmability. This leads us to the conclusion that an important challenge facing the programmable networking community is the development of programming environments for creating network architectures.

## 2.2 Methodologies

There has been an increasing demand to add new services to networks or to customize existing network services to match new application needs. Recent examples of this include the introduction of peer-to peer overlay networks supporting distributed file sharing applications. The introduction of new services into existing networks is usually a manual, time consuming and costly process. The goal of programmable networking is to simplify the deployment of new network services, leading to networks that explicitly support the process of service creation and deployment. There is general consensus that programmable network architectures can be customized, utilizing clearly defined open programmable interfaces (i.e., network APIs) and a range of service composition methodologies and toolkits.

Two schools of thought have emerged in the past on how to make networks programmable. The first school is spearheaded by the Opensig community, which was established through a series of international workshops. The other school, established by DARPA, constituted a large number of diverse AN projects.

The Opensig community argued that by modeling communication hardware using a set of open programmable network interfaces, open access to switches and routers could be provided, thereby enabling third party software providers to enter the market for telecommunications software. The Opensig community argued that by "opening up" the switches in this manner, the development of new and distinct architectures and services (e.g., virtual networking [96, 141, 142]) could be realized. Open signaling as the name suggests takes a telecommunications approach to the problem of making the network programmable. Here, there is a clear distinction between transport, control and management that underpin programmable networks and an emphasis on service creation. Physical network devices are abstracted as distributed computing objects (e.g. virtual switches [2], switchlets [141], and virtual base stations [32]) with well-defined open programmable interfaces. These open interfaces allow service providers to manipulate the states of the network using middleware toolkits (e.g., CORBA) in order to construct and manage new network services.

The AN community advocated the dynamic deployment of new services at runtime mainly within the confines of existing IP networks. The level of dynamic runtime support for new services goes far beyond that proposed by the Opensig community, especially when one considers the dispatch, execution and forwarding of packets based on the notion of "active packets". In one extreme case of active networking, "capsules" [147] comprise executable programs, consisting of code (for example Java code) and data. In active networks, code mobility represents the main vehicle for program delivery, control and service construction. The granularity of control can range from the packet and flow levels through the installation of completely new switchware [4]. The term 'granularity of control' [27] refers to the scope of switch/router behavior that can be modified by a received packet. At one extreme, a single packet could boot a complete software environment seen by all packets arriving at the node. At the other extreme, a single packet (e.g., a capsule) can modify the behavior seen only by that packet. Active networks allow the customization of network services at packet transport granularity, rather than through a programmable control plane. Active networks offer maximum flexibility in support of service creation but with the cost of adding more complexity to the programming model. The AN approach is, however, an order of magnitude more dynamic than Opensig's quasi-static network programming interfaces.

Both communities shared the common goal to go beyond existing approaches and technologies for the construction, deployment and management of new services in telecommunication networks. Both movements included a broad spectrum of projects with diverse architectural approaches. Few AN projects considered every packet to be an active capsule and similarly few Opensig projects considered programmable network interfaces to be static.

## 2.3 Programmable Networking Model

### 2.3.1 Communications and Computation

A programmable network is distinguished from any other networking environment by the fact that it can be programmed from a minimal set of APIs from which one can ideally compose an infinite spectrum of higher level services. We present a generalized model for programmable networks as a three-dimensional model illustrated in Figure 1. This model shows the Internet reference model (viz. application, transport, network, link layers) augmented with transport , control and management planes. The division between transport, control and management allows our model to be generally applicable to telecommunications and Internet technologies. The notion of the separation between transport, control and management is evident in architectures. In the case of Internet there is a single data path but clearly one can visualize transport (e.g., video packets), control (e.g., OSPF) and management (e.g., SMNP) mechanisms. In the case of telecommunication networks there is typically support in the architecture for transport, control and management functions. This division is motivated by the different ways these networking functions utilize the underlying hardware and by the distinct time scales over which they operate. In both cases, the planes of our generalized model remain neutral supporting the design space of different networking technologies.

The programmability of network services is achieved by introducing computation inside the network, beyond the extent of the computation performed in existing routers and switches. To distinguish the notion of a “programmable network architecture” from a “network architecture”, we have extended the communication model and augmented it with a computation model, explicitly acknowledging the programmability of network architectures. We can view the generalized model for programmable networks as comprising conventional communication, encompassing the transport, control and management planes, and computation as well, as illustrated in Figure 1. Collectively, the computation and communication models make up a programmable network. The computation model provides programmable support across the transport, control and management planes, allowing a network architect to program individual layers (viz. application, transport, network and link layers) in these planes. Another view is that programmable support is delivered to the transport, control and management planes through the computation model.

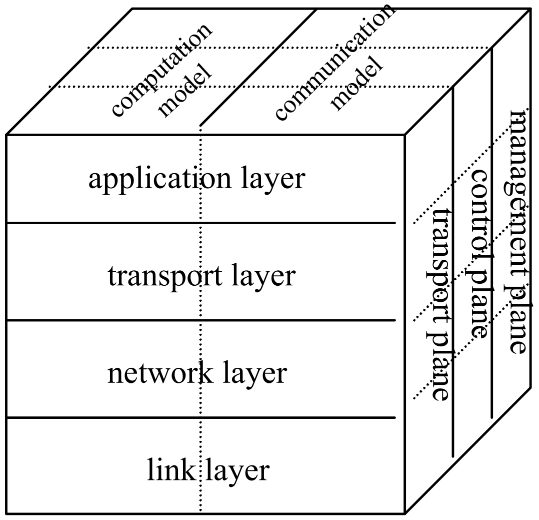


**Figure 1: Communication and Computation Models**

In Figure 2, an alternative view of the generalized model is shown. The key components of the computation model are represented as a distributed network programming environment and a set of "node kernels". Node kernels are node operating systems realizing resource management. Node kernels have local significance only, that is, they manage single node resources, potentially shared by multiple programmable network architectures. The network programming environment provides middleware support to distributed network programming services. Figure 2 illustrates the separation of switching hardware from programming and communication software. Two sets of interfaces are exposed. The first set of interfaces represents the network programming interfaces between network programming environments and programmable network architectures. The lower set of interfaces represents the node interfaces between node kernels and network programming environments. Research on programmable networks is focused on all facets of this model. Different programming methodologies, levels of programmability, and communication technologies have been investigated. Some projects, especially from the Opensig community have placed more emphasis on API definitions. Others focus on issues related to code mobility or contribute to the application domain. Dynamic "plug-ins" have been investigated for the construction or potential extension of new protocols or applications. In what follows, we provide a more detailed overview of the components of our generalized model.

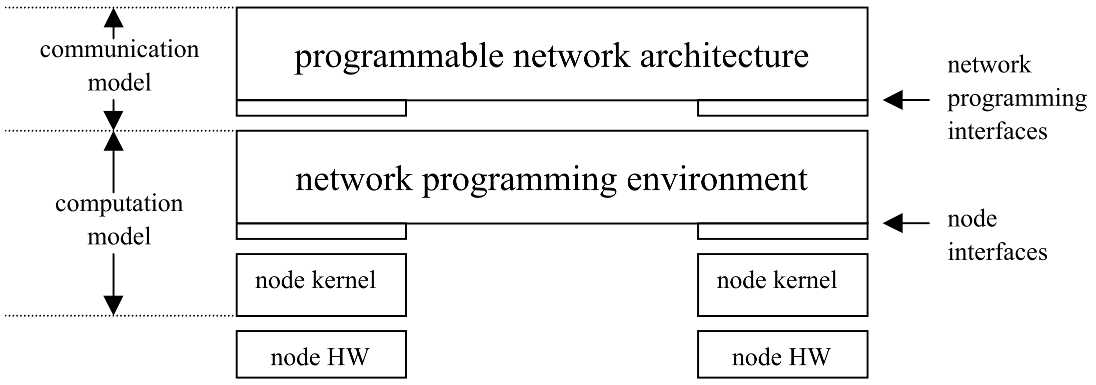

communication model
programmable network architecture
network programming interfaces
computation model
network programming environment
node interfaces
node kernel
node kernel
node HW
node HW

**Figure 2: Generalized Model for Programmable Networks**

### 2.3.2 Node Kernel

Many node vendors incorporate operating system support into their switches and routers to handle communication functions of network nodes. Typically, these node operating systems support a variety of communications activities, e.g., signaling, control and management processes, inter-process communication, forwarding functions, and downloading of new boot images. Currently, these node operating systems are closed to third party providers because of their proprietary nature, and they are limited in their support for evolving network programming environments. While the idea of introducing computation power into nodes is not new, there is a greater need for computation elements to abstract node functionality and allow it to be open and programmable. The computation model, introduced in the previous section, enables the programmability of the communication model and requires low-level programmable support for communication abstractions (e.g., packets, flows, tunnels, virtual paths), dynamic resource partitioning and security considerations.

We describe this low-level programming environment that runs on switch/routers as the node kernel. The node kernel represents the lowest level of programmability, providing a small set of node interfaces.

These interfaces support the manipulation of the node state (e.g., accessing and controlling the node resources) and the invocation of communication services (e.g. communication abstractions and security). The node kernel is responsible for sharing node computational (e.g., sharing the CPU) and communication resources, (e.g., partitioning the capacity of a multiplexer), as well supporting core security services. A node kernel may operate on any type of network node, end-system or device, for example, IP router, switch, or base station. It may also provide access to dedicated hardware offering fast packet processing services to network programming environments. A node kernel has local significance only, providing the network programming environment with a set of low-level programming interfaces that are used by network architects to program network architectures in a systematic manner.

### 2.3.3 Network Programming Environment

Network programming environments support the construction of networks, enabling the dynamic deployment of network services and protocols. Network programming environments support different levels of programmability, programming methodologies, networking technologies and application domains. Network programming environments operate over a set of well-defined node kernel interfaces offering distributed toolkits for the realization of programmable network architectures through the deployment of distributed service components. In this sense, one can view network-programming environments as the “middleware glue” between executing network architectures and the node kernels themselves, as illustrated in Figure 2. Network programming environments provide network architect/designers with the necessary environment and tools for building distinct network architectures that run in a distributed fashion on multiple node kernels. In this sense network programming environments support the programmability of network architectures in the same way that software development kits (SDKs) allow developers to build new applications that run on native operating systems.

This “middleware glue” can be constructed from scratch or be built on top of well-defined distributed object computing environments. For example, the xbind [87, 90] and mobiware [32] toolkits address programmability of broadband and mobile networks, respectively, and are built using COBRA middleware technology. Other approaches use mobile code technology and virtual machines to dynamically program the network. For example, the Active Network Transport System (ANTS) incorporates capsule technology

[147], leveraging the Java Virtual Machine for new protocol deployment. Both approaches result in toolkits that execute on node kernels offering a high level of programmability for service creation and deployment of distinct network architectures.

Network programming environments offer a set of open interfaces and services to network designers/architects to program distinct network architectures. Network programming environments support the construction of network architectures through service composition, service control, and resource and state management. Services offered by network programming environments can range from simple Remote Procedure Calling (RPC) between distributed network objects to sophisticated dynamic loading of mobile code and fast compilation of intermediate machine-independent representations. Different types of network programming environments offer different levels of programmability to network architectures. For example, mobile code technologies offer the most radical solution to the development of services in programmable networks when compared to RPC-based object middleware. We identify the 'level of programmability' as an important characteristic of programmable networks.

### 2.3.4 Programmable Network Architecture

The goal of network programming environments is to provide the necessary support to dynamically program new network architectures. Network programming environments do not offer core network algorithms (e.g., routing, signaling) that define and differentiate network architecture in the same way that operating systems do not embed application specific algorithms in the kernel. Rather, a network programming environment offers a set of network programming interfaces for constructing network architectures. Philosophically this is similar to constructing new applications using software development kits. However in this case the application is the network architecture.

We broadly define network architecture as having the following attributes:

- network services, which the network architecture realizes as a set of distributed network algorithms and offers to the end systems;
- network algorithms, which includes transport, signaling/control and management mechanisms;
- multiple time scales, which impact and influence the design of the network algorithms; and

- network state management, which includes the state that the network algorithms operate on (e.g., switching, routing, QOS state) to support consistent services.

Network programming environments offer creation and deployment tools and mechanisms that allow network architects to program and build new network architectures. Programmable network architectures are realized through the deployment of a set of network algorithms that take into account network state and reflect the time scales over which these algorithms operate. Network algorithms are potentially as diverse as the application base that exists in the end-systems today. Programmable network architectures may range from simple best-effort forwarding architectures to complex mobile protocols that respond dynamically to changes in connectivity. Given this diversity, it is necessary that both network programming environments and node kernels are extensible and programmable to support a large variety of programmable network architectures.

## 2.4 Programmable Networks

Following on from the discussion of the generalized model for programmable networks, we now survey a number of programmable networking projects that have emerged in the literature. We attempt to identify essential contributions of the various projects to the field in terms of the characteristics of the generalized programmable networking model. The survey is not intended to represent an exhaustive review of the field. Rather, we discuss a set of projects that are representative of each programmable network characteristic, focusing on the pertinent and novel features of each project and then, in Section 2.5, we compare them to the generalized model introduced in the preceding section.

### 2.4.1 Node Kernels

**Active Node Abstractions: NodeOS**

Members of the DARPA active network program [43, 134, 135] have developed an architectural framework for active networking [27]. A node operating system called NodeOS [114] represents the lowest level of the framework. NodeOS provides node kernel interfaces at routers utilized by multiple execution environments, which support communication abstractions such as threads, channels and flows. Encapsulation techniques based on an active network encapsulation protocol (ANEP) [3] support the deployment of multiple execution environments within a single active node. ANEP defines an encapsulation format allowing packets to be routed through multiple execution environments coexisting on the same physical nodes. Portability of execution environments across different types of physical nodes is accomplished by the NodeOS, by exposing a common, standard interface. This interface defines programmable node abstractions such as threads, memory, channels and flows. Threads, memory and channels abstract computation, storage, and communication capacity used by execution environments, whereas flows abstract user data-paths with security, authentication and admission control facilities. An execution environment uses the NodeOS interface to create threads and associate channels with flows. The NodeOS supports QOS using scheduling mechanisms that regulate the access to node computation and communication resources.

**Smart Packets Active Node Architecture**

The University of Kansas has developed smart packets, a code-based specialized packet concept implemented in a programmable IP environment [84]. Smart packets represent elements of in-band or out-of-band mobile code based on Java classes. Smart packets propagate state information in the form of serialized objects and carry identifiers for authentication purposes. An active node architecture supports smart packets by exposing a set of resource abstractions and primitives made accessible to smart packets. Active nodes incorporate:

- resource controllers, which provide interfaces to node resources;
- node managers, which impose static limits on resource usage; and
- state managers, which control the amount of information smart packets may leave behind at an active node.

The active node supports a feedback-scheduling algorithm to allow partitioning of CPU cycles among competing tasks and a credit-based flow-control mechanism to regulate bandwidth usage. Each smart packet is allocated a single thread of CPU and some amount of node resources. Active nodes also include router managers that support both default routing schemes and alternative routing methods carried by smart packets. The smart packets testbed has been used to program enhanced HTTP and SMTP services that show some performance benefits over conventional HTTP and SMTP by reducing excessive ACK/NAK responses in the protocols. A beacon routing scheme supports the use of multiple routing algorithms within a common physical IP network based on smart packets.

**Open Programmable Switches**

The xbind project [87, 90] investigated network programmability by opening-up the interfaces to communication nodes. The xbind broadband kernel [90] is based on a binding architecture and a collection of node interfaces referred to as Binding Interface Base (BIB) [2]. The BIB provides abstractions to the node state and network resources. Binding algorithms run on top of the BIB and bind QOS requirements to network resources via abstractions. The BIB is designed to support service creation through high-level programming languages. The interfaces are static while supporting universal programmability. The quasi-static nature of the BIB interfaces, allow for complete testing and verification of the correctness of new functions, on emulation platforms, before any service is deployed. The concept of active packets or capsules containing both programs and user data is not considered in the xbind approach to programmability. Rather, communication is performed using RPCs between distributed objects and controllers based on OMG's CORBA. The approach taken by xbind promotes interoperability between multi-vendor switch market supporting resource sharing and partitioning in a controlled manner.

## 2.4.2 Network Programming Environments

**The ANTS Toolkit**

ANTS [147], developed at MIT, enables the uncoordinated deployment of multiple communication protocols in active networks providing a set of core services including support for the transportation of mobile code, loading of code on demand and caching techniques. These core services allow network architects to introduce or extend existing network protocols. ANTS provides a network programming environment for building new capsule-based programmable network architectures. Examples of such programmed network services include enhanced multicast services, mobile IP routing and application-level filtering. The ANTS capsule-driven execution model provides a foundation for maximum network programmability in comparison to other API approaches. Capsules serve as atomic units of network programmability supporting processing and forwarding interfaces. Incorporated features include node access, capsule manipulation, control operations and soft-state storage services on IP routers. Active nodes execute capsules and forwarding routines, maintain local state and support code distribution services for automating the deployment of new services. The ANTS toolkit also supports capsule processing quanta as a metric for node resource management.

**The Switchware Toolkit**

Switchware [4] has been developed at University of Pennsylvania and attempts to balance the flexibility of a programmable network against the safety and security requirements needed in a shared infrastructure such as the Internet. The Switchware toolkit allows the network architects to trade-off flexibility, safety, security, performance and usability when programming secure network architectures. At the operating system level, an active IP-router component is responsible for providing a secure foundation that guarantees system integrity. Active extensions can be dynamically loaded into secure active routers through a set of security mechanisms that include encryption, authentication and program verification. The correct behavior of active extensions can be verified off-line by applying 'heavyweight' methods, since the deployment of such extensions is done over slow time scales.

Active extensions provide interfaces for more dynamic network programming using active packets. Active packets can roam and customize the network in a similar way as capsules do. Active packets are written in functional languages (e.g., Caml and PLAN [65]) and carry lightweight programs that invoke node-resident service routines supported by active extensions. There is much less requirement for testing and verification in the case of active packets than for active extensions, given the confidence that lower level security checks have already been applied to active extensions. Active packets cannot explicitly leave state behind at nodes and they can access state only through clearly defined interfaces furnished by active extension software. As mentioned earlier, Switchware applies heavyweight security checks on active extensions, which may represent major releases of switch code, and more lightweight security checks on active packets. This approach allows the network architect to balance security concerns against performance requirements. The security model of Switchware considers public, authenticated and verified facilities.

**The NetScript Toolkit**

The Netscript project [44] at Columbia University takes a functional language-based approach to capture network programmability using universal language abstractions. Netscript is a strongly typed language that creates universal abstractions for programming network node functions. Unlike other active network projects that take a language-based approach Netscript is being developed to support Virtual Active Networks as a programmable abstraction. Virtual Active Network abstractions can be systematically composed, provisioned and managed. In addition, Netscript automates management through language extensions that generate MIBs. Netscript leverages earlier work on decentralized management and agent technologies that automatically correlate and analyze the behavior monitored by active MIB elements. A distinguishing feature of Netscript is that it seeks to provide a universal language for active networks in a manner that is analogous to postscript. Just as postscript captures the programmability of printer engines, Netscript captures the programmability of network node functions. Netscript communication abstractions include collections of nodes and virtual links that constitute virtual active networks.

### 2.4.3 Programmable Network Architectures

**The Darwin Architecture**

The Darwin Project [40] at Carnegie Mellon University has developed an architecture for the next generation IP networks with the goal of offering Internet users a platform for value-added and customizable services. The Darwin project is focused toward customizable resource management that supports QOS. Architecturally, the Darwin framework includes Xena, a service broker that maps user requirements to a set of local resources, resource managers that communicate with Xena using the Beagle signaling protocol, and hierarchical scheduling disciplines based on service profiles. The Xena architecture takes the view that the IP forwarding and routing functions should be left in tact and only allows restricted use of active packet technology in the system.

Alongside the IP stack, Darwin introduces a control plane that builds on similar concepts such as those leveraged by broadband kernels [90] and active services [5]. The Xena architecture is made programmable and incorporates active technologies in a restricted fashion. A set of service delegates provides support for active packets. Delegates can be dynamically injected into IP routers or servers to support application specific processing (e.g., sophisticated semantic dropping) and value-added services (e.g., transcoders). A distinguishing feature of the Darwin architectural approach is that mechanisms can be customized according to user specific service needs defined by space, organization and time constraints. While these architectural mechanisms are most effective when they work in unison each mechanism can also be combined with traditional QOS architecture components. For example, the Beagle signaling system could be programmed to support RSVP signaling for resource reservation, while the Xena resource brokers and hierarchical schedulers could support traffic control.

### The Tempest Architecture

The Tempest project at the University of Cambridge [141, 142] has investigated the deployment of multiple coexisting control architectures in broadband ATM environments. Novel technological approaches include the usage of software mobile agents to customize network control and the consideration of control architectures dedicated to a single service. Tempest supports two levels of programmability and abstraction. First, switchlets, which are logical network elements that result from the partition of ATM switch resources, allow the introduction of alternative control architectures into an operational network. Second, services can be refined by dynamically loading programs that customize existing control architectures. Resources in an ATM network can be divided by using two software components: a switch control interface called ariel and a resource divider called prospero. Prospero communicates with an ariel server on an ATM switch, partitions the resources and exports a separate control interface for each switchlet created. A network builder creates, modifies and maintains control architectures.

### The Active Services Architecture

In contrast to the main body of research in active networking, Amir et al. [5] call for the preservation of all routing and forwarding semantics of the Internet architecture by restricting the computation model to the application layer. The Active Services version 1 (AS1) programmable service architecture enables clients to download and run service agents at strategic locations inside the network. Service agents called "servents" are restricted from manipulating routing tables and forwarding functions that would contravene the IP-layer integrity. The AS1 architecture contains a number of architectural components:

- a service environment, which defines a programming model and a set of interfaces available to servents;
- a service-location facility, which allows clients to ‘rendezvous’ with the AS1 environment by obtaining bootstrapping and configuration mechanisms to instantiate servents ;
- a service management system, which allocates clusters of resources to servents using admission control and load balancing of servents under high-load conditions;
- a service control system, which provides dynamic client control of servents once instantiated within an AS1 architecture;
- a service attachment facility, which provides mechanisms for clients that can not interact directly with the AS1 environment through soft-state gateways; and
- a service composition mechanism, which allows clients to contact multiple service clusters and interconnect servents running within and across clusters.

The AS1 architecture is programmable at the application layer supporting a range of application domains. In [5], the MeGa architecture is programmed using AS1 to support an active media gateway service. In this case, servents provide support for application-level rate control and transcoding techniques.

## 2.5 Discussion

We have introduced a set of characteristics and a generalized model for programmable networks to help understand and differentiate the diverse set of programmable network projects discussed in this chapter. In what follows we provide a brief comparison of these projects and other work in the field. The use of open programmable network interfaces is evident in many programmable network projects discussed in this survey. Open interfaces provide a foundation for service programming and the introduction of new network architectures. Many network programming environments shown in Table 1 take fundamentally different approaches to providing open interfaces for service composition.

| Projects | Characteristics | | | | | | | | Generalized Model for Programmable Networks | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | Node Kernels | | | | | Network Programming Environments | | | | Programmable Network Architectures | | | | |
| | | | | | programming methodology | | | | | | | | | | | | | | | | | |
| | architectural domail | networking technology | programmable communications abstractions | level of programmability | composition languages | distributed object technology | mobile code technology | encapsulation | node interfaces and binding mechanisms | resource management support | support for multiple network architectures | support for multiple programming environments | security support | service composition | service control | resource management capability | security capability | application-level services | network management | network control | information transport | routing |
| Active Services [5] | composing application level services | Internet | application services | dynamic | Tcl/oTcl | | X | | X | | | | | X | X | X | | X | | | | |
| Smart Packets, BBN [122] | network management | Internet | managed nodes | dynamic, discrete | Sprocket & Spanner | | X | X | X | | | | | X | X | | X | | X | | | |
| NetScript [44] | composing network services and VANs | Internet | VANS | dynamic, discrete | NetScript | | X | X | X | | X | | | X | X | | | | X | | | |
| ANTS [147] | composing network services | Internet | Internet protocols | dynamic, integrated | JAVA | | X | X | X | | | | | X | X | X | | X | | | | X |
| CANEs [35] | composing services | Internet | composable services | dynamic | LIANE | | X | X | | | | | | X | X | | | | | | | |
| SwitchWare [4] | composing network services | Internet | Internet protocols | dynamic | PLAN & Caml | | X | X | X | | | | X | X | X | | X | | | | | X |
| SmartPackets, U, Kansas [84] | composing network services | Internet | Internet protocols | dynamic | JAVA | | X | X | X | X | | | | X | X | X | X | X | | | | X |
| Liquid Software [64] | investigating mobile code technology | Internet | | dynamic | JAVA | | X | | | | | | | | | | | X | | | | |
| ANN [45] | composing network services | Internet | network node | dynamic, discrete | object code | | X | | X | X | | | | X | X | | X | | | | | X |
| NodeOS [114] | enabling network programmability | Internet | network node | | | | X | X | X | X | | X | | | | | | | | | | |
| xbind [2] | enabling telecommunications service creation | ATM | multimedia networks | Static | CORBA/IDL | X | | | X | X | | | | X | X | | | X | | X | | |
| DARWIN [40] | integrated resource management and value added services | Internet | flows | quasi-static | | X | X | | X | X | | | | X | X | | | X | X | X | | |
| Mobiware [32] | wireless QoS and mobile QoS control | Mobile | universal mobile channels | quasi-static | CORBA/IDL & JAVA | X | X | | X | X | | | | X | X | | | X | | X | X | |
| Tempest [142] | enabling alternative control architectures | ATM | network control architectures | quasi-static | CORBA/IDL | X | X | | X | X | X | | | X | X | | | X | | X | | |
| X-Bone [137] | automating the deployment of IP overlays | Internet | IP overlays | | | | | X | | | X | | | | | X | X | | | | | X |
| Supranet [48] | Virtual Network Services | Internet | Virtual Networks | | | | | X | | | X | | | | | X | X | | | | | X |
| Genesis [80] | Spawning Virtual Network Architectures | Internet | Spawning Networks | dynamic | Metabus/IDL | X | X | | X | X | X | X | | X | X | X | | | | | | |

**Table 1: Comparison of Programmable Networks**

The programming methodology adopted (e.g., distributed object technology based on RPC, mobile code or hybrid approaches) has a significant impact on an architecture's level of programmability; that is, the granularity, time scales and complexity incurred when introducing new APIs and algorithms into the network. Many projects use virtualization techniques to support the programmability of different types of communication abstractions. The Tempest framework [142] presents a good example of the use of virtualization of the network infrastructure. Low-level physical switch interfaces are abstracted creating sets of interfaces to switch partitions called switchlets. Switchlets allow multiple control architectures to coexist and share the same physical switch resources (e.g., capacity, switching tables, name space, etc.).

Typically, abstractions found in programmable networks are paired with safe resource partitioning strategies that enable multiple services, protocols and different programmable networking architectures to coexist. The dynamic composition and deployment of new services can be extended to include the composition of complete network architectures as virtual networks. The Netscript project [44] supports the notion of Virtual Active Networks over IP networks. Virtual network engines interconnect sets of virtual nodes and virtual links to form virtual active networks. These design principles, have been taken into account in the design and implementation of the Genesis Kernel, our network programming system for creating network architectures. The Genesis Kernel is discussed in Chapter 3.

## 2.6 Summary

In Chapter 2 we present a generalized model for programmable networks. We examine the state of the art in programmable networks and present two schools of thought on programmable networks advocated by the Active Networks (AN) [43] and Open Signalling (Opensig) [108] communities. The main results of our analysis are: (i) a programmable network is distinguished from any other networking environment by the fact that it can be programmed from a minimal set of APIs from which one can ideally compose an infinite spectrum of higher level services; (ii) we view the generalized model for programmable networks as comprising conventional communication, encompassing the transport, control and management planes, and computation. Collectively computation and communication models make up a programmable network; (iii) a programmable network consists of a collection of “node kernels”, a “network programming environment” and a “programmable network architecture”.

# Chapter 3

# A Programming System for Spawning Network Architectures

## 3.1 Introduction

In this chapter we describe the design, implementation and evaluation of the Genesis Kernel, a programming system that automates the creation, deployment, management and refinement of network architectures, based on the programmable networking model described in the previous chapter. The Genesis Kernel supports the deployment of network architectures as programmable virtual networks. We call a virtual network installed on top of a set of network resources a 'parent virtual network'. We investigate the realization of parent virtual networks with the capability of creating 'child virtual networks' operating on a subset of network resources and topology. This is a departure from the operating system analogy. The two architectures (i.e., parent and child) would be deployed in response to possibly different user needs and requirements. For example, part of an access network to a wired network might be re-deployed as a pico-cellular virtual network supporting fast handoff. Other examples include virtual networks that can be either under the control of a service provider (such as an ISP) or under customer control. Child networks operate on a subset of the topology of their parents and are restricted by the capabilities of their parent's underlying hardware and resource partitioning model. While parent and child networks share resources, they do not necessarily use the same software for controlling those resources. In this thesis we use the term ‘spawning networks’ to refer to programmable virtual networks that use the Genesis Kernel to create new network architectures as child virtual networks. The term ‘network spawning’ is first introduced in [88].

The Genesis Kernel automates a virtual network life cycle process, which comprises ‘profiling’, ‘spawning’, ‘management’ and ‘architecting’ phases. The profiling phase captures the blueprint of a network architecture in terms of a comprehensive profiling script. The spawning phase systematically sets up the topology and address space, allocates resources and binds transport, control and management objects to the physical network infrastructure. The management phase supports virtual network resource management [28, 49] while the architecting phase allows network designers to add, remove or replace distributed network algorithms on-demand analyzing the pros and cons of the network design space.

In order to evaluate our approach we have built a spawning network testbed and designed a set of experiments to help verify the Genesis Kernel’s capability to dynamically create, manage and architect network architectures. We have spawned a parent network architecture that supports IP forwarding, and interior and exterior routing. The spawning networks testbed comprises a number of heterogeneous link layers including Ethernet, wireless LAN and ATM technologies. Two distinct child networks have been spawned over the parent network based on the Cellular IP [140] and Mobiware [32] architectures offering wireless data and multimedia services to mobile users, respectively. Both of these architectures were previously developed by the COMET Group, and have been fully implemented and evaluated in standalone testbeds; see [30] and [32] for details. We refer to the spawned IP, Cellular IP and Mobiware architectures as the baseline architectures. We also show how the Mobiware and Cellular IP child networks can be architecturally refined.

This chapter is structured as follows. In Section 3.2 we describe the Genesis Framework and discuss the principles that underpin programmable network architectures. Following this, in Section 3.3 we describe our prototype implementation. In Section 3.4 we discuss how network services can be programmed using the Genesis kernel. In Section 3.5 we present our experiences with using the Genesis Kernel, focusing on the dynamic creation, deployment and management of the baseline network architectures. In Section 3.6, we present related work in the area of automating the process of creating network architectures. Finally, we present some concluding remarks and summary in Section 3.7.

## 3.2 The Genesis Kernel

### 3.2.1 The Genesis Kernel Framework

The Genesis Kernel has a layered structure that is derived from the programmable networking model described in Chapter 2. Three distinct levels of the Genesis Kernel support spawning, as illustrated in Figure 3. At the lowest level, a transport environment delivers packets from source to destination end-systems through a set of open programmable virtual router nodes called routelets. Routelets are the realization of the node kernels of our generalized model for programmable networks (see Chapter 2) in the Genesis Kernel programming system. Routelets represent the lowest level operating system support dedicated to a virtual network. A virtual network is characterized by a set of routelets interconnected by a set of virtual links, where a set of routelets and virtual links collectively form a virtual network topology. Routelets process packets along a programmable data path at the internetworking layer, while control algorithms (e.g., routing and resource reservation) are made programmable using the virtual network kernel, (i.e., the Genesis Kernel). A Genesis router is capable of supporting multiple routelets, which represent components of distinct virtual networks that share computational and communication resources.

Child routelets are instantiated by the parent network during spawning, as illustrated in Figure 3. The parent virtual network kernel acts as a resource allocator, arbitrating between requests made by spawned routelets. In addition, routelets are controlled through separate programming environments. Each virtual network kernel can create a distinct programming environment that enables the interaction between distributed objects that characterize a spawned network architecture (e.g., routing daemons, bandwidth brokers, etc.), as illustrated in Figure 2. The programming environment comprises a metabus, which is a per-virtual network software bus for object interaction (akin to CORBA, DCOM and Java RMI software buses). The metabus creates isolation between the distributed objects associated with different spawned virtual networks. A binding interface base [2] supports a set of open programmable interfaces on top of the metabus, which provide open access to a set of routelets and virtual links that constitute a virtual network. A key capability of the Genesis Kernel is its ability to support a virtual network life cycle process that supports the dynamic creation, deployment and management of network architectures. The life cycle process comprises four phases:

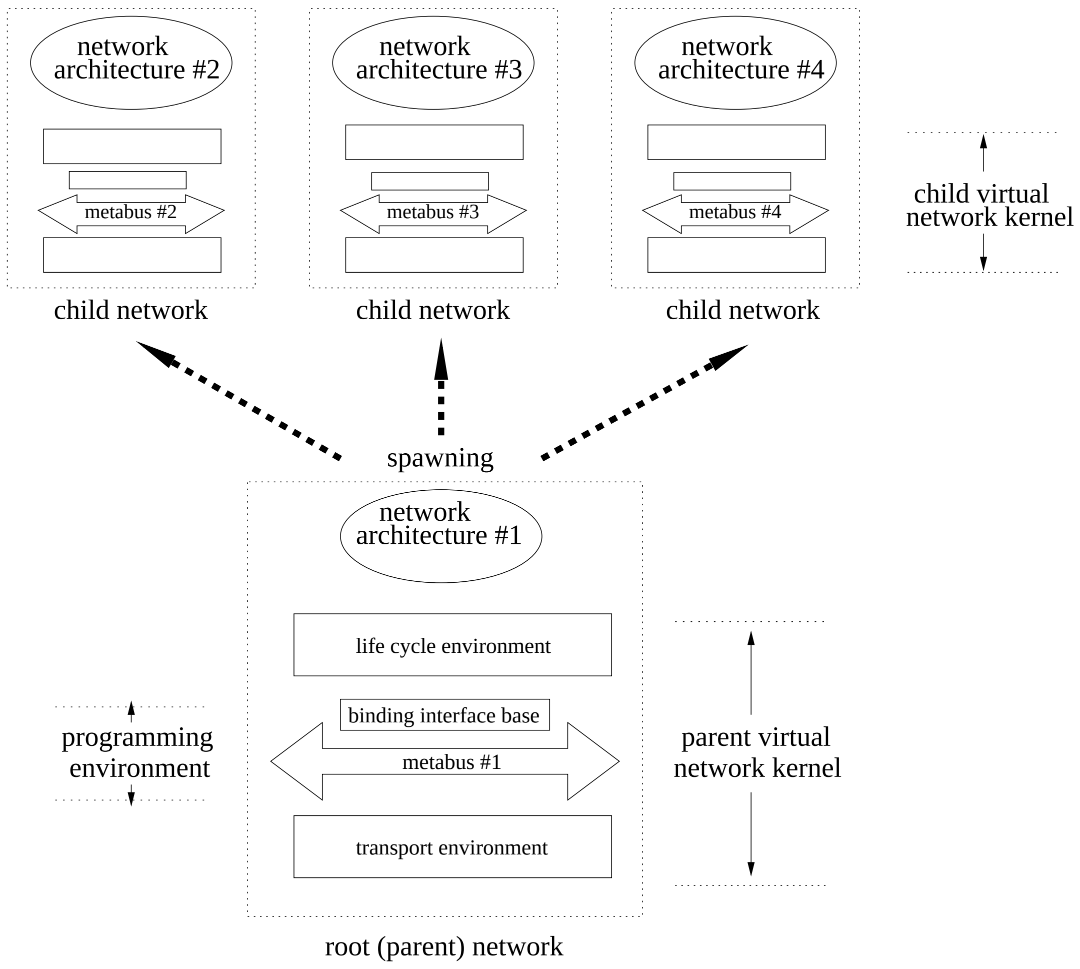


**Figure 3: The Genesis Kernel Framework**

- profiling, which captures the blueprint of the virtual network architecture in terms of a comprehensive profiling script. Profiling captures addressing, routing, signaling, security, control and management requirements in an executable profiling script that is used to automate the deployment of programmable virtual networks;
- spawning, which systematically sets up the topology and address space, allocates resources and binds transport, control and network management objects to the physical network infrastructure. Based on the profiling script and available network resources, network objects are created and dispatched to network nodes thereby dynamically creating a new virtual network architecture;
- management, which supports virtual network resource management based on per-virtual network policy to exert control over multiple spawned network architectures; and

- architecting, which allows network designers to analyze the pros and cons of the architectural design space and to dynamically modify a spawned architecture by changing transport, signaling, control and management mechanisms.

As illustrated in Figure 3, the metabus and binding interface base also support the life cycle environment, which realizes the life cycle process. When a virtual network is spawned a separate virtual network kernel is created by the parent network on behalf of the child. The transport environment of the child virtual network kernel is dynamically created through the partitioning of network resources used by the parent transport environment. In addition, a metabus is instantiated to support the binding interface base and life cycle service objects associated with a child network. The profiling and spawning of a child network is controlled by its parent virtual network kernel. In contrast, the child virtual network kernel is responsible for the management of its own network. The terms virtual network kernel, child virtual network kernel and parent virtual network kernel all refer to instantiations of the Genesis Kernel. The terms child virtual network kernel and parent virtual network kernel refer to the instantiation of the Genesis Kernel at different levels in a virtual network inheritance tree (see next section).

### 3.2.2 Design principles

The Genesis Kernel is governed by the following set of design principles.

- Separation Principle: Spawning results in the composition of a child network architecture in terms of transport, control and management algorithms. Child networks operate in isolation with their traffic being carried securely and independently from other virtual networks. The allocation of parent network resources used to support a child network is coupled with the separation of responsibilities and the transparency of operation between parent and child architectures. The reason why the separation principle is important is because different network applications require different protocols for transport control and management. As a result the profiling, spawning, management and refinement cycle for a particular architecture may be different from the life cycle of another. In order for the network architectures to operate safely over a guaranteed set of resources isolation is required.
- Nesting Principle: A child network inherits the capability to spawn other virtual networks creating the notion of 'nested virtual networks' within a virtual network. This is consistent with the idea of creating

infrastructure that supports relatively long-lived virtual networks (e.g., a corporate virtual network that operates over a long time-scale) and short-lived virtual networks (e.g., collaborative child group networks operating within the context of the corporate parent network but only active for a short period). The parent-to-child relationship represents a 'virtual network inheritance tree' [28]. The Genesis Kernel ideally supports iterative spawning resulting in hierarchy of nested virtual networks. Earlier results on the performance of network architectures indicate nesting may be a feasible technique for improving the cost and performance of a network from the service provider/network designer point of view. For example, the performance of non-cooperative networks is studied in [78]. It is shown in [78] that if among the users of a non-cooperative network one user plays the role of a manager and if its bandwidth demands exceed a certain threshold, then the manager can enforce a network optimum operating point. The same can be argued about the resources used by the manager. The resources of the manager can be iteratively divided among users, where one user could play the role of a manager (i.e., the manager of the manager of the network). Resource partitioning can take place iteratively, resulting in more optimal use of network resources according to some performance objective.

- Inheritance Principle: Child networks can inherit architectural components (e.g., resource management capabilities and provisioning characteristics) from parent networks. The Genesis Kernel, which is based on a distributed object design, uses inheritance of architectural components when composing child networks. Child networks can inherit any aspect of their parent architecture, which is represented by a set of distributed network objects for transport, control and management.

### 3.2.3 The Transport Environment

The transport environment consists of a set routelets, which represent open programmable virtual nodes. A routelet operates like an autonomous virtual router that forwards packets at layer three, from its input ports to its output ports, scheduling virtual link capacity and computational resources. Routelets support a set of transport modules that are specific to a spawned virtual network architecture, as illustrated in Figure 4. A routelet comprises a forwarding engine, a control unit and a set of input and output ports, and may optionally support higher level protocol stacks.

**Ports and Engines**

Ports and engines, shown in Figure 4, manage incoming and outgoing packets as specified by a virtual network profiling script. A profiling script captures the composition of routelet components. Ports and engines are dynamically created during the spawning phase from a set of transport modules, which represent a set of generic routelet plug-ins having well defined interfaces and globally unique identifiers. Transport modules (e.g., encapsulators, forwarders, classifiers, schedulers) can be dynamically loaded into routelets by the Genesis Kernel to form new and distinct programmable data paths.

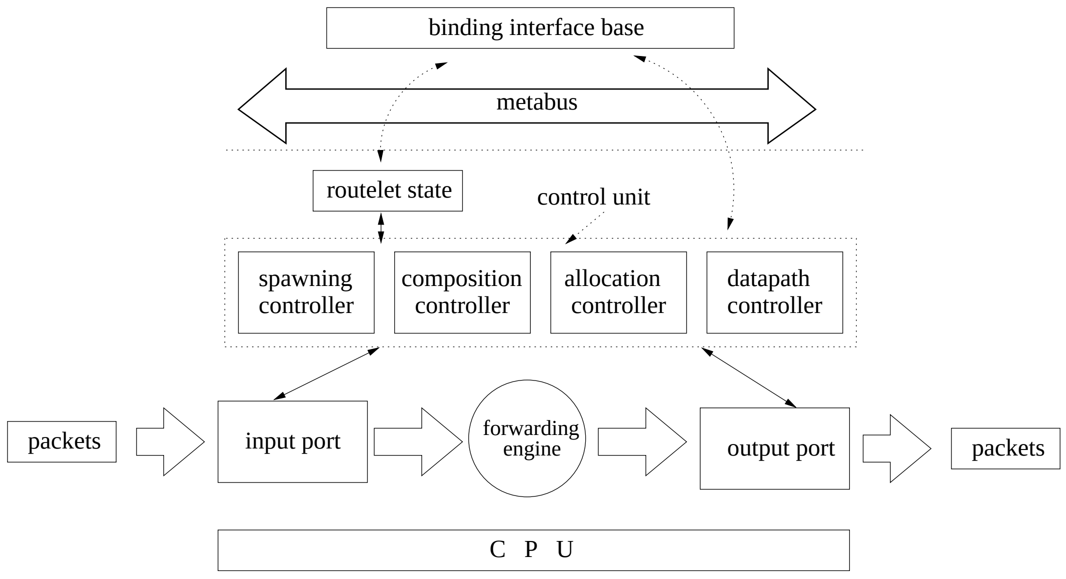


**Figure 4: Routelet Architecture**

Child ports and engines can be constructed by directly inheriting their parent's transport modules or through dynamic composition by selecting new modules on-demand. Forwarding engines bind to input and output ports constructing a data path to meet the specific needs of an embryonic network architecture. Input ports process packets as they enter the routelet based on the instantiated transport modules. In the case of a differentiated services [124] routelet for example, the input ports would contain differentiated service specific mechanisms (e.g., meters and markers used to maintain traffic conditioning agreements at boundary routelets of a differentiated service virtual network). A virtual link is typically shared by user/subscriber traffic generated by end-systems associated with the parent network and by aggregated traffic associated with child networks. User and child network traffic contend for the parent's virtual link capacity. The output port regulates access to the communication resources (which are associated with a virtual link) among these competing elements.

**Control Unit**

A routelet is managed by a control unit that comprises a set of controllers:

- a spawning controller, which "bootstraps" child routelets through virtualization;
- a composition controller, which manages the composition of a routelet using a set of transport module references and a composition script to construct ports and engines;
- an allocation controller, which manages the computation resources associated with a routelet, and
- a datapath controller, which manages the communication resources and the transportation of packets.

The spawning, composition and allocation controllers are common for all routelets associated with a virtual network. In contrast, datapath controllers are dynamically composed during the spawning phase based on a profiling script. Datapath controllers manage transport modules that represent architecture-specific data paths supporting local routelet treatment (e.g., QOS control using transport modules such as policers, regulators, buffering, queuing and scheduling mechanisms).

Routelets also maintain 'state' information that comprises a set of variables and data structures associated with their architectural specification and operation. Architectural state information includes the operational transport modules reflecting the composition of ports and forwarding engines. State information includes a set of references to physical resource partitions that maintain packet queuing, scheduling, memory and name space allocations for routelets. Routelet state also contains virtual network specific information (e.g., routing tables, traffic conditioning agreement configurations).

Routelets generalize the concept of partitioning physical switch resources introduced in [141, 142]. Routelets are designed to operate over a wide variety of link layer technologies including Ethernet, wireless LAN and ATM. The underlying link layer technology, however, may impact the level of programmability and QOS provisioning that can be delivered at the internetworking layer.

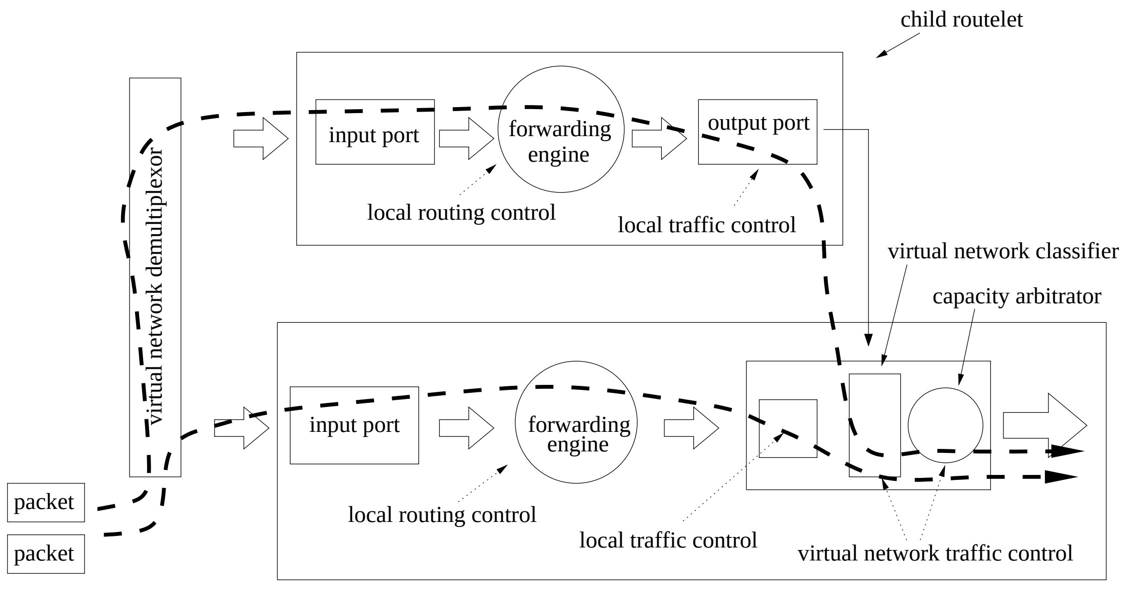


**Figure 5: Nested Routelets**

**Nested Routelets**

Nested routelets operate on the same physical node and maintain their structure according to a virtual network inheritance tree. Child routelets are dynamically created and composed during the spawning process when the parent’s computational and communication resources are allocated to support the execution of a child routelet. Each reference to a physical resource made by a child routelet is mapped into a partition controlled and managed by its parent. In addition, user traffic associated with a child routelet is handled in an aggregated manner by the parent routelet. Routelets are unaware that packets are processed according to an inheritance tree. A routelet simply receives a packet on one of the input ports associated with its virtual links, sends the packet to its forwarding engine for processing which then forwards the packet to an output port where it is finally scheduled for virtual link transmission.

We use an example scenario to illustrate how nesting is supported in the router. As illustrated in Figure 4, two packets arrive at a Genesis router. Every packet arrival must be demultiplexed to a given spawned virtual network. A virtual network demultiplexor is programmed to identify each packet's targeted virtual network (i.e., its routelet) based on a unique virtual network identifier assigned during the spawning phase. Each packet that arrives at a Genesis router must eventually reach the input port of its targeted virtual network routelet, as illustrated in Figure 5. The first packet in the example traverses the first level (child) routelet. The other packet traverses the parent network routelet directly. Mapping is always performed between the child and parent transport environments. Mapping is done through the management of transport module references by parent and child composition controllers, which are capable of realizing specified 'binding models' between the ports and engines of parent and child networks. This mapping is performed at each virtual network layer (i.e., routelet) down to the root of the inheritance tree.

A 'capacity arbitrator' [28] located at the parent's output port controls access to the parent's link capacity. Every packet is treated according to a virtual network policy, which may be different at each routelet or virtual network. In one extreme case each packet traverses the nested hierarchy tree until it is scheduled and exits onto the physical output link. In another case, a common fast path can be used by all virtual networks. The fast path is supported by the root network of the inheritance tree in this case. Child networks can inherit the fast path from their parents. The fast path supports hierarchical resource management and scheduling.

**Virtual Network Demultiplexing**

The Genesis Kernel supports explicit virtual network demultiplexing at the cost of an additional protocol field inserted in the frame format. This is accomplished by inserting a virtual network identifier between the internetworking and link layer headers. Although this appears to be a radical approach, it represents a simple way to differentiate traffic between programmable virtual networks without introducing virtual network semantics into the internetworking layer. In addition, classification of packets between virtual network flows can be done with a single memory access (e.g., a single hash lookup), without burdening the data path with significant overhead. Virtual networks are allowed to manage their own name space (e.g., addressing schemes) independent of each other, utilizing different forwarding mechanisms. The virtual network identifier is dynamically allocated and passed into the routelets of a virtual network by the life cycle environment of the parent kernel. The virtual network demultiplexor maintains a database of virtual network identifiers to map incoming packets to specific routelets (e.g., child routelets, fast path routelets).

### 3.2.4 The Programming Environment

Each network architecture comprises a set of distributed controllers that realize communication algorithms (e.g., routing, control, management), as discussed in Section 3.2.1. These distributed controllers use the programming environment for interaction. While the implementation of routelet transport modules is platform-dependent, the programming environment offers platform-independent access to these components allowing a variety of protocols to be dynamically programmed. The programming environment is illustrated in Figure 6 and discussed below.

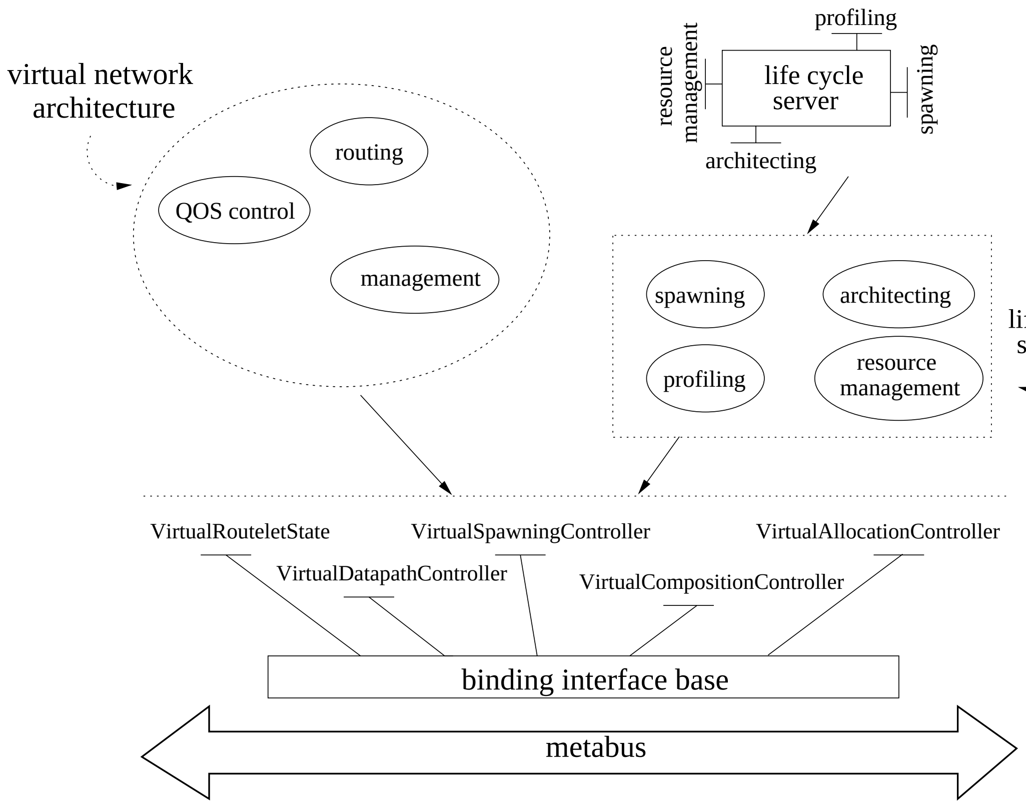


**Figure 6: Programming Environment**

**Metabus**

A metabus supports a hierarchy of distributed objects that realize a number of virtual network specific communication algorithms including routing, signaling, QOS control and management. At the lowest level of this hierarchy binding interface base objects provide a set of handlers to a routelet's controllers and resources allowing for the programmability of a range of internetworking architectures using the programming environment. The binding interface base separates the implementation of the finite state machine, which characterizes communication algorithms (e.g., the RIP finite state machine, the RSVP finite state machine, etc.), from the implementation of the mechanisms that transmit signaling messages inside the network. Communication algorithms can be implemented as interactions of distributed objects, independent of the network transport mechanisms. Distributed objects that comprise network architectures (e.g., routing daemons, bandwidth brokers, etc.) are not aware of the existence of routelets. Distributed objects give the 'illusion' of calling methods on local objects whereas in practice call arguments are 'packaged' and transmitted over the network via one or more routelets. This abstraction is provided by the metabus.

We have chosen to realize the metabus abstraction as an orblet, a virtual Object Request Broker (ORB) derived from the CORBA [108, 144] object-programming environment. Typically, CORBA is used in enterprise networking solutions and runs on client and server nodes to support distributed applications. We have developed network kernels [32] that use CORBA technology for service creation, signaling and management in previous projects, that are not directly related to this thesis. The use of off-the-shelf CORBA allows us to quickly develop simple programmable network architectures and spawn them using the Genesis Kernel. See Section 3.3 for details on the orblet implementation.

The use of CORBA in the network presents a number of scalability issues that the metabus resolves. Distributed objects that comprise distinct spawned network architectures need to be isolated for scalability reasons. Existing ORB technology supports a number of ad-hoc solutions for realizing isolation between distributed object computing environments. The metabus extends the capabilities offered by CORBA by supporting the dynamic creation of multiple isolated software buses for spawned virtual network architectures.

**Binding Interface Base**

The interfaces that constitute the binding interface base are illustrated in Figure 5. A VirtualRouteletState interface allows access to the internal state of a routelet (e.g., architectural specification, routing tables). The VirtualSpawningController, VirtualCompositionController and

VirtualAllocationController interfaces are abstractions of a routelet's spawning, composition and allocation controllers, respectively. The VirtualDatapathController is a 'container' interface to a set of objects that control a routelet's transport modules. When the transport environment  (e.g., output port) is modified the binding interface base is dynamically updated to include new module interfaces in the VirtualDatapathController.

Every routelet is controlled through a number of implementation-dependent system calls. Binding interface base objects wrap these system calls with open programmable interfaces that facilitate the interoperability between routelets that are possibly implemented with different technologies. Routing services can be programmed on top of a VirtualRouteletState interface that allows access to the routing tables of a virtual network. Similarly, resource reservation protocols can be deployed on top of a VirtualDatapathController interface that controls the classifiers and packet schedulers of a routelet's programmable data path.

### 3.2.5 The Life Cycle Environment

The life cycle environment provides support for the profiling, spawning, management and architecting of virtual networks. Profiling, spawning, management and architecting provide a set of services and mechanisms, which are common to all virtual networks that inherit from the same parent. Life cycle services can be considered as kernel 'plugins' because they can be replaced or modified on-demand. Life cycle services can be programmed using the metabus and binding interface base of the Genesis Kernel. The life cycle is realized through the interaction of the transport, programming and life cycle environments. In what follows, we provide an overview of the life cycle services.

**Profiling**

Before a virtual network can be spawned the network architecture must be specified and profiled in terms of a set of software and hardware building blocks annotating their interaction. These software building blocks include the complete definition of the communication services and protocols that characterize a network architecture as outlined in Section 3.2. The process of profiling captures addressing, routing, signaling, control and management requirements in an executable profiling script that is used to automate the deployment of programmable network architectures. During this phase, a virtual network architecture is specified in terms of a topology graph (e.g., routers, base stations, hosts and links), resource requirements (e.g., link capacities and computational requirements), user membership (e.g., privileges, confidentiality and connectivity graphs) and security specifications. Programmability enables the architect to include addressing, routing, signaling, control and management mechanisms in the profiling script. The output from this phase of the life cycle is a comprehensive profiling script.

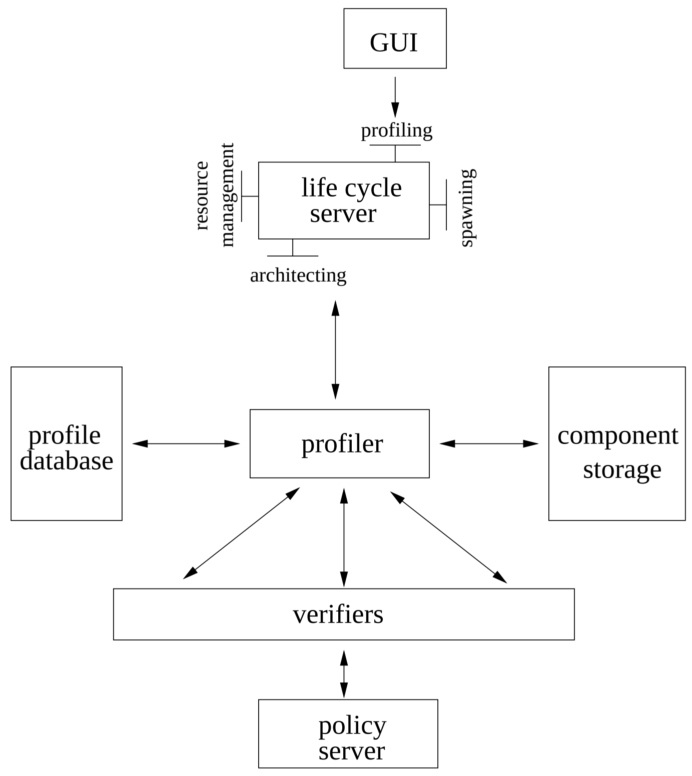


**Figure 7: Profiling**

The first step in profiling a target virtual network is the selection of nodes and links from a parent provider network and the composition of a customized topology graph. The details of the parent network are stored, managed and presented in a profile database maintained by the parent virtual network kernel. The topology may span wireline and wireless sub-networks and cover a wide area interconnecting a number of Intranets or it may be restricted to a few local sub-networks. When profiling large-scale networks the architect has the flexibility to provide an outline of the topology graph, specifying strategic sub-networks or backbone routers that should be included in the topology. In this case a profiling tool completes the specification. Once the topology is specified it is augmented with a user connectivity model and membership assignments.

The Genesis profiling system allows the network designer to dynamically select architectural components and instantiate them as part of a spawned network architecture. For example, a number of routing protocols for intra-domain and inter-domain routing can be made available using a component storage of the parent network (e.g., RIP, OSPF, BGP). Similarly, QOS or resource provisioning architectures based on well-founded models can be dynamically selected and used for controlling resources in virtual networks. Transport protocols (e.g., TCP, RTP, UDP) and network management components (e.g., SNMP) made available as software building blocks can be instantiated on-demand.

A number of important characteristics of a virtual network are realized as component parts to routelets, including forwarding algorithms, addressing schemes, QOS provisioning capability, encryption, tunneling and multicast support. These features can be explicitly specified in the virtual network profile, selected from a database of existing architectural components or inherited from a parent. Routelets can be explicitly specified in the virtual network profile or network designers can select routelets from a library to realize a certain service. An important part of profiling is the description of the routelets and virtual links that connect and form a network-wide topology. Ports and forwarding engines are composed to meet the specific architectural requirements of the virtual network architecture in terms of the characteristics of a programmable low-level data path.

The Genesis profiling system is illustrated in Figure 7. Network architects utilize a graphical utility profiling tool to generate the virtual network profile. The profiling system interacts with a parent virtual network kernel through a life cycle server as illustrated. A profiler service queries information about the parent network from a profile database. The profile database provides information about the topology and architecture of the parent network. The first level of information exposes a coarse presentation of the network topology while the second level presents details on intermediate nodes and links. The database provides quantitative and qualitative characteristics of virtual networks.

**Spawning**

Once the network architecture has been fully specified in terms of a profiling script it can be dynamically created. The process of spawning a network architecture relies on the dynamic composition of the communication services and protocols that characterize it and the injection of these algorithms into the nodes of the physical network infrastructure constituting a virtual network topology. The spawning process systematically sets up the topology and address space, allocates resources, and binds transport, routing and network management objects to the physical network infrastructure. Throughout this process a virtual network admission test is in operation.

Spawning child network architectures includes creating child transport and programming environments and instantiating the control and management objects that characterize network architectures. The creation process associated with spawning a child transport environment centers around the creation and composition of routelets, the bootstrapping of routelets into physical routers based on the child network topology, and finally, the binding of virtual links to routelets culminating in the instantiation of a child transport environment over a parent network. The Genesis Kernel allows a child network to inherit the life cycle support from its parent.

**Management**

Once a profiled architecture has been successfully spawned the virtual network needs to be controlled and managed. The management phase supports virtual network resource management based on per-virtual network policy that is used to exert control over multiple spawned network architectures. The resource management system can dynamically influence the behavior of a set of virtual network resource controllers through a slow timescale allocation and re-negotiation process. Virtual network resource management is a subject of on-going study, whereas several virtual network resource management systems have been studied by the community [28, 49].

**Architecting**

By observing the dynamic behavior of virtual networks, spawned network architectures can be refined. Through the process of architecting a network designer uses visualization and management tools to analyze the pros and cons of the virtual network design space and through refinement modify network objects that characterize the spawned network architecture. For example, the Cellular IP architecture could be refined to optimally operate in pico-, campus- and metropolitan-area environments through the process of architecting and refinement.

Architecting appears to be an exceedingly difficult task. One of the goals of our work is to build more powerful tools to help with the architecting process allowing for a more systematic study of the design space under operational conditions. The development of visualization tools is an important part of this work. However, the effective use of architecting depends on more than a visualization tool. Rather, it depends on a well founded understanding of what should be achieved versus what may be achieved and how to modify a prototype network architecture accordingly.

## 3.3 Implementing Spawning Networks

We have been working on the design and implementation of the kernel since the Spring of 1998 and have completed the implementation of Genesis Kernel over PC-based and network processor-based routers [105]. The transport and programming environments have been implemented using commodity operating systems, binding tools for network processors (see Chapter 5), and distributed systems technology. There remain a number of technical barriers to realizing the lifecycle service capability, particularly in the areas of analyzing, synthesizing and improving network architectures.

### 3.3.1 The Transport Environment

In this section we describe our initial implementation of the Genesis Kernel framework for PC-based routers. The transport environment has been implemented in user space using dynamically linked libraries (i.e., shared libraries and DLLs) in the FreeBSD and Windows NT operating systems. The transport modules have been implemented as C++ objects. Our implementation balances the flexibility of user-space development [136] against the performance issues associated with the lack of high-resolution timers and context switching. The transport environment is derived from the BSD kernel implementation of TCP/IP. The networking code was extracted as transport modules and used as a basis for implementing a programmable IP datapath. A parent network architecture was developed in this manner supporting the majority of features found in IP [115]. In addition, we modified the Mobiware and Cellular IP software distributions [30] [32] to create child network architectures. Figure 8 illustrates the implementation of an IPv4 routelet.

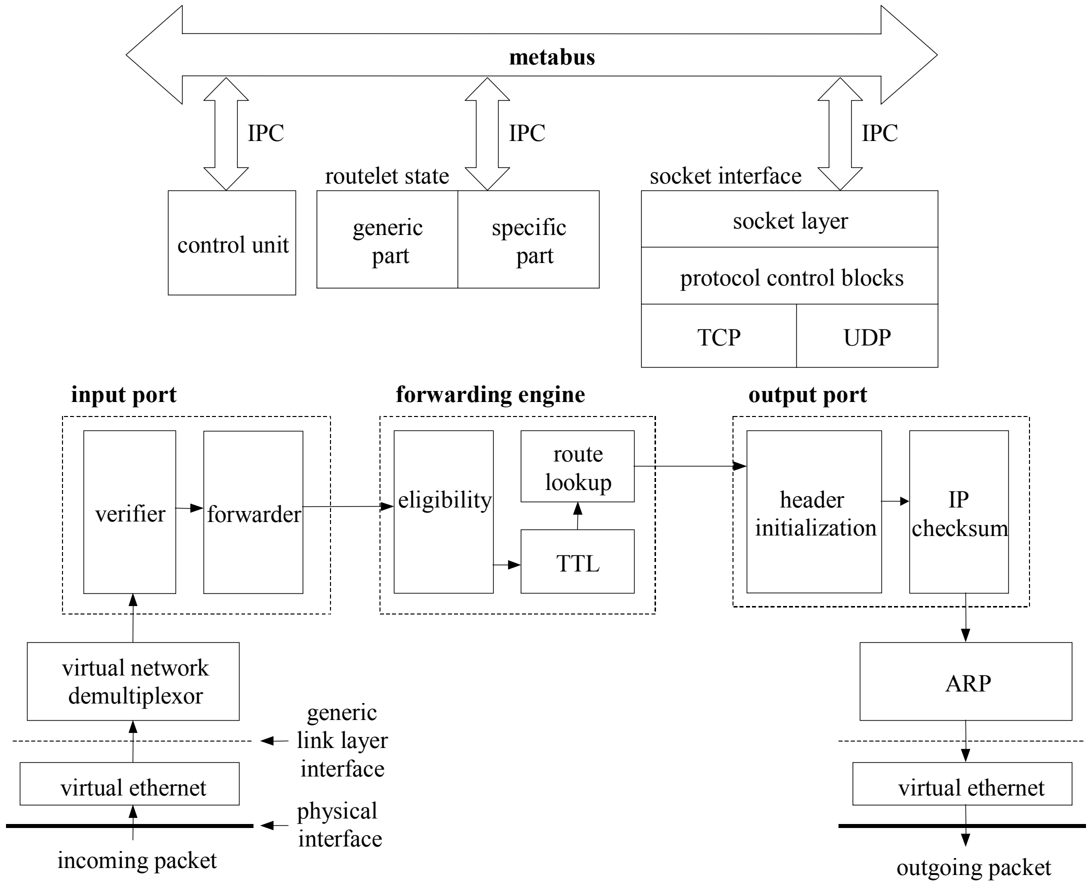


**Figure 8: IPv4 Routelet Implementation**

**Link Layer Support**

The Genesis Kernel separates the link and internetworking layers through a generic link layer interface, as illustrated in Figure 8. We have taken care to decouple the data structures describing the link and internetworking layers. Information associated with the layer two interface is managed by link layer modules while information associated with the layer three interface is managed as part of the routelet state, as is the case with the IPv4 routelet. Typically, spawned virtual networks transmit packets through their parent network's capacity arbitrators. Only the transport environment of the virtual network at the 'root' of the inheritance tree (i.e., the root network) needs to interact with the physical link layer.

The link layer interface supports generic methods for sending and receiving frames and configuring the link layer software. In Figure 8, the link layer modules represent virtual Ethernet modules. We use the term 'virtual' in this context because link layer modules use low-level programming APIs to send and receive frames to and from the network device drivers. For example, we have used the BSD Packet Filter (BPF) as a network programming API in FreeBSD.

**Packet Flows**

At a router, packets are forwarded from incoming to outgoing physical interfaces traversing virtual network demultiplexors and routelets. Memory management is realized as follows. Transport modules can drop packets when and where needed (e.g., a queue may drop a packet if the length of the queue exceeds a given threshold). In addition, allocation controllers enforce hierarchical memory management according to the virtual network inheritance tree. Virtual network demultiplexors configure link layer modules controlling the devices where packets are received from.

**Routelet Components**

Routelet components are shown in Figure 8. Ports and engines are modular elements that perform basic functions on packets. In the current implementation IP option processing and fragmentation and reassembly mechanisms have not been implemented. The verifier module inspects the IP header to determine if the header of an incoming packet is valid. The forwarder module checks whether a packet has reached its final destination or not. The eligibility module checks whether a packet is eligible to be forwarded. Link level broadcasts, loopback packets and packets addressed to class D and E destinations are dropped. The TTL module decrements the TTL field in the packet header. After this processing, the forwarding engine performs a route lookup to determine the packet's outgoing interface. The output port accepts packets from forwarding engines and higher level protocols. If a packet is received from a higher level protocol, the output port initializes the packet header using the header initialization module. The IP checksum module computes the IP header checksum. Finally the packet is forwarded to an ARP module, which performs layer two address resolution that takes into account the specific link layer technology used. An ARP module is selected when the root network is bootstrapped on to the hardware.

**Routelet State**

Routelet state comprises a virtual network generic part and virtual network specific part. The generic part includes pointers to all transport modules that are used by a routelet and a script that reflects the composition of routelet ports and engines. The composition of the specific part is dependent on the particular routelet being programmed. In the case of the IPv4 routelet shown in Figure 6, the virtual network specific part contains information associated with the routelet's interfaces and the routing tables used for IP forwarding.

**Transport Protocol Stacks**

In many cases routelets subsume transport protocol stacks. For example, the IPv4 routelet supports TCP and UDP protocol stacks. TCP is used for exchanging signaling messages for routing, resource reservation, and control and management. Routelet support for TCP communications is similar to the BSD kernel implementation. The socket layer realizes high-level communication functions such as connection establishment and release. Programmable network objects use Inter-Process Communication (IPC) to interact with the socket layer, where IPC is used as a replacement for system calls that an operating system kernel employs to transfer control to the protected environment of the kernel. TCP port numbers can be re-used over multiple TCP connections provided that these connections are realized by different routelets. Routelets use different IPC channels, which are created dynamically during the spawning phase. The socket layer is not the only component of a routelet that uses IPC. Routelet controllers (i.e., spawning, composition, allocation and datapath controllers) and the routelet state management also use IPC.

## 3.3.2 The Programming Environment

**Metabus**

The metabus comprises an orblet component and set of metaservers, as illustrated in Figure 9. The orblet represents the metabus component that provides a communication medium between object clients and servers. Current ORB implementations are tailored toward a single monolithic transport service. This limitation makes existing CORBA implementations unsuitable for programming virtual network architectures that may use different transport environments. To resolve this issue we have implemented the 'acceptor-connector' software pattern [121] in the orblet. The acceptor connector pattern wraps low-level connection management tasks (e.g., managing a TCP connection) with a generic software API. The orblet can use a range of transport services on-demand in this case. To use a specific transport service, the orblet dynamically binds to an inter-ORB protocol engine supported by the Genesis Kernel. We have created an IIOP protocol engine for interacting with IP-based routelets.

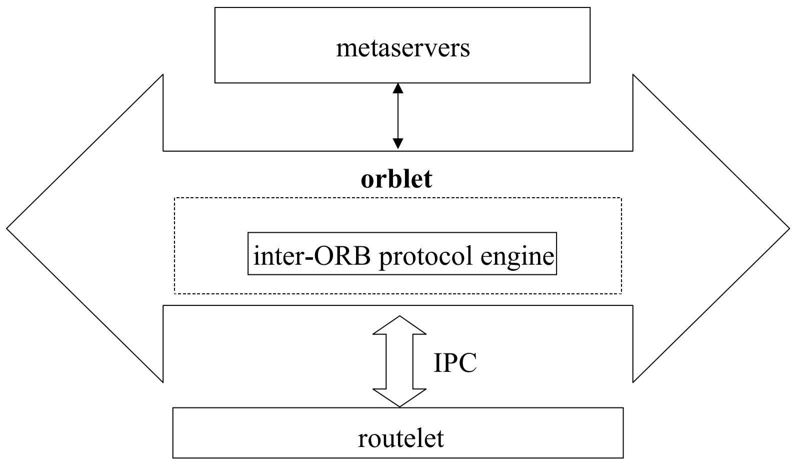


**Figure 9: Metabus Architecture**

Metaservers provide naming services for the metabus. The kernel automates the process of creating naming services and associating naming services with objects. Currently, the reference to a naming service is hard-coded in existing CORBA programming environments. In contrast, metaserver references are dynamically passed to objects during the spawning phase, where metaservers communicate using their spawned transport environment. In this manner, isolation between distinct sets of architectural objects that define spawned network architectures is maintained by metabuses. In summary, isolation between virtual networks is realized as follows. Each metabus uses a separate transport environment for object interaction where the transport environment is dependent on the spawned network architecture. Each metabus offers dedicated naming services to the spawned network architecture.

The orblet is implemented using the OmniORB [97] from AT&T Research Labs, Cambridge, which represents a lightweight CORBA implementation. Currently, we use a single metaserver per spawned virtual network. In the future, we plan to use multiple metaservers and develop metabridges that would support interaction between different virtual networks.

**Binding Interface Base.**

The binding interface base shown in Figure 6 represents a collection of interfaces for programming network architectures. CORBA/IDL is used for describing object interfaces. The following interfaces are common to all routelets:

- a VirtualSpawningController interface, which abstracts the spawning controller, is used for creating new routelets and querying configuration information associated with a spawned virtual network (e.g., routelet specific IPC channel identifiers);
- a VirtualCompositionController interface, which abstracts the composition controller, is used to modify routelet ports and engines, and to access system parameters that characterize the operational behavior of the transport modules. The structure of ports and engines is captured by composition scripts which are exchanged between the VirtualCompositionController object and higher level objects that use the interface; and
- a VirtualAllocationController interface, which is used to access resource allocation information associated with a spawned virtual network. Typically, allocated resources include communication (i.e., link capacities) and computation (i.e., memory and CPU) resources. Currently, only memory allocations are supported by the Genesis Kernel.

The VirtualRouteletState and VirtualDatapathController interfaces illustrated in Figure 6 are specific to the network architecture being programmed. For example, the IPv4 routelet supports interfaces for the configuration of virtual links and the insertion and removal of routing table entries. In this respect, the binding interface base replaces the ‘ioctl’ function calls and routing sockets used in the BSD networking code distribution.

## 3.3.3 The Life Cycle Environment

**Profiling Service**

The current release of the Genesis Kernel only supports a subset of the virtual network requirements discussed in Section 3.2. The profiling of the communication protocols, network services, address space and topology, which characterize spawned virtual network architectures are supported. However, other virtual network requirements (e.g., security, QOS) are for further study.

An overview of the profiling process is illustrated in Figure 10. The profiling process separates the 'binding rules', which define the transport, control and management systems (e.g., a rule for placing a bandwidth broker inside the network), from the 'binding data' (e.g., system parameters, user preferences, etc.). Spawned virtual networks represent the instantiation of a set of binding rules over binding data and are composed using profiling scripts. A profiling script is written in two distinct forms:

- a compact form, which is the form that the network designer uses to specify an architecture and where the separation between binding rules and binding data is applied; and
- an analytical form, which is an internal representation that the Genesis Kernel uses to drive the spawning process.

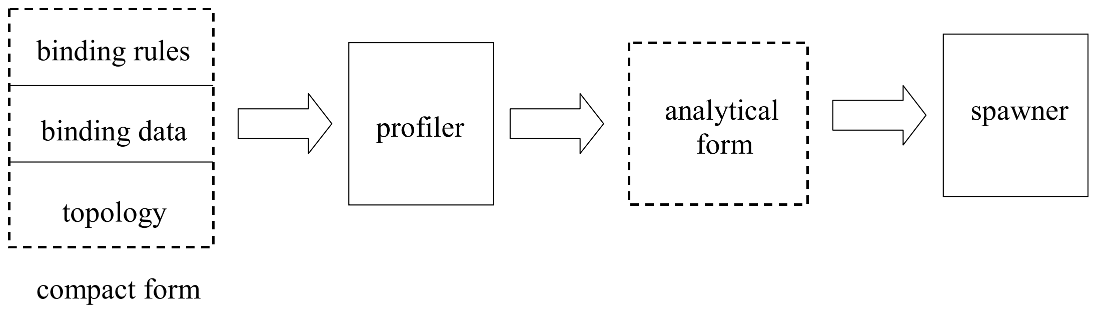


**Figure 10: Profiling Process**

As illustrated in the figure, the compact form comprises three parts. The first part represents a set of binding rules characterizing the composition of routelets and higher level protocols. Binding rules specify which components should be used for constructing a network architecture and arguments used to initialize these components. The binding rules also specify which architectural components are inherited from the parent network. The second part of the compact form represents binding data that captures the arguments that customize the architectural components of virtual networks. The binding data defines the operating point within the network design space for a particular spawned network architecture. The third part of the compact form defines the virtual network topology and address space. The topology is specified as a virtual network graph where all virtual links are annotated and network node addresses declared. Collectively, these three parts of the compact form specify a virtual network architecture in terms of its protocols, services, topology and address space.

The compact form is not well suited to drive the spawning process for a number of reasons. First, the compact form may be syntactically incorrect. Second, the virtual network topology is specified using the addressing scheme of a child network not a parent. Parent network addresses are needed for spawning a child network because the spawning service is supported by the parent network's kernel. Binding rules and binding data need to be associated with each other so that the spawning service can create new communication services at network nodes in a parent network. Given these comments, the profiling service converts the virtual network script from a compact to an analytical form.

There are various steps involved in the script conversion process. First, the profiling service converts the topology description from the child's address space to the parent's address space. This may involve the selection of parent virtual links that satisfy a given set of constraints. As described by the Genesis Framework, the profiling service interacts with the parent network's resource management system to allocate link resources for child networks. Currently, we have not addressed topology conversion and resource management issues. Once the child's network topology has been converted and mapped to its parent's network topology the profiler associates binding rules with binding data. The compact form groups binding rules according to the type of node they describe (e.g., an edge router, core router or base station). To produce the analytical form, the profiler combines the binding rules with topology and binding data, customizing each node in a spawned virtual network with a specific set of parameters. The node type is used as a key for associating binding rules with binding data. Because virtual network architectures are characterized by a finite set of binding rules and network nodes, the complexity of associating binding rules with binding data is polynomial as a function of the number of nodes in the virtual network graph and the number of node types. The conversion to the analytical form results in the creation of separate scripts that describe each network node in the spawned network architecture. Scripts are sent to all parent nodes associated with a spawned child network. A separate script specifies the bindings that take place across child routelets (e.g., bindings between network control and management objects).

```
<?xml version="1.0"?>
<compact_form>
<binding_rules>
<architecture>"cip"</architecture>
  <node_types>
  <type>
    <name>"base_station"</name>
    <data>
      <parameter>"number_of_leaves"</parameter>
      <parameter>"root_address"</parameter>
      <parameter_array>
        <length>"number_of_leaves"</length>
        <parameter>"leaf_addresses"</parameter>
      </parameter_array>
      <parameter>"soft_state_timer"</parameter>
      <parameter>"delay_buffer_size"</parameter>
      <parameter_array>
        <length>"number_of_leaves" + 1</length>
        <parameter>
          <name>"vn_demuxors"</name>
          <type>VN_DEMUX</type>
        </parameter>
      </parameter_array>
      <parameter_array>
        <length>"number_of_leaves" + 1</length>
        <parameter>
          <name>"arbitrators"</name>
          <type>ARBITRATOR</type>
        </parameter>
      </parameter_array>
    </data>
  </type>
  </node_types>
  <routelets>
  <routelet>
    <name>"cip_routelet"</name>
    ...
```

**Figure 11: Profiling Script: A Snippet of the Binding Rules for the Cellular IP Network Architecture.**

We have completed the first version of the profiling service. Both the compact and analytical forms are written in XML, which is suitable for describing information structures. We have used a limited XML grammar with tags for declaring architectural components and their parameters and bindings. To associate binding rules with binding data, we manipulate tree structures derived from profiling scripts. The profiler performs the association between the different parts of the profiling script to produce the analytical form. The profiler has been developed using Expat [51].

Figure 11 shows a snippet from the binding rules describing the Cellular IP network architecture. The snippet describes how the Cellular IP routelet is parameterized. The profiling XML grammar allows for the composition of network architecture components including ports, forwarding engines, routing daemons, handoff controllers and mobility agents. The profiling service is far from complete, however. The profiling service applies syntactic control over scripts but not semantic control. A syntactically correct script may hide erroneous object bindings. Object bindings are resolved during the spawning phase, however. An incorrect profiling script would result in the termination of the spawning process. A more important issue is associated with the capability of the kernel to determine whether a profiled network architecture satisfies the needs of the users that the architecture was spawned for.

**Spawning**

The spawning process, illustrated in Figure 12, is initiated once the analytical form is generated. Spawning services include the following:

- a spawner service, which applies centralized control over the spawning process interacting with the profiling and management services;
- a component storage, which represents a distributed database of virtual network software building blocks; and
- a set of constructor objects, which run on all nodes in a parent topology and interact with the spawner to create a child network.

Constructors support the creation of routelets, the instantiation of a metabus and the deployment of child network architecture objects on a single network node. The spawner is a distributed system, which controls the spawning process, through the execution of a profiling script. We currently use a single spawner object in our spawning networks testbed. The component storage represents a database for transport modules and network objects. The spawner "announces" the child network's bandwidth requirements to a virtual network resource manager. The resource manager is associated with the virtuosity kernel plug-in [28] and represents a distributed controller, which performs admission testing for child networks. If the admission test is successful the child network is spawned.

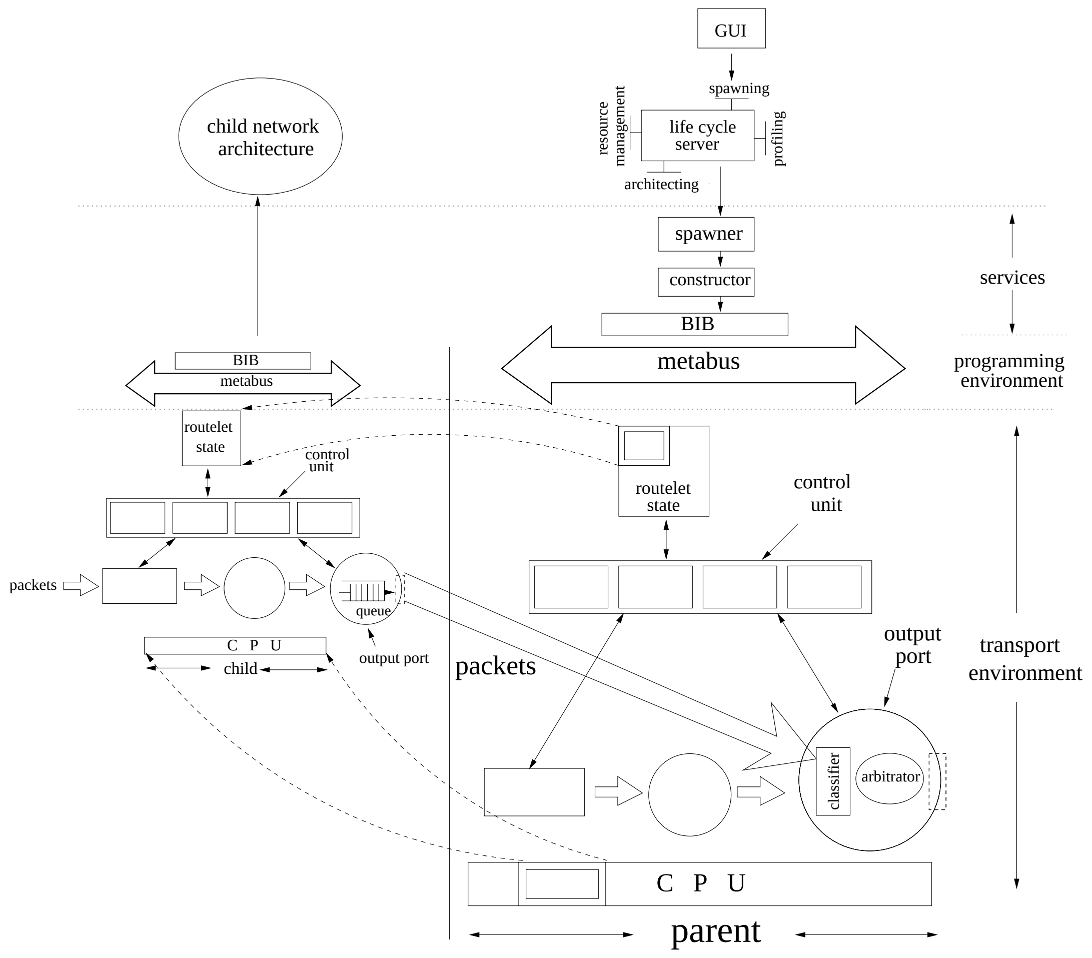


**Figure 12: The Spawning Phase**

Child routelets are bootstrapped by the parent’s spawning controller. The spawning controller interacts with the allocation controller to reserve parent routelet’s computational resources for the execution of a child routelet. Following this, the child routelet’s state information is initialized. During this phase of the spawning process a spawner acquires all the necessary transport modules that were not available at its local node. Transport modules are stored in a component storage as dynamically linked libraries and metabus objects. When the initialization of the routelet’s state is complete, the child control unit is spawned. During this phase the standard controllers are created, specifically the spawning, composition and allocation controllers.

When the bootstrapping process is complete the child routelet is capable of undertaking all the remaining spawning tasks. The composition of a routelet’s ports and engines is carried out by the child’s composition controller. Finally, the child network's data path controller is composed and its queues configured to forward traffic to the parent network queues. This represents the last phase of the spawning process where routelets bind to virtual links forming a virtual network topology. Currently, we use FCFS queues as capacity arbitrators [28]. Virtual network capacity scheduling has been investigated in [28, 132]. Following the creation of the transport environment, the spawning process creates the programming environment and instantiates the child network architecture objects (i.e., network control and management objects). At this point the child network is executing on the Genesis Kernel and the network hardware.

## 3.4 Programming Network Services

In this section we describe how network services can be programmed using the Genesis Kernel. We focus on routing. Routing protocols differ significantly. For example, consider two well-known intra-domain routing protocols: RIP and OSPF. Although these protocols both operate on the same type of networks and their task is similar (i.e., to route packets based on the optimal path between a pair of nodes) each protocol takes a distinct approach to the formulation of forwarding tables. RIP performs the computation of the optimal routes in a distributed fashion between autonomous system (AS) routers using the Bellman-Ford algorithm [75]. OSPF floods information about adjacencies to all routers in the network where each router locally computes the shortest paths by running the Dijkstra's algorithm. BGP, the inter-domain routing protocol, is based on a different design approach, where backbone routers exchange routing information in terms of path vectors [75]. Path vectors are used for loop prevention and policy-based routing and include the complete list of autonomous systems to each destination.

### 3.4.1 Binding Model

To program routing services, we introduce a binding model that decomposes routing protocols into fundamental building blocks, define programmable objects for routing protocols and use these objects to construct well known (i.e., distance vector, link state, path vector) as well as new routing services. To formulate our binding model we have studied the design and implementation of existing Internet routing protocols [75]. We have found a set of attributes that are common to routing protocols and identified the role of each attribute in the process of formulating forwarding tables.

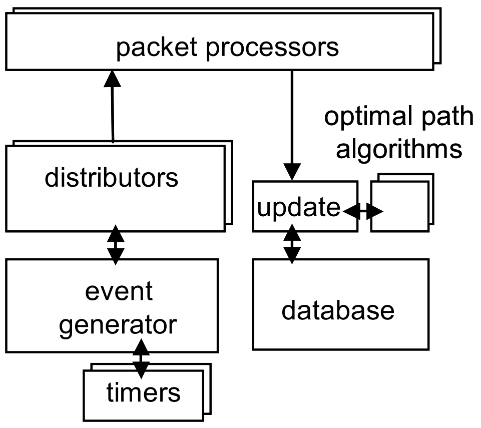


**Figure 13: Binding Model for Routing Protocols**

We observe that routing protocols use some type of database. This can be a link state database (e.g., as in the case of OSPF), a distance vector database (e.g., as in the case of RIP), or a path vector database (e.g., as in the case of BGP). In addition, routing protocols use some mechanism for announcing routing information inside the network. This mechanism can be link state flooding such as in the case of OSPF or periodic announcements to neighbor routers as in the case of RIP. Finally, routing protocols use different algorithms and metrics for calculating optimal paths to various destinations. We believe that routing protocol implementations should separate the routing database from the routing information announcement and the optimal path calculation introducing standard interfaces between these components. This technique allows protocol developers to create routing architectures in a modular fashion, and Internet Service Providers (ISPs) to introduce new routing services into their networks more dynamically.

A binding model characterizing each routing protocol is illustrated in Figure 13. The binding model reflects the structure of routing protocol implementations in a single network node. The components of our binding model include:

- a database object, which is used for maintaining distributed routing state. It can represent a distance vector, link state or path vector database;
- an update object, which updates the database when new routing information is received. The manner in which the database is updated is dependent on the routing protocol being programmed;
- a set of optimal path algorithms, which operate on the contents of the database to calculate forwarding tables. Each algorithm may use a different metric and operate over a different timescale;

- an event generator, which initiates the transmission of routing information, updating of the database or the calculation of optimal paths. The event generator is programmable and can be triggered by timer objects;
- distributors, which disseminate routing information in the network. Distribution systems can simply send updates periodically as in the case of RIP or implement complex protocols such as the 'hello' and 'flooding' protocols as in the case of OSPF; and
- packet processors, which process routing packets before they are transmitted to the network or after they are received from the network.

By introducing standard interfaces between these routing components we enable code reuse and ease the process of developing new routing protocols. Component interfaces are independent of the specific routing protocols being programmed. By allowing components to create bindings at run-time we enable the dynamic introduction of new routing services into networks.

### 3.4.2 An SDK for Routing Protocols

To realize our binding model we have designed a routing SDK consisting of a hierarchy of base classes, as illustrated in Figure 14. Our SDK called ‘Routerware’ consists of classes can be implemented using object oriented programming languages and environments such as C++, CORBA and Java. Currently, Routerware runs on top of the metabus, described in Section 3.3.

**Generic Classes**

Routerware consists of three groups of classes. At the lowest level, a set of generic classes offers basic functionality to the layers above. A database class supports generic methods for creating new databases and for adding, removing or searching entries. An authentication algorithm class implements a set of authentication algorithms used by routing protocols (e.g., the MD5 algorithm), while a timer management class supports a simple start/stop/reset timer. The network connection class is more complex since it provides the mechanisms used by the protocol at the lowest level for transmitting routing messages to the network. A range of diverse communication mechanisms are supported including TCP and UDP socket management and remote method invocations. Because many routing protocols are multithreaded, a thread management class wraps operating system specific methods for creating and managing threads.

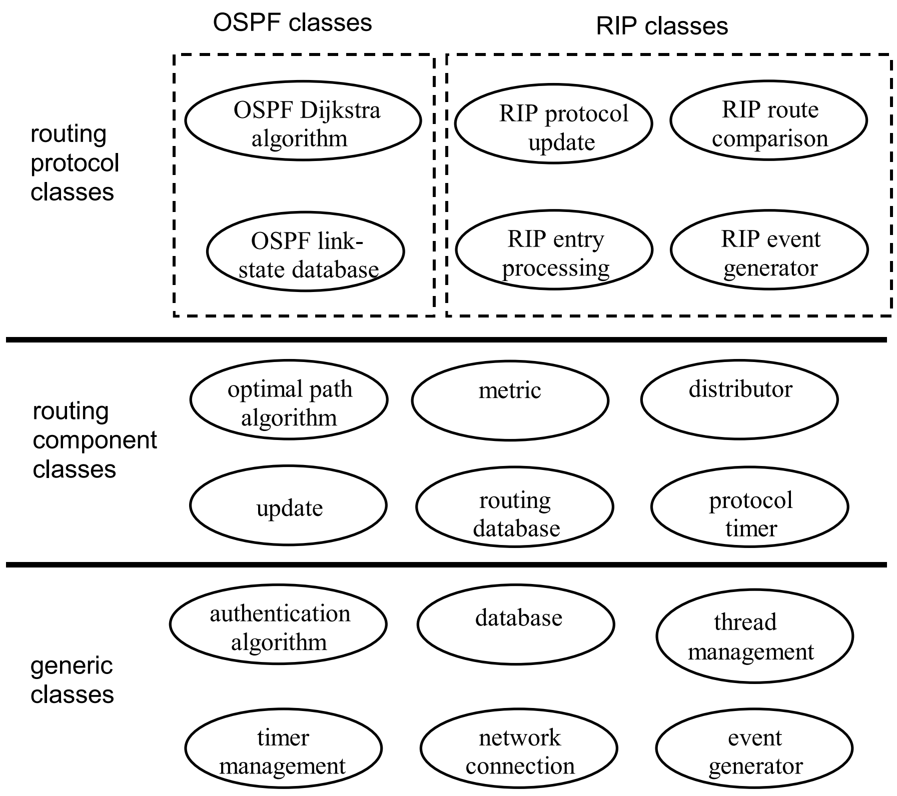


**Figure 14: Routerware**

**Routing Component Classes**

The next layer of abstraction contains routing component classes. Routing component classes represent building blocks that can be used to construct different routing architectures. Routing component classes are either new classes or extend the generic classes described above. A routing database extends the generic database class encapsulating all the route entries available at a network node. The routing database class provides information about network nodes and their associated paths, which can be used for performing route entry lookups. We use a generic data structure to represent a routing database entry. An update class performs the steps necessary for a generic type of routing update. A distributor class retrieves all the route entries that have to be transmitted, sets up the necessary network connections, and finally sends the update information. This is a generic class that can be further extended or used as part of more complex protocol-specific distributor classes.

A metric class defines different types of metrics and a set of fuzzy comparison methods for these metrics. Different types of metrics can be combined to support customized routing policies. A protocol timer class extends the generic timer class to support a range of timer events used by routing algorithms. Timer objects bind to routing protocol specific routines that handle events generated by timers. An optimal path algorithm class provides abstractions used for configuring algorithms that calculate forwarding tables. Programmers can use this class to specify families of protocols associated with optimal path algorithms, types of metrics or database fields. Optimal path algorithm class is an abstract class. Each routing protocol is expected to extend this class in order to define the core of the algorithm.

**Routing Protocol Classes**

Routing protocol classes use the remaining Routerware classes to implement specific routing protocols. Routing protocol classes fully describe the objects that participate in a routing protocol implementation. Examples of these classes, specific to the RIP and OSPF protocols, are illustrated in Figure 14. A RIP protocol update class implements routing updates that are specific to RIP. Another class could implement link state updates specific to OSPF. Each protocol is associated with a protocol entry processing class. For example the RIP entry processing class supports methods for processing RIP route entries received from the network. The operations implemented here concern route entry processing only. The reception of packets containing these entries and the extraction of entries is handled by other generic classes described above.

Protocol-specific event generator classes (e.g., the RIP event generator class) interact with timers to generate events when needed. Timer-related constants can be set and the distribution of the interval between two subsequent events specified in case this interval is stochastic. Event generator objects can be configured with information about the actions taken in the case of various events. For example, a RIP event generator can be programmed to call a regular update when the corresponding timer expires. Finally, route comparison classes provide methods for comparing route entries. Route comparison classes make use of metric classes to produce results. Optimal path algorithm classes (e.g., the OSPF Dijkstra algorithm class) make use of metric classes.

The classes described above can be used by many well-known routing protocols. Some routing protocols require additional components. In the case of link-state protocols for example, a set of classes describing flooding mechanisms are needed. Flooding protocol classes extend a generic distributor class (shown in Figure 14) using functionality provided by the lower layers. For example, a flooding protocol class can make use of a database class in order to retrieve entries for transmission. Similarly, a flooding protocol class can make use of a network connection class in order to create the necessary end-to-end connections to transmit routing entries.

### 3.4.3 Routing Protocol Composition

In what follows, we show how the building blocks discussed in the previous section can be used to construct routing protocols. We begin by examining intra-domain routing protocols focusing on RIP as an example. We then discuss inter-domain routing protocols focusing on BGP.

**Intra-domain Routing**

The part of a programmable RIP implementation (shown in Figure 15) that directly communicates with the network is the network connection object. The network connection object uses the metabus to send or receive data to and from other routers. Another variation of RIP could use UDP sockets or remote method invocations of other types (e.g., CORBA IIOP, Java RMI). In the case of sockets, a RIP packet processing object is involved which reads routing protocol specific headers from received packets and applies appropriate incoming filters in order to verify the origin and the validity of the received packets. In the case of remote method invocations, content processing is not applied to headers but on the arguments of remote method invocations. An authentication password is passed to the object that implements authentication algorithms in order to perform validity checks and notify the RIP packet processing object. The algorithms implemented can be simple password or MD5 cryptographic checksum. These algorithms are used by many routing protocols including EIGRP, OSPF and RIPv2.

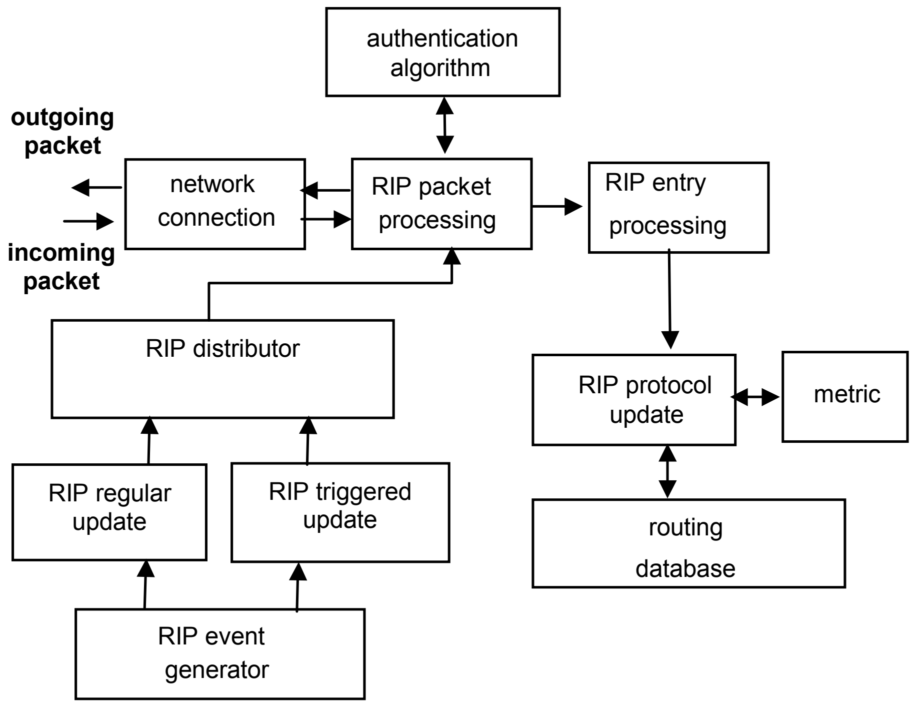


**Figure 15: Programmable RIP**

Once a packet has been examined and verified a route entry is extracted and sent to the RIP entry processing object. The RIP protocol update object is then invoked, which in turn exchanges information with the metric object and the routing database. The routing database replies with the route entry for the same node (if one exists). Following this, the metric object inputs the two entries and produces the comparison result based on a specified set of metrics. If the received entry is better than the existing one the routing database object is used to replace the existing entry. Following this, the RIP event generator initiates a RIP triggered update. Updated entries are finally transmitted to the neighboring routers via the network connection object.

Regular RIP updates are periodically initiated using a distributor object, as shown in Figure 15. The distributor object enforces the 'split horizon' [75] principle to the way routing entries are announced in the network. The split horizon principle suggests that announcements concerning destinations should not be sent to routers that are in the shortest paths to these destinations. This feature makes RIP more robust against routing loops and link failures.

Link state routing architectures can be realized using the same binding model and base classes as in the case of RIP. The mechanisms for distributing and processing route entries are different, however. For example, an incoming OSPF packet needs to traverse a packet processing object specific to OSPF. State advertisements need to be extracted and entered in a link-state database. A shortest path algorithm object can apply Dijkstra's algorithm to the contents of the link state database in order to calculate forwarding tables. If there is a need for an update, a flooding protocol can make use of the network connection object to communicate with other routers in the network.

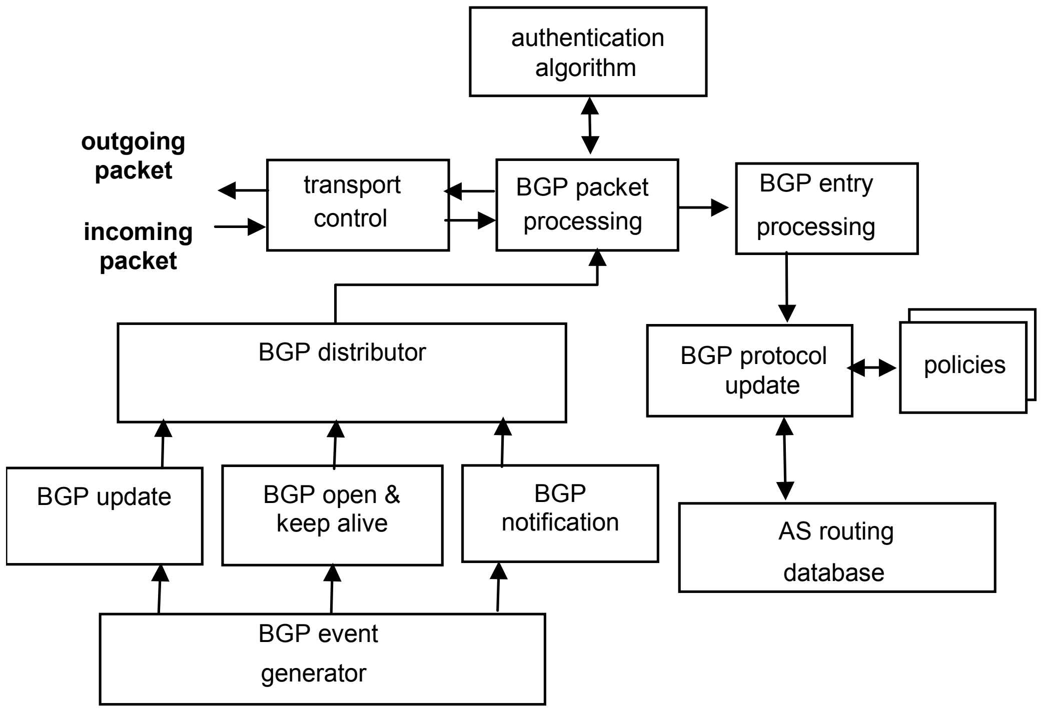


**Figure 16: Programmable BGP**

## Inter-domain Routing

Inter-domain routing protocols can be implemented by combining objects from Routerware, as shown in Figure 16. In the case of BGP, a transport control object extends the network connection class, providing the reliable data transfer mechanisms required by BGP messages. The transport control object receives path entries from the network in the form of BGP update messages. Path entries are processed in a manner similar to the way RIP processes route entries. Each entry is compared with the corresponding entry for the same destination at the local AS routing database. The AS routing database maintains information about destination autonomous systems, their associated paths and their path attributes. The shortest path to a destination is selected by default. Alternatively, the network administrator can program customized policies for selecting the best path. Policies can be implemented as Routerware objects that dynamically bind to the BGP protocol as shown in Figure 16. A BGP protocol update object compares path entries selecting the best path to an autonomous system.

Other features of the BGP protocol are supported as separate Routerware objects. Open, update, keep alive and notification messages are sent by their associated Routerware objects, as shown in Figure 16. Protocol enhancements can take place with simple software upgrades. For example, one can add support for classless inter-domain routing (CIDR) into BGP by replacing the BGP update and entry processing objects shown in Figure 16. These objects can be replaced by other objects that exchange network layer reachability information (NLRI) [75] in addition to the path entries contained in the update messages. Network layer reachability is a feature introduced in the fourth version of BGP for route aggregation.

## 3.5 Experiences

In order to evaluate the Genesis Kernel we have built a spawning networks testbed and designed a set of experiments to verify the kernel's capability to dynamically create, manage and architect the baseline network architectures. We have evaluated the performance of the spawned baseline architectures against the performance observed when the same architectures are implemented in 'standalone' testbeds. The goal of the evaluation is more qualitative in nature and aimed at showing proof of concept rather than a quantitative comparison.

### 3.5.1 Spawning Networks testbed

The spawning networks testbed has been built using PC-based routers and network processor-based routers. In what follows we describe experiments using PC-based routers. The performance of the Genesis Kernel on network processor-based routers is studied in Chapter 5. The spawning networks testbed has been designed to support the spawning of the baseline network architectures. We deployed a parent network architecture that supports IP routing as the root network, as illustrated in Figure 17. Once the root (parent) network was boostrapped onto the network hardware, we were able spawn the Cellular IP [30] and Mobiware [32] child networks over the root network providing wireless data and multimedia services to mobile users, respectively. The Mobiware and Cellular IP architectures were refined (i.e., architected) using the Genesis Kernel by adding new handoff control algorithms to the Mobiware child network and tuning the parameters associated with the Cellular IP child network.

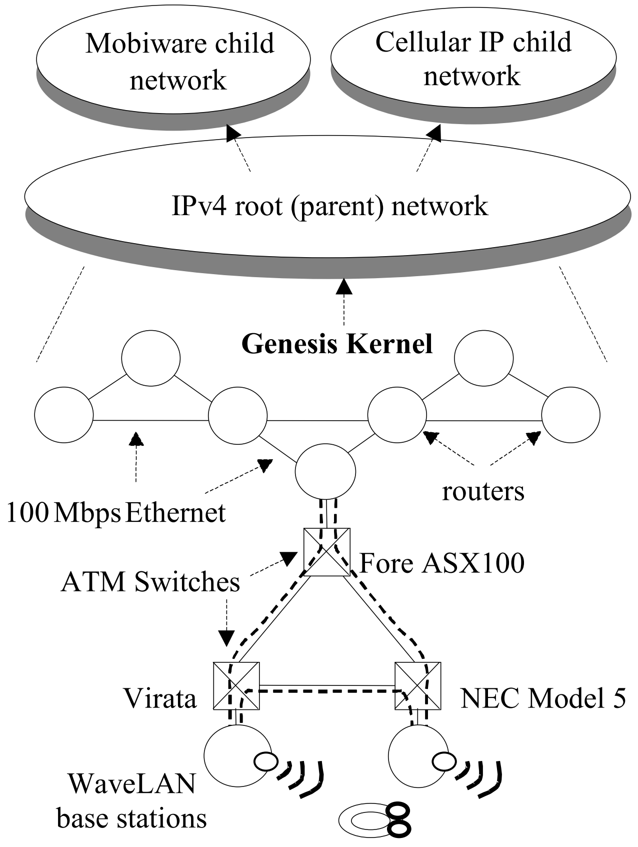


**Figure 17: Spawning Networks Testbed**

The spawning networks testbed comprises heterogeneous link layer technologies that interconnect routers, switches and base stations, as illustrated in Figure 17. The testbed provides wireless access to mobile hosts and comprises seven multi-homed 300 MHz Pentium PC routers, three ATM switches (viz. ATML Virata, Fore ASX/100, and NEC model 5 switches) and two PC base stations. Link interconnects between PC routers, PC base stations and ATM switches comprise 100 Mbps Ethernet links and 155 Mbps wireline ATM links. PC base stations provide radio access to the wireline network. The radios are based on

WaveLAN operating in the 2.4-2.8 GHz band. We use the 2 Mbps WaveLAN cards (not IEEE 802.11 compliant) over the 10 Mbps cards because these cards support a low-level radio utility API for programming beacons.

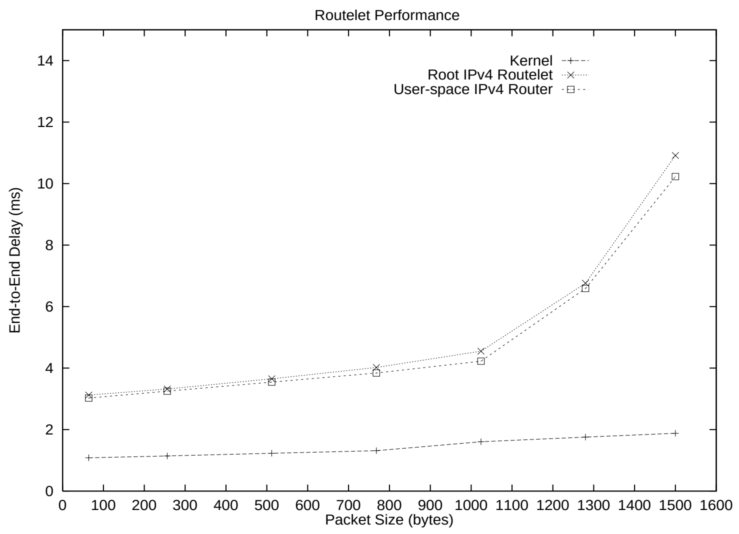
In our experiments, the spawning capability is supported in the network and not in end-systems. Programmability support for the end-systems is discussed in Chapter 5. The Genesis Kernel code has been designed to run on Windows NT and FreeBSD operating systems.

### 3.5.2 An IPv4 Root (Parent) Network

In this experiment we investigate the capability of the kernel to spawn an IP virtual network architecture supporting standard IPv4 packet forwarding, and interior and exterior routing services. We deployed a parent network architecture supporting IPv4 over the spawning testbed as a root network. The architecture consists of the IPv4 routelet discussed in Section 3.2 and a set of distributed objects offering interior and exterior routing services. We have developed an object-based implementation of the Routing Information Protocol (RIP), which is used in the Internet for interior routing, and the Border Gateway Protocol (BGP), which is used for interconnecting autonomous systems, based on our routing SDK. In order to spawn the IP routing architecture we deviate from the standard spawning procedure described in Section 3.3. The reason for this is that there is no communication capability in the network hardware to support the spawning process, (i.e., the root network has no parent). To resolve this problem we have added a 'bootstrap' interface to the Genesis router process. The first architecture "spawned" onto the hardware is actually bootstrapped. Therefore the root network is always a special case in spawning networks.

In Figure 18, we compare three implementations of IPv4 that include a spawned IPv4 root (parent) network and two standalone FreeBSD IPv4 implementations, (i.e., one user-space and one FreeBSD kernel). We include a user-space implementation of IP because routelet forwarding engines are currently implemented in user-space too. It should be noted that one difference between the user-space implementations is that routelets implement a virtual network demultiplexor in the datapath while the standalone user-space IPv4 implementation does not.

Figure 18 shows the end-to-end delay and throughput results across a single hop. We observe that the difference between the routelet and the standalone kernel IPv4 router is 2 ms for small packet sizes increasing to 9 ms when the MTU is 1500 bytes. Figure 18 also shows the throughput achieved by the three IP implementations over a single hop. On average the routelet system attains about 75% of the kernel router throughput. One performance penalty paid by the routelet implementation is associated with copy-in/copy-out operations that take place between when a packet enters and leaves a Genesis router. The next steps of our work included porting the Genesis Kernel to the Intel IXA router architecture [69-72] where the routelet implementation gained performance from executing on the network processor IXP1200 (see Chapter 5).



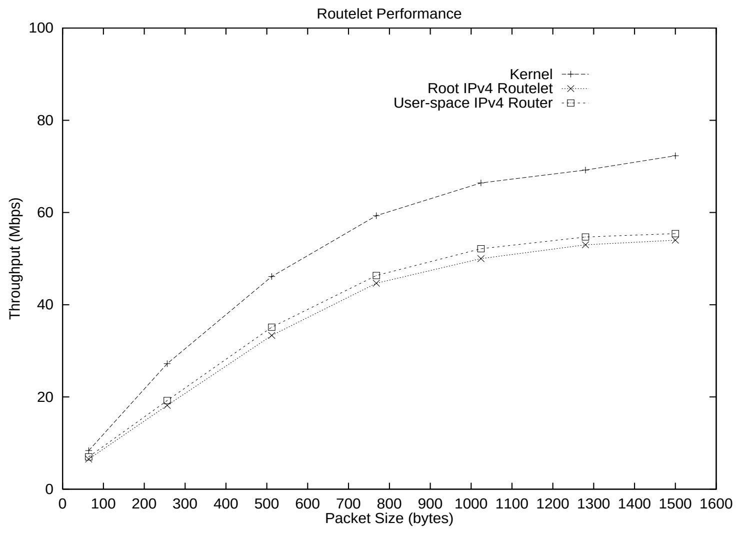


**Figure 18: Routelet Performance**

### 3.5.3 Wireless Child Networks

In this experiment we investigate the ability of the Genesis Kernel to spawn baseline child networks on the IPv4 root (parent) network. This represents one level of nesting and executes the spawning capability, which could not be exercised during deployment of the root network. We spawned Mobiware and Cellular IP child networks on the testbed. Mobiware is specifically designed to support multimedia services with service-level assurances, whereas Cellular IP, is designed to deliver packet data with fast handoff and paging support. The implementation of Mobiware and Cellular IP datapaths is illustrated in Figure 19.

Mobiware [32] is a connection-oriented mobile network architecture that includes session rerouting, mobility state management and wireless transport configuration algorithms. All sessions that operate between a mobile host and its associated Internet gateway are abstracted and represented as a single state entity called a flow bundle. Flow bundles are used during handoff to switch multimedia flows that are supported using an adaptive QOS scheme [94, 95]. Open programmable switches allow for the establishment, removal, rerouting and adaptation of flow bundles.

The Mobiware network uses two distinct types of datapath. An IP datapath for signaling and an ATM datapath for transport. The IP datapath is used for network control and management. The ATM datapath, which is independent of the Genesis transport environment is used for transporting audio and video flows. IP packets do not traverse the Mobiware routelet. Rather, IP packets are forwarded using the ports and engines of the root network, as illustrated in Figure 19. A GSMP client engine is incorporated into the Mobiware routelet and used for controlling the ATM switches in the spawning networks testbed. The GSMP engine does not receive packets from virtual network demultiplexors but communicates via ATM sockets with ATM switches in the network.

Cellular IP [30] is a packet-based mobile network architecture that is designed to give high performance delivery of data with fast handoff and scalability support through paging. In Cellular IP, packets sent from a mobile host create a soft-state routing path between the mobile host and its Internet gateway. The wireless access network maintains mobile-specific routing cache in support of fast handoff and paging cache to track idle mobile hosts.

The Cellular IP routelet comprises an ‘uplink’ interface and a set of ‘downlink’ interfaces. The uplink interface connects the routelet with an Internet gateway. The downlink interfaces receive packets from mobile hosts and forward them to the gateway. Each interface is associated with a different forwarding engine. When a virtual network demultiplexor receives a packet carrying the Cellular IP identifier it forwards the packet to the Cellular IP routelet, as illustrated in Figure19. Forwarding engines update paging and routing caches inserting a pointer to the downlink path on behalf of the mobile host that sends the packet.

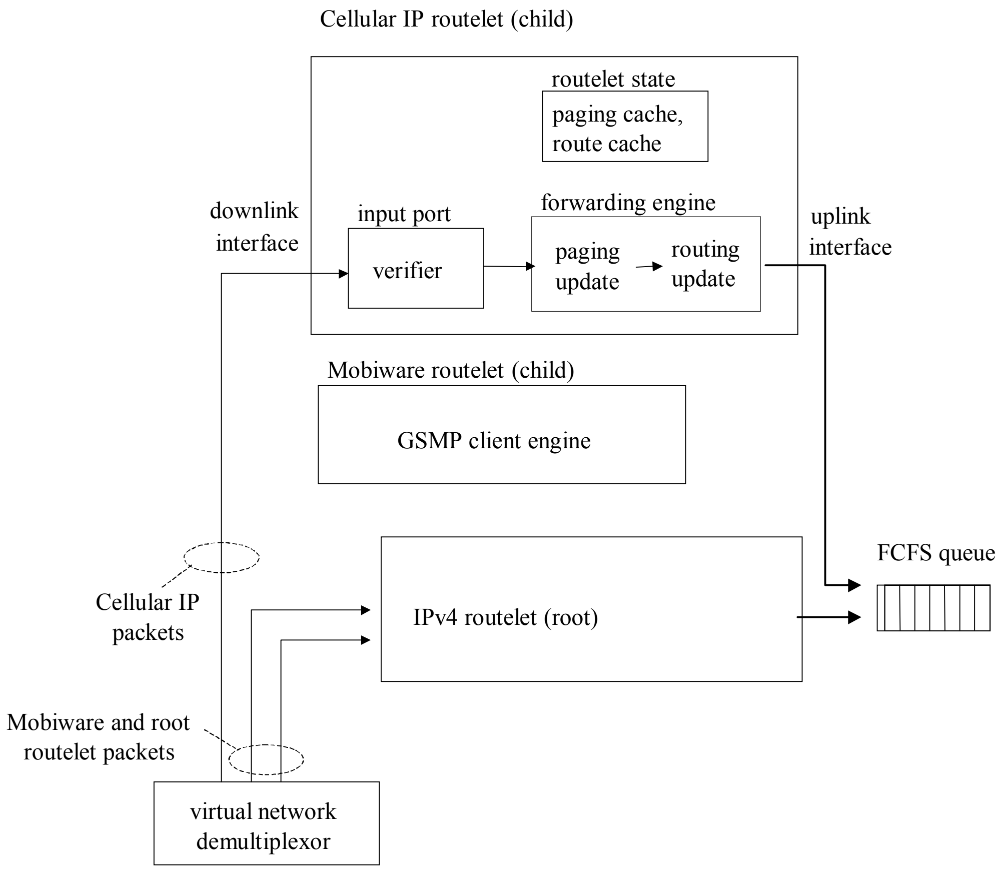


**Figure 19: Mobiware and Cellular IP Datapaths**

Currently, we have not fully implemented the ability of child networks to spawn their own children. This is topic for further study. Therefore, the baseline child networks do not inherit life cycle services, as discussed in Section II. Both child networks inherit the topology and address space of the root (parent) network, however. A mobile host can take advantage of both child networks to receive real-time multimedia and data services. For example, the signaling overhead of the Mobiware child network makes it unsuitable for packet data delivery, whereas the Cellular IP child network does not implement QOS support. A detailed description of the Mobiware and Cellular IP architectures is beyond the scope of this chapter. For full details of their specification, performance and source code release see [32] and [30], respectively.

We have conducted a set of tests that compare the performance of the Mobiware and Cellular IP child networks against their 'standalone' counterparts. In all cases the spawning and standalone testbeds were lightly loaded during the experiments. In order to evaluate the Mobiware child network we streamed a number of video flows to a mobile host and performed continuous handoffs, as shown in Figure 20. We varied the number of flows delivered to a mobile host and measured the average handoff latency for the Mobiware child and standalone architectures. The standalone Mobiware architecture uses OmniORB for object interaction. The main performance difference between the spawned and standalone Mobiware architectures is related to the transport environment used for signaling. The Mobiware child network uses the metabus, whereas standalone Mobiware uses OmniORB and the kernel transport services. The performance results for the comparison are shown in Figure 20. The figure shows the average handoff latency experienced by a mobile host when using the standalone and spawned Mobiware architectures as the number of flows in a flow bundle increases. The plot also shows the performance with and without flow bundling. We observe higher latency in the case of the spawned Mobiware architecture because of the metabus and routelet overheads.

To evaluate the spawned Cellular IP child network we measure the TCP throughput across a Cellular IP virtual wireless link. We compare the TCP performance of the Cellular IP child network with the standalone system. Both the standalone and spawned architectures are implemented in user space. The main difference between the two systems is that the datapath for the standalone system is not burdened with virtual network demultiplexing, as is the case with the Cellular IP child network.

Measurements are taken for the 'hard' and 'semisoft' Cellular IP handoff modes [30], as shown in Figure 20. The hard handoff mode represents a 'break before make' style of handoff where the mobile host switches to the new base station and then forwards a packet to create the new downlink soft-state path between the mobile and the cross over switch. The Cellular IP semisoft handoff improves handoff performance by reducing packet loss during handoff. Before handoff, a mobile host sends a short control message called a semisoft packet to the new base station and then returns immediately to listen to the old base station. The semisoft packet configures routing cache mappings and sets up the soft-state path between a cross over switch and the mobile host. After a very short semisoft delay, the host performs regular hard handoff. In addition, forwarding delay is introduced at the cross over switch in order to compensate for the time needed to accomplish semisoft handoff. We observe from Figure 20 that the child network achieves similar performance to the standalone network architecture over a wide range of handoff rates.

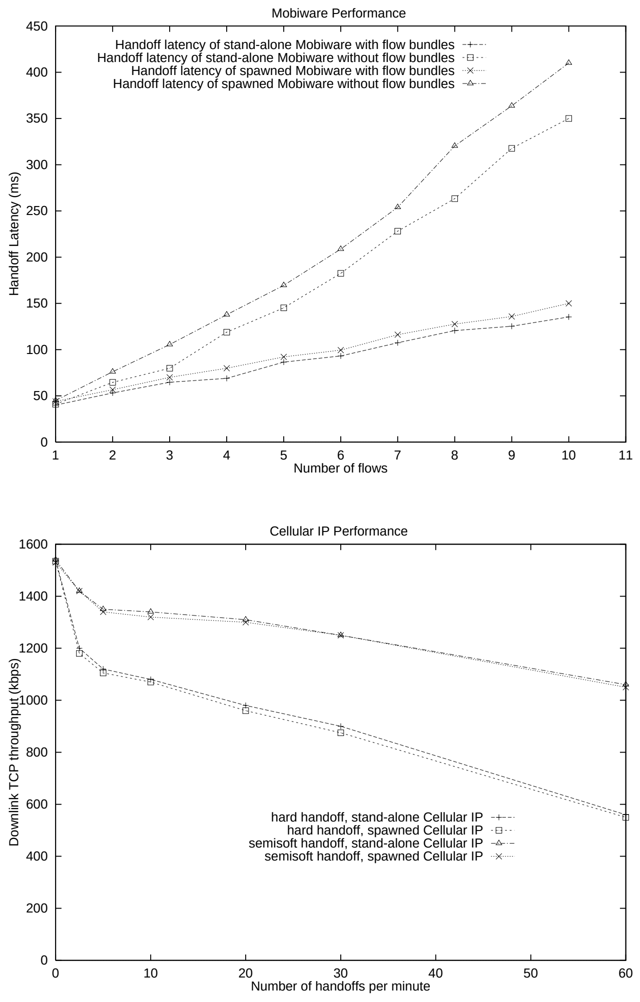


**Figure 20: Performance of Spawned Network Architectures**

## 3.5.4 Architectural Refinement

Currently, the Genesis Kernel does not fully support architecting. However, profiling and spawning tools allow us to experiment with modifying the structure and building blocks of network architectures. In what follows, we discuss two examples of architectural refinement. The first experiment allows different handoff algorithms to be added to a Mobiware child network. The second experiment allows us to refine a Cellular IP child network that improves TCP performance with handoff

Mobiware is designed to support multiple styles of handoff control through the separation of handoff control and mobility management. Handoff control and mobility management systems are implemented as separate programmable architecture, as discussed in detail in Chapter 4. By hiding the implementation details of mobility management algorithms from handoff control systems the handoff detection state (e.g., the best candidate access point for a mobile host) can be managed separately from the handoff execution state (e.g., mobile registration information). In this case, Mobiware allows different styles of handoff control to seamlessly share the same mobility management services. An intermediate layer of distributed objects called handoff adapters serve as the glue between handoff control systems and mobility management services.

Handoff control objects include beacon producer and measurement producer objects, which invoke low-level wireless APIs for transmitting beacons and generating raw channel quality measurements. Signal strength monitor objects collect average wireless signal strength measurements on-demand. Detection algorithm objects make handoff decisions. Handoff control objects can be dispatched to strategic locations in the network (e.g., base stations and mobile capable routers/switches) to simultaneously serve the needs of different handoff styles. The initially spawned Mobiware architecture only supports the mobile controlled handoff style. We modified the Mobiware profiling script to introduce additional handoff styles. The profiling script was modified to include new distributed objects that support mobile assisted and network controlled handoff styles. These two schemes place the complexity associated with controlling handoff into the network. This has the benefit of serving low-power mobile hosts that may not be capable of continuously taking signal strength measurements.

Cellular IP base stations do not buffer packets during handoff causing packet loss and reduced TCP performance. To eliminate packet loss during handoff we have introduced a packet circular buffer called a ‘delay device’ at base stations. The delay device also helps resolve the problem of the new base station ‘getting a head’ of old base stations when using semisoft handoff. A mobile host 'sees' gaps in TCP streams if the forward base station gets ahead of the old base station. This has an adverse impact on TCP performance. The delay device resolves this issue supporting a loose form of synchronization control typically found in cellular systems.

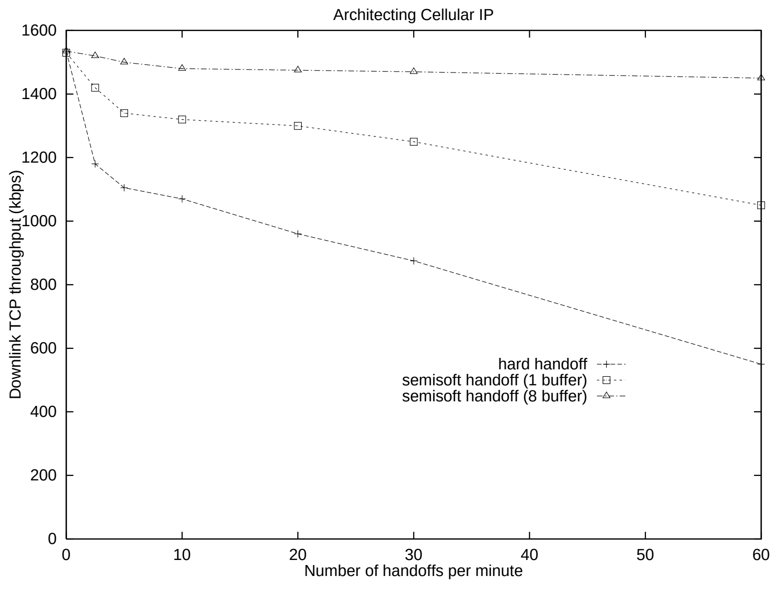


**Figure 21: Architecting Cellular IP**

Figure 21 shows the downlink performance of a TCP flow as the rate of handoff increases. The Cellular IP child network is spawned and supports hard and semisoft handoff capability but has no delay device implemented. The plot shows wireless TCP throughput associated with the initial Cellular IP child network when a mobile host performs hard handoff. Through profiling we modified the original script to include the delay device and spawned a new child network. To add the delay device, we modified the binding model of the Cellular IP root forwarding engine. Instead of using the default routelet lookup module, we introduced a new forwarding element that delays and buffers packets during handoff. The TCP improvement from using the delay device is shown in Figure 21. The figure shows the TCP performance for a delay device that could be programmed to buffer 1 or 8 packets. The plot shows that semisoft handoff outperforms hard handoff. In the case of semisoft handoff, we observe that the deeper the buffer the better the TCP performance. Note that when the buffer is programmed to accommodate 8 packets during handoff we observe that TCP performance is equivalent to the case were the mobile host is stationary. This represents the best possible performance of 1.6 Mbps.

## 3.6 Related Work

The Tempest project [141, 142] has investigated the deployment of multiple coexisting control architectures in broadband ATM environments. Tempest supports programmability at two levels of granularity. First, switchlets are logical network elements that result from the partitioning of ATM switch resources supporting the introduction of alternative control architectures in the network. Second, services can be refined by dynamically loading programs into the network that customize existing control architectures. Resources in an ATM network can be divided by using a switch control interface called a resource divider. In Genesis, the divider mechanism is integrated into the routelet rather than being externally supported as in the case of switchlets. This capability allows a child routelet to spawn its own child networks supporting the nesting principle that underpins spawned network architectures. Routelets apply the concept of resource partitioning to the internetworking layer supporting the programmability of new internetworking architectures with programmable QOS. Routelets are designed to operate over a wide variety of link layer technologies rather than simply ATM technology as is the case with virtual switches [2] and switchlets [141].

Virtual private network services have been the subject of a substantial amount of research in broadband ATM networks. In [37], the concept of a virtual path group is introduced as a virtual network building block to simplify virtual path dynamic routing. In [152], the concept of nested virtual ATM networks is discussed and an architecture that supports resource management of broadband virtual networks presented. The Genesis Kernel framework also uses the concept of "nesting" pursuing the programmability and automated deployment of network architectures spanning transport, control and management planes at the internetworking layer and above. Typically, spawned network architectures support alternative signaling protocols, communications services, QOS control and network management in comparison to parent architectures. A related project called Virtual Network Service (VNS) [96] investigated QOS provisioning in IP virtual networks. The project proposes the partitioning and allocation of network resources such as link bandwidth and router buffer space to virtual networks according to some predetermined policy.

The X-Bone [137] project aims to automate the process of establishing IP overlay networks. Currently, overlays (e.g., M-Bone, 6-Bone, A-Bone) are deployed manually by system administrators and the configuration of tunneled connectivity between routers and hosts that characterize overlay networks is handcrafted. X-Bone constitutes the natural evolution of the M-Bone and uses a two layer multicast IP system to facilitate the dynamic deployment of different overlays in the Internet. X-Bone overlays are not programmable, however. The Supranet [48] project considered a network-less society where networks and service creation are facilitated and tailored to group collaborative needs. A Supranet is a virtual network that requires the definition of the characteristics of the collaborative environment that benefits from the services it provides. Group membership, network topology, resource capacity, security mechanisms, controlled connectivity, and secure multicast represent the requirements for a specific virtual network service to any group.

The active networking community [134, 135] has investigated the deployment of multiple coexisting execution environments through appropriate operating system support and an active network encapsulation protocols. In [135], the use of active networking technology is studied for the deployment of IP based virtual networks. In most of the current research in active networks the dynamic deployment of software at runtime is accomplished within the confines of a given network architecture and node operating system. In contrast, we have investigated ways to construct network architectures that are fundamentally different from their underlying infrastructures.

Perhaps the piece of work that is most related to our own is an attempt to define a unifying set of programming interfaces for networks described in [102]. The main difference between our network programming interfaces and the interfaces suggested in [102] is that in [102] the main abstraction for network service creation is a graph capturing the connectivity that is supported between the users of a network. In our case we do not provide explicit interfaces for graph creation but we support this capability implicitly by allowing the network designers to program distinct routing strategies from a minimal set of routing APIs. In this way the communication graph is created through the interaction of distributed objects programmed using our APIs and classes.

## 3.7 Summary

In this chapter we have presented the design, implementation and evaluation of the Genesis Kernel; a programming system capable of spawning network architectures on-demand. The Genesis Kernel presents a new approach to the deployment of network architectures through the automation of a virtual network life cycle process. We have presented the implementation of the Genesis Kernel and discussed our experiences in building a “spawning” network testbed that is capable of creating distinct network architectures on-demand. Network architectures are created as programmable virtual networks. Our programming system is based on a methodology that allows a “child” network to operate on top of a subset of its “parent’s” network resources and in isolation from other spawned virtual networks. We have showed through experimentation how a number of diverse network architectures can be spawned and architecturally refined.

# Chapter 4

# End-System Connectivity

## 4.1 Introduction

In Chapter 4 we address the problem of supporting end-system connectivity focusing on programmable mobile networks. Programmable mobile networks can be spawned using programming systems such as the Genesis Kernel discussed in Chapter 3. The problem of supporting end-system connectivity with programmable network architectures concerns the design of appropriate end-system and network software support so that hosts can be dynamically connected to programmable networks with diverse sets of protocols for transport control and management. While the problem has a trivial solution for static hosts (i.e., static hosts can install the software support needed for connecting to programmable network architectures off-line), the problem is more interesting in wireless cellular networks. Supporting end-system connectivity in programmable mobile networks is not an easy task because mobile devices may roam across access networks with heterogeneous mobility management architectures. While a variety of handoff algorithms have been proposed and investigated in the past [123, 138, 145] these algorithms are mostly tailored toward the needs of some specific type of mobile device or access network. The diversity in signaling systems that characterize wireless access network architectures poses a challenge in realizing inter-system handoff.

In this chapter we propose a solution to the intersystem handoff problem where the implementation details of mobility management algorithms are hidden from handoff control systems, allowing the handoff detection state (e.g., the best candidate access point for a mobile device) to be managed separately from handoff execution state (e.g., mobile registration information). The same detection algorithms operating in mobile devices, or access networks can interface with multiple types of mobility management architectures, operating in heterogeneous access networks.

The main results of our work are: (i) we present the design, implementation and evaluation of a 'reflective handoff' service that allows access networks to dynamically inject signaling systems into mobile devices before handoff. Thus, mobile devices can seamlessly roam between wireless access networks that support radically different mobility management systems; and (ii) we show how a 'multi-handoff' access network service can simultaneously support different styles of handoff control over the same wireless access network. This programmable approach can benefit service providers who need to be able to satisfy the mobility management needs of a wide range of mobile devices from cellular phones to more sophisticated palmtop and laptop computers. To allow a range of mobile devices to connect to programmable mobile networks we further decompose handoff control process into programmable objects, separating the transmission of beacons, from the collection of wireless channel quality measurements and from the handoff detection algorithm.

This chapter is structured as follows. In Section 4.2, we provide a description of a programmable handoff architecture that solves the end-system connectivity in wireless access networks. Following this, in Sections 4.3 and 4.4 we present the implementation and evaluation of our architecture, respectively, focusing on the deployment of the multi-handoff and reflective handoff services. In Section 4.5 we discuss the related work. Finally, in Section 4.6, we provide some concluding remarks.

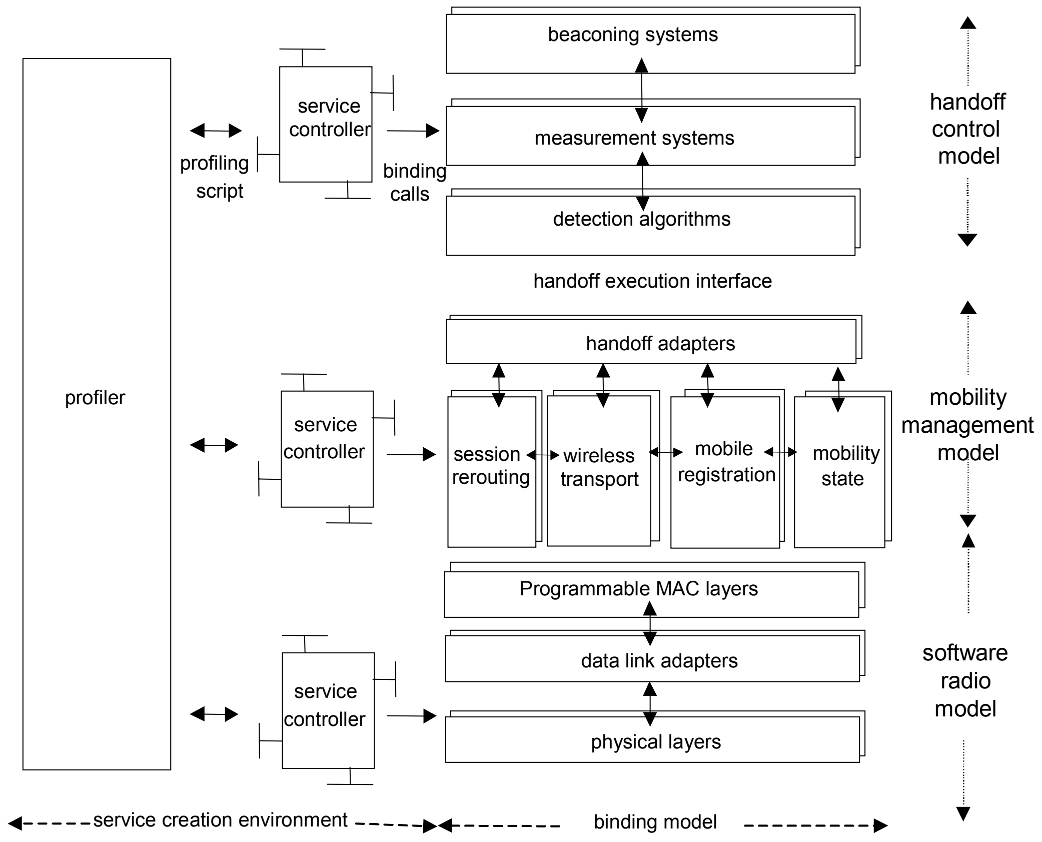


**Figure 22: Programmable Handoff Architecture**

## 4.2 Programmable Handoff Architecture

The aim of our work is to support connectivity between heterogeneous mobile devices and heterogeneous wireless access networks. To accomplish our goal we introduce a number of guidelines for designing access network and mobile terminal software. Our guidelines, discussed below, are included into a programmable handoff architecture that can be used for profiling, composing and deploying handoff services. Our programmable handoff architecture is illustrated in Figure 22. The architecture comprises a binding model and a service creation environment. The binding model describes how distributed objects can be combined to form programmable handoff services on-demand. The service creation environment, which constitutes part of the Genesis Kernel life cycle environment, allows network architects to design and dynamically deploy handoff services taking into account user, radio and environmental factors.

The binding model comprises a handoff control model, a mobility management model and a software radio model. Service controllers realize each model separately. To support connectivity between heterogeneous mobile devices and heterogeneous wireless access networks, the binding model supports the separation of handoff control from mobility management, the decomposition of the handoff control process and the programmability of the physical and data link layers. A handoff execution interface separates handoff control from mobility management. Handoff adapters integrate handoff control systems with mobility management services. In what follows, we describe each component of the binding model in detail.

### 4.2.1 Handoff Control Model

A handoff control model separates the algorithms that support beaconing, channel quality measurement and handoff detection, as illustrated in Figure 22. Typically, these functions are supported as a single 'monolithic' software structure in existing mobile systems. By separating the handoff detection from wireless channel quality measurements, we allow for new detection algorithms to be dynamically introduced in access networks and mobile devices. For example, detection algorithms specific to wireless overlay networks can be introduced into mobile devices allowing them to perform vertical handoffs [123], or detection algorithms specific to micro-cellular networks can be selected to compensate against the street-corner effect [138]. By separating the collection of wireless channel quality measurements from the beaconing system, mobile networks can support different styles of handoff control over the same wireless infrastructure. For example a WISP may want to offer a network-controlled handoff service (e.g., supported in AMPS [59] cellular systems) for simple mobile devices, a mobile-assisted handoff service (e.g., as in GSM [59]) for more sophisticated mobile devices involved in the process of measuring channel quality and a mobile-controlled handoff service (e.g., the handoff scheme considered by the Mobile IP Working Group) for more sophisticated laptop or palmtops mobile computers.

The handoff control model comprises the following services:

- detection algorithms, which determine the most suitable access points that a mobile device should be attached to. Wireless access points can be selected based on different factors including channel quality measurements, resource availability and user-specific policies [145]. A mobile device can be attached to one or more access points at any moment in time.

- measurement systems, which create and update handoff detection state. By handoff detection state we mean the data used by detection algorithms to make decisions about handoff. Detection algorithms and measurement systems use the same representation for handoff detection state.
- beaconing systems, which assist in the process of measuring wireless channel quality. Programmable beacons can be customized to support service-specific protocols like QOS-aware beaconing [32] or reflective handoff as discussed in Section 4.3.

### 4.2.2 Mobility Management Model

The mobility management model reflects the composition of services that execute handoff, as illustrated in Figure 22. We adopt a generalized architectural model that is capable of supporting the design space of different mobile networking technologies. To program mobility management systems one needs to be able to introduce new forwarding functions at mobile capable routers/switches (e.g., Cellular IP [30] or HAWAII [118] forwarding engines) as well as distributed controllers that manage mobility (e.g., Mobile IP foreign agents). The mobility management model discussed in this chapter is limited to supporting handoff services only. Other mobility management functionality (e.g., location, fault and account management) typically found in mobile networks will be considered as future work.

We identify the following services as part of the handoff execution process:

- session rerouting mechanisms, which control the datapath in access networks in order to forward data to/from mobile devices through new points of attachment. Rerouting services may include admission control and QOS adaptation for the management of wireless bandwidth resources.
- wireless transport objects, which interact with the physical and data link layers in mobile devices and access points to transfer active sessions between different wireless channels. A channel change may be realized through a new time slot, frequency band, code word or logical identifier. Transport objects can provide valued-added QOS support (e.g., TCP snooping [123]).
- mobile registration, which is associated with the state information a mobile device exchanges with an access network when changing points of attachment.

- mobility state, which can be expressed in terms of a mobile device's connectivity, addressing and routing information, bandwidth and name-space allocations and user preferences.

### 4.2.3 Software Radio Model

The software radio model defines the composition of physical and data link layer services, as illustrated in Figure 22. The software radio model supports functions such as the dynamic assignment of channel locations and widths, and the selection of modulation and coding techniques used on each channel. Software radios allow mobile devices to dynamically 'tune' to the appropriate air-interface of the serving access network, while roaming between heterogeneous wireless environments. MAC layer protocols can be made programmable [81] supporting services with different QOS requirements. Physical and data link layer modules can be implemented in various ways [20, 21, 81, 98, 99]. Data link 'adapters' separate data link layer modules from the lower physical layer components. For example, data link adapters allow programmable MAC protocols to operate on top of any type of channel coding or modulation scheme, as discussed in [81].

### 4.2.4 Handoff Execution Interface

A handoff execution interface, illustrated in Figure 22, separates handoff control from mobility management. Handoff control and mobility management systems are implemented as separate programmable architectures. By hiding the implementation details of mobility management algorithms from handoff control systems the handoff detection state (e.g., the best candidate access points for a mobile device) can be managed separately from handoff execution state (e.g., mobile registration information). This software approach can be used to enable inter-system handoff between different types of wireless access networks. The basic idea behind realizing inter-system handoff is that the same detection mechanisms operating in mobile devices and access networks can interface with multiple types of mobility management architectures that operate in heterogeneous access networks (e.g., Mobile IP [11], Cellular IP [30], Mobiware [32] and HAWAII [118] access networks). Handoff control systems issue a number of generic service requests through the handoff execution interface, which mobility management systems execute according to their own programmable implementation. For example, a generic 'pre-bind' method call to a candidate access point would be executed by establishing a signaling channel in a Mobiware architecture [32], or by joining a multicast group specific to a mobile device and buffering packets in a Daedalus/BARWAN [123] architecture. In one extreme case where the location of the handoff control system is at the mobile device, different mobility management protocols can be dynamically loaded into mobile devices allowing them to roam between heterogeneous access networks in a seamless manner.

Examples of handoff execution methods include:

- handoff methods, which map down to mobility management services that execute handoff (e.g., register the care-of address of a mobile device with its home agent).
- pre-bind methods, which initiate 'priming actions' at candidate access points associated with a mobile device (e.g., load an active filter for transport adaptation [32], start buffering packets, etc.).
- configure methods, which install new signaling systems or delete existing ones (e.g., replace the mobile device's Mobiware control plane with a Cellular IP one during reflective handoff).

We observe that it is difficult to consider a single handoff execution interface that is capable of encompassing all existing and future wireless systems. Rather, it is more likely that a handoff execution interface will support a particular 'family' of wireless technologies. In this case, network architects can select execution methods that satisfy the set of handoff control and mobility management systems which they wish to program.

### 4.2.5 Handoff Adapters

Programmable access networks allow different styles of handoff control (e.g., mobile controlled, mobile assisted and network controlled handoff) to seamlessly share the same mobility management services offered. However, seamless integration of handoff control systems with mobility management services is difficult to realize. In our framework, we introduce the concept of 'handoff adapters' which represent an intermediate layer of distributed objects that can be used for integrating the handoff control model with the mobility management model. Handoff adapters represent a set of distributed objects that serve as the 'glue' between handoff control systems and mobility management services. Handoff adapters and mobility management services collectively implement handoff execution algorithms. Handoff adapters can be centralized (i.e., running in a single host or network node) or distributed. In the distributed case, handoff adapters are deployed at mobile devices, access points or mobile capable routers/switches.

Handoff adapters are an important part of our programmable handoff architecture. First, handoff adapters control the handoff execution process. Handoff adapters invoke mobility management services in an order that is specific to the handoff style being programmed. Mobility management services (i.e., session rerouting, wireless transport, mobile registration and mobility state management services) are invoked as part of the handoff execution process. For example, in a forward mobile controlled handoff, an adapter would invoke a radio link transfer service before session rerouting. In the backward, mobile assisted handoff the order of this execution would be reversed. Each handoff style uses a separate adapter. To invoke mobility management services, adapters distribute method invocations to the network nodes or hosts where mobility management services are offered. Handoff is usually detected at a single host or network node (e.g., a mobile device, access point, or mobile capable router/switch). In contrast, mobility management services can be offered at multiple hosts or network nodes inside a wireless access network (e.g., at wireless access points or by mobility agents in the network).

Second, handoff adapters 'translate' the handoff execution interface to the interfaces supported by specific mobility management architectures. In this manner, adapters hide the heterogeneity of mobility management architectures enabling inter-system handoffs. Adapters may interact with distributed mobility agents, databases supporting mobile registration information, or open network nodes to realize handoff in many different ways. For example, a 'Mobile IP' adapter would interact with a Mobile IP scheme to support connectivity for mobile devices (e.g., acquiring a care-of address through DHCP and registering the care-of address with a home agent). A Cellular IP or HAWAI 'adapter' would transmit control messages for establishing mobile-host specific routing entries in the access network. The role of adapters is further discussed in the Section 4, where we describe distributed algorithms for programmable handoff.

## 4.3 Design and Implementation

We have designed and implemented two new handoff services based on the programmable handoff architecture. A multi-handoff access network service simultaneously supports three styles of handoff control over the same physical wireless access network that are commonly found in mobile networks:

Network Controlled HandOff (NCHO), Mobile Assisted HandOff (MAHO) and Mobile Controlled Handoff (MCHO). In addition, a reflective handoff service allows mobile devices to reprogram their protocol stacks in order to seamlessly roam across heterogeneous wireless environments. Wireless access networks dynamically load signaling system support into mobile devices. We call this service 'reflective' in this context because mobile devices can identify the mobility management architectures and radios supported by neighboring wireless access networks, and customize their signaling systems and wireless links in order to interact with disparate access networks.

In order to support the dynamic introduction of handoff control and mobility management services, we have implemented a service creation environment as part of the Genesis Kernel spawning service that explicit supports transportable code by dynamically selecting, deploying and binding distributed objects. The service creation environment is implemented on top of the programming environment of the Genesis Kernel. The service creation process allows the network architect to create new objects using inheritance of abstract classes (e.g., an abstract handoff detection algorithm class). Service controllers activate objects invoking binding calls on object control interfaces for the deployment of services. Programmable handoff services are composed using profiling scripts. Network architects can customize objects during the profiling process. In this case parameters characterizing the operation of a service (e.g., user, service specific or environmental parameters) can be passed in objects at run-time through the profiler and service controllers.

In what follows, we present the design and implementation of the multi-handoff and reflective handoff services, and discuss the profiling process.

### 4.3.1 Multi-handoff Access Network Service

**Objects**

A multi-handoff access network service is composed from a set of distributed objects. We have deployed a multi-handoff access network service over our experimental testbed based on WaveLAN radios. Figure 23 shows the implementation of the handoff control model for the multi-handoff access network service. As discussed earlier, the handoff control model separates the algorithms that support beaconing, channel quality measurement and handoff detection. Objects shown in Figure 23 are grouped into beaconing systems, measurement systems, and detection algorithms, which are the components of the handoff control model. Figure 2 also shows object interactions and their invocation order (e.g., NCHO-1, NCHO-2, etc). The handoff control objects comprise:

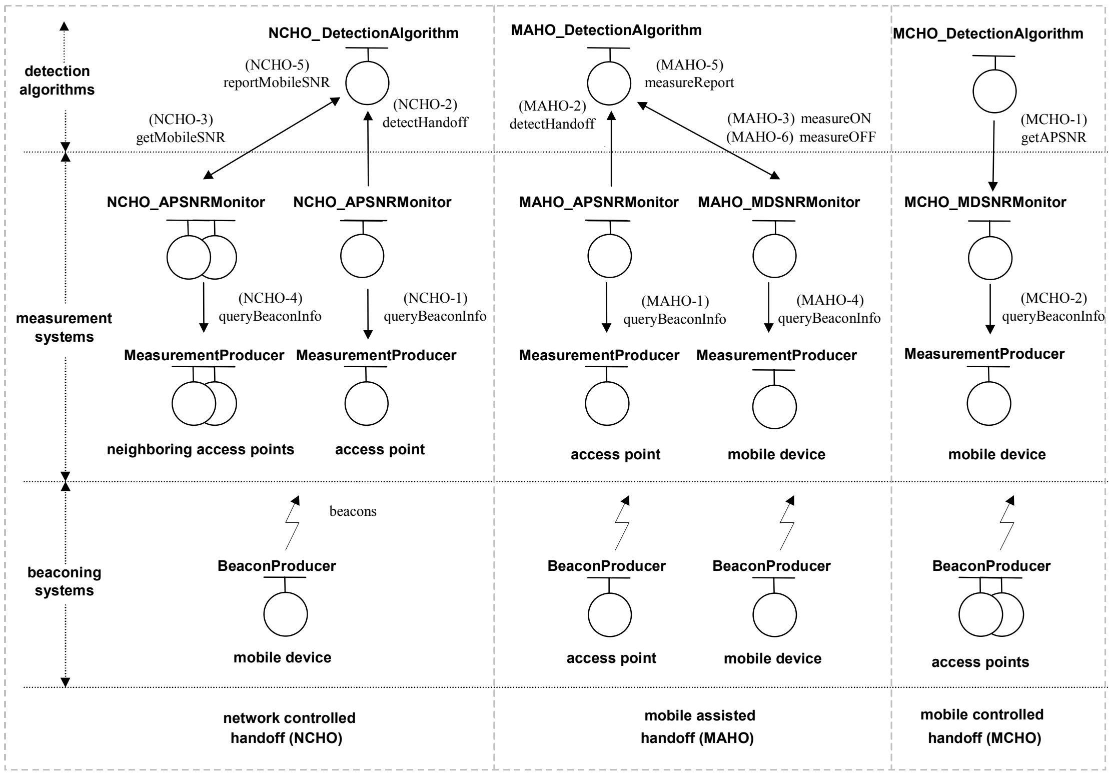


**Figure 23: Implementation of the Handoff Control Model**

- beacon producer (BeaconProducer) and measurement producer (MeasurementProducer) objects, which invoke low-level wireless LAN utility functions. Beacon producer objects transmit beacons at specified frequencies. Measurement producer objects generate 'raw' channel quality measurements. Measurement and beacon producer objects can simultaneously participate in multiple styles on handoff control.
- signal strength monitor (*_APSNRMonitor, *_MDSNRMonitor) objects, which collect and average wireless signal strength measurements. SNR represents only one of the many measurements that can be used for handoff decision-making.
- detection algorithm (*_DetectionAlgorithm) objects, which make decisions for handoff based on signal strength measurements. Each handoff style employs its own set of signal strength monitors and

detection algorithms in order to determine the best access points that mobile devices should be attached to.

Our implementation of the mobility management model is based on an extended Mobiware [32] architecture. Mobiware is programmable, promoting the separation between signaling, transport and state management. Mobility management services include session rerouting, mobility state management and wireless transport configuration, as discussed in Section 4.2. All sessions that operate between a mobile device and an associated Internet gateway are abstracted as a single state entity called a 'flow bundle' in a Mobiware wireless access network. Flow bundles are used during handoff to switch IP flows in Mobiware access networks and provide general purpose encapsulating and routing services similar to ATM virtual paths or IP tunnels. Open programmable switches allow the establishment, removal, rerouting and adaptation of flow bundles. Thus, the access network behaves as a pool of resources allowing different handoff styles to operate in parallel. Mobiware comprises the following mobility management objects:

- mobility agent (MobilityAgent) objects, which reroute sessions to/from mobile devices when mobile devices change their points of attachment. Mobility management is a fully distributed algorithm that includes one or more mobility agents for scalability. Using flow bundles a mobility agent object only has to discover a single crossover switch and reroute all sessions to/from the new access point of a mobile device. All handoff styles can simultaneously use the same mobility agent objects.
- mobile registration database (MobileRegistrationDB) objects, which cache flow-bundle state in wireless access points. Unique flow bundle identifiers characterize mobile devices and their associated state. Flow bundle state is expressed in terms of namespace (i.e., VCI/VPI) allocations, QOS adaptation profiles and active transport preferences.
- datapath (AP_Datapath) objects, which configure wireless transport mechanisms in wireless access points and mobile devices. For example, datapath objects support the dynamic introduction of value-added QOS algorithms (e.g., media scaling and adaptive FEC) in wireless access points to compensate against time-varying impairments.

The handoff execution interface comprises a generic 'pre-bind' method that initiates 'priming' actions at wireless access points and a generic 'handoff' method that executes handoff. The arguments passed into the handoff method include the access point and mobile device identifiers that participate in the handoff operation. Access points are identified by name, IP address and WaveLAN NWIDs where NWIDs are logical channel identifiers. Mobile devices are only identified by name and IP address. No NWID is required because the mobile device's NWID changes during handoff. The name and IP address of a wireless interface (i.e., an access point or mobile device) is included in WaveLAN beacons. In this manner, mobile devices can identify access points in a wireless access network and access points can detect the presence of mobile devices in coverage areas.

Currently, our handoff execution interface is simple and supports programmable handoff services using WaveLAN-based radios. Handoff adapters have been implemented for the network controlled, mobile assisted and mobile controlled handoff styles. Handoff adapters comprise adapter objects deployed in mobile devices, access points and in the network. The order in which mobility management services are invoked is dependent on the specific handoff style executing.

## Distributed Algorithms

Using the distributed objects described above we implemented a set of algorithms to support alternative styles of handoff control in a multi-handoff access network. Figures 24-26 illustrate a set of distributed algorithms that realize network controlled, mobile assisted, and mobile controlled handoff styles, respectively. Each algorithm comprises handoff detection and handoff execution processes. The object interaction and invocation order for each handoff algorithm is shown in Figure 23. Handoff control objects implement the handoff detection process, and mobility management objects and handoff adapters implement the handoff execution process. Handoff styles differ in the location where handoff control takes place and is executed.

### *Network Controlled Handoff*

In the network controlled handoff scheme a signal strength monitor object, running in a wireless access point, continuously measures the signal strength of a mobile device, as indicated by NCHO-1 in the handoff detection algorithm shown in Figure 24(a). The signal strength monitor initiates handoff (NCHO-2) when the signal strength drops below a certain threshold. A detection algorithm, running in the wireless access network, queries and compares mobile-device measured signal strength from all neighboring access points (NCHO-3 to NCHO-5). Wireless SNR values determine the best candidate access point for the mobile device. Neighboring monitors report average the SNR measured over two consecutive beacon query intervals. Beacon query intervals are user defined, typically being 300 msec in duration. Handoff execution is initiated when a candidate access point is detected with a better service quality than the current point of attachment. The handoff detection process is repeated once handoff execution is complete. Network controlled handoff moves most of the complexity for controlling handoff from the mobile device to the network. This style of handoff simplifies the mobile device software design.

```
HANDOFF-DETECTION  //Network-Controlled HandOff (NCHO)
( access_point AP, mobile_device mobile, int thres ) {

   let am be a NCHO_APSNRMonitor object located at AP;
   let mp  be a  MeasurementProducer object located at AP;
   let da be a NCHO_DetectionAlgorithm object located
   inside the access network;

   while (true) {
NCHO-1:  am invokes mp.queryBeaconInfo (mobile);
      am calculates the average signal strength SNR-avg for mobile;
      if (SNR-avg <= thres) {
NCHO-2:  am invokes da.detectHandoff (AP, mobile);
         break; }
      else {
         sleep (am.queryInterval); }}
      for (each neighboring access point AP-neigh) {
         let am-neigh be a NCHO_APSNRMonitor object located at AP-neigh;
NCHO-3:  da invokes am-neigh.getMobileSNR();  }

NCHO-4,5: obtained SNR values are compared
   da calculates the best candidate access point AP-best for mobile;
   HANDOFF-EXECUTION (AP-best, AP, mobile);
   HANDOFF-DETECTION (AP-best, mobile, thres); }
```

**(a) handoff detection**

```
HANDOFF-EXECUTION
( access_point AP-new, access_point AP-old, mobile_device mobile) {

   let ha be a NCHO_Adapter object located inside the access network;
   let ha-old be a NCHO_Adapter object located at AP-old;
   let ha-new be a NCHO_Adapter object located at AP-new;
   let ma be a MobilityAgent object located inside the access network;
   let rdb-old be a MobileRegistrationDB object located at AP-old;
   let rdb-new be a MobileRegistrationDB object located at AP-new;
   let dp-new be a AP_Datapath object at AP-new;

   //query for mobility state
NCHO-6:  ha invokes rdb-old.getFlowBundleState (mobile);
   let state-old be the flow-bundle state returned by rdb-old;

   //session re-routing
NCHO-7:  ha invokes ma.handoffFlowBundle (state-old, mobile);
   ma calculates the cross-over switch for mobile;
   ma reroutes the flow-bundle associated with mobile;
   let state-new be the flow-bundle state returned by ma;

   //wireless transport configuration
NCHO-8:  ha invokes dp-new.setUpDatapath (state-new, mobile);
NCHO-9:  ha-old invokes mobile.handoffNotice (AP-new);
NCHO-10:  mobile invokes this.radioLinkTransfer (AP-new);

   //mobile registration
NCHO-11:  ha invokes rdb-new.mobileRegistration (state-new, mobile);
NCHO-12:  ha-new binds to mobile; }
```

**(b) handoff execution**

**Figure 24: Network Controlled Handoff**

The wireless access network initiates the handoff execution process. The detection algorithm interacts with a handoff adapter object to realize handoff. The handoff adapter invokes a getFlowBundleState() method on a mobile registration database located at the mobile device's old access point to obtain the flow bundle state information, as indicated by NCHO-6 in Figure 24(b). As discussed previously, the flow bundle state represents the aggregate state information for all flows/sessions to and from a mobile device.

This state information is used to speed handoff in a Mobiware access network [32]. The mobility agent object interacts with a router object and open programmable switch servers to calculate the cross-over switch on behalf of a mobile device and reroute the mobile device's active sessions (NCHO-7) represented by the flow bundle state. Following this the handoff adapter object invokes a setUpDatapath() method to create a channel in the new access point accommodating active transport 'plug-ins' for value-added QOS support (NCHO-8). Once this is complete, the wireless radio link transfer takes place at the mobile device (NCHO-9, NCHO-10). Channel change is realized as a change in WaveLAN NWID. Finally, the handoff adapter registers the mobile device with the new access point (NCHO-11).

```
HANDOFF-DETECTION  //Mobile-Assisted HandOff (MAHO)
( access_point AP, mobile_device mobile, int thres ) {

   let am be a MAHO_APSNRMonitor object located at AP;
   let mp be a MeasurementProducer  object located at AP;
   let da be a MAHO_DetectionAlgorithm object located at AP;
   let mm be a MAHO_MDSNRMonitor object located at mobile;
   let mp-mobile be a MeasurementProducer  object located at mobile;

   while (true) {
MAHO-1:  am invokes mp.queryBeaconInfo(mobile);
      am calculates the average signal strength SNR-avg for mobile;
      if (SNR-avg <= thres) {
MAHO-2:  am invokes da.detectHandoff  (AP, mobile);
         break; }
      else {
         sleep (am.queryInterval); }}
MAHO3:  da invokes mm.measureON();
   while (true) {
      //mobile measures signal strength from all neighboring access points
MAHO-4:  mm invokes mp-mobile.queryBeaconInfo();
MAHO-5:  mm invokes da.measureReport();
      da calculates the best candidate access point AP-best for mobile;
      if (AP-best != AP) {
MAHO-6:  da invokes mm.measureOFF();
         break; }
      sleep (mm.queryInterval); }

 //handoff execution and binding
   HANDOFF-EXECUTION (AP-best, AP, mobile);
   let da-best be a MAHO_DetectionAlgorithm object
   located at AP-best;
MAHO-7:  da-best binds to mm;
MAHO-8:  mm binds to da-best;
   HANDOFF-DETECTION (AP-best, mobile, thres); }
```

**(a) handoff detection**

```
HANDOFF-EXECUTION
( access_point AP-new, access_point AP-old, mobile_device mobile) {

   let ha-old be a MAHO_Adapter object located at AP-old;
   let ha-new be a MAHO_Adapter object located at AP-new;
   let ma be a MobilityAgent object located inside the access network;
   let rdb-old be a MobileRegistrationDB object located at AP-old;
   let rdb-new be a MobileRegistrationDB object located at AP-new;
   let dp-new be a AP_Datapath object at AP-new;

   //query for mobility state
MAHO-9:  ha-old invokes rdb-old.getFlowBundleState  (mobile);
   let state-old be the flow-bundle state returned by rdb-old;

   //session re-routing
MAHO-10:  ha-old invokes ma.handoffFlowBundle (state-old, mobile);
   ma calculates the cross-over switch for mobile;
   ma reroutes the flow-bundle associated with mobile;
   let state-new be the flow-bundle state returned by ma;

   //wireless transport configuration
MAHO-11:  ha-old invokes dp-new.setUpDatapath  (state-new, mobile);
MAHO-12:  ha-old invokes mobile.handoffNotice (AP-new);
MAHO-13:  mobile invokes this.radioLinkTransfer (AP-new);

   //mobile registration
MAHO-14:  ha-old  invokes rdb-new.mobileRegistration  (state-new, mobile);
MAHO-15:  ha-new binds to mobile; }
```

**(b) handoff execution**

**Figure 25: Mobile Assisted Handoff**

## *Mobile Assisted Handoff*

The mobile assisted handoff scheme moves some of the functional support, and therefore complexity, to the mobile device. In this scheme, an access point continuously measures the signal strength of a mobile device, as indicated by MAHO-1 in Figure 25(a). If the signal level drops below a certain threshold, a detection algorithm running in the mobile device's serving access point invokes a MeasureON() method to initiate signal strength measurements at the mobile device (MAHO-3). Measurements are reported via a MeasureReport() invocation (MAHO-5). The detection algorithm does not need to collect any additional measurements in support of handoff. Rather, the handoff decision is based on data collected by the mobile device. Our implementation of mobile assisted handoff is based on the GSM mobile assisted handoff scheme [59]. In the mobile assisted handoff scheme, handoff execution is driven by an adapter object running at the mobile device's old access point and not in a server operating in the wireless access network. Handoff execution includes flow-bundle rerouting, wireless transport management and mobile registration, as in the case of network controlled handoff, as indicated by MAHO-9 to MAHO-14 in Figure 25(b). In the case of mobile assisted handoff the handoff execution algorithm is distributed resulting in shorter handoff completion times. When handoff execution is complete, the handoff adapter and detection algorithm objects operating in the new access point bind to the mobile device (MAHO-7, MAHO-15).

### *Mobile Controlled Handoff*

In the mobile controlled handoff scheme the signal strength measurements are taken by the mobile device, as indicated by MCHO-1 in Figure 26(a). If a candidate access point having better signal strength is detected then the handoff execution process is initiated. The mobile controlled handoff scheme moves most of the complexity for detecting handoff to the mobile device alleviating the network from centralized control of the handoff process. In this respect mobile controlled handoff is scalable and more distributed than the other schemes. However, it assumes that the mobile device (e.g., a wireless laptop device) can continuously monitor signal strength measurements.

A handoff adapter object located at the mobile device drives handoff execution. Mobile controlled handoff is executed as a forward, soft handoff. To accomplish soft handoff, our radios based on WaveLAN operate in promiscuous mode so that the mobile device simultaneously receives data from multiple access points. First, wireless radio link transfer takes place (MCHO-6) as a change in WaveLAN NWID. Following this a mobility agent object is invoked via adapter objects located at the mobile device and new access point. The mobility agent object reroutes the flow bundle, which is specific to a mobile device (MCHO-8). Finally, mobile registration (MCHO-9) and wireless transport configuration steps take place

(MCHO-10). Forward handoff requires the creation of a binding between an adapter object running in the mobile device and an adapter object operating at the new access point. These adapter objects are used to contact the mobility agent operating in the wireless access network. To eliminate the latency introduced by object bindings we setup and cache bindings prior to handoff execution (MCHO-3 to MCHO-5). An object binding is established at the best candidate access point when the signal strength in the main communication path drops below a certain threshold. The ‘pre-bind’ method call supported by the handoff execution interface is used for this purpose. Different combinations of handoff adapters result in different styles of programmable handoff (e.g., backward, hard handoff, etc).

```
HANDOFF-DETECTION  //Mobile-Controlled HandOff (MCHO)
( access_point AP, mobile_device mobile, int thres) {

   let mm be a MCHO_MDSNRMonitor object located at mobile;
   let mp be a MeasurementProducer object located at mobile;
   let da be a MCHO_DetectionAlgorithm object located at mobile;
   let ha be a MCHO_Adapter object located at mobile;

   while (true) {
      //mobile measures signal strength from all neighboring access points
MCHO-1: da invokes mm.getAPSNR();
MCHO-2: mm invokes mp.queryBeaconInfo();
      da calculates the best candidate access point AP-best for mobile;
      if (AP-best != AP) {
         HANDOFF-EXECUTION (AP-best, AP, mobile);
         HANDOFF-DETECTION (AP-best, mobile, thres);
         return; }
      let SNR-avg be the average signal strength from AP
      if ((AP-best == AP) && (SNR-avg <= thres)) {
         //pre-bind
         da calculates the second best access point AP-sec for mobile;
         let ha-sec be a MCHO_Adapter object located at AP-sec;
MCHO-3: mobile invokes this.radioLinkTransfer (AP-sec);
MCHO-4:  ha binds to ha-sec;
MCHO-5:  mobile invokes this.radioLinkTransfer (AP); }
         sleep (da.queryInterval); }}
```

**(a) handoff detection**

```
HANDOFF-EXECUTION
( access_point AP-new, access_point AP-old, mobile_device mobile) {

   let ha be a MCHO_Adapter object located at mobile;
   let ha-new be a MCHO_Adapter object located at AP-new;
   let ma be a MobilityAgent object located inside the access network;
   let rdb-new be a MobileRegistrationDB object located at AP-new;
   let dp-new be a AP_Datapath object at AP-new;
MCHO-6: mobile invokes this.radioLinkTransfer (AP-new);

   //flow-bundle state is stored at mobile devices as well as access points
   let state-old be the flow-bundle state stored at mobile;
MCHO-7 ha invokes ha-new.handoffExecution  (AP-old, state-old, mobile);
MCHO-8: ha-new invokes ma.handoffFlowBundle (state-old, mobile);
   ma calculates the cross-over switch for mobile;
   ma reroutes the flow-bundle associated with mobile;
   let state-new be the flow-bundle state returned by ma;

   //mobile registration
MCHO-9: ha-new invokes rdb-new.mobileRegistration  (state-new, mobile);

   //wireless transport configuration
MCHO-10: ha-new invokes dp-new.setUpDatapath  (state-new, mobile);
MCHO-11: ha-new invokes mobile.handoffNotice (AP-new);  }
```

**(b) handoff execution**

**Figure 26: Mobile Controlled Handoff**

Our system implementation has been optimized to reduce the binding and Remote Procedure Call (RPC) overhead (i.e., latency) associated with open signaling. Oneway CORBA calls (as part of the CORBA-based metabus) have been used to increase the level of parallelism for the interaction of programmable handoff objects. A mobility agent has been designed to setup wireline connections in the wireline access network by sending parallel invocations to switch servers [90]. This results in a speed up of the re-routing phase of the handoff algorithm over conventional hop-by-hop signaling Mobile registration databases have been implemented as hash tables to allow information retrieval in $\Theta(1)$ time. Flow bundle identifiers have been used as hash keys. Mobility agents have been implemented in ‘multi-threaded’ as well as ‘sequential’ modes. For the experiments reported in this chapter, mobility agents operate in a sequential mode (e.g., reroute flow bundles one-by-one). Results from our experiments are discussed in Section 4.4.

### 4.3.2 Reflective Handoff Service

We have implemented reflective handoff as a mobile controlled handoff scheme. Access points transmit beacons that additionally carry globally unique identifiers designating specific access networks. A reflective detection algorithm uses access network identifiers to determine whether a mobile device is likely to move to the coverage area of a new access network. Each mobile device maintains a local cache of signaling system modules. Signaling system modules are collections of objects supporting mobility management services in mobile devices. Before a mobile device performs a handoff to a new access network, it checks whether a signaling module associated with the new candidate access network is cached. If a signaling module is not cached it is dynamically loaded. Access points support module loaders deployed during the service creation process. A signaling system is loaded from the old access network. A two-way handshake mechanism is used for loading signaling modules, which are loaded before reflective handoff is executed. Access networks schedule the transmission of signaling modules over the air interface, to avoid flooding the wireless network.

Reflective handoff is managed by handoff adapters, which activate or deactivate signaling modules on-demand. The handoff execution interface for the reflective handoff service includes a ‘configure’ method, which is used for binding new signaling systems. Parameters associated with access network state (e.g., the address of the gateway to the Mobile IP Internet) are passed into signaling modules upon activation. Module loaders transmit access network state when loading signaling system support into mobile devices. Reflective handoff may involve the loading of entire mobility management protocol stacks or configuration scripts, which customize objects already cached at mobile devices.

Two distinct types of access networks support reflective handoff in our testbed as shown in Figure 27: Mobiware and Cellular IP access networks. Cellular IP [30] delivers fast local handoff control in datagram oriented access networks. In addition, Cellular IP supports per-mobile host state, paging, routing and handoff control in a set of access networks that are interconnected to the Internet through gateways. In Cellular IP, packets sent from mobile hosts create routing caches pointing to the downlink path so that packets destined to a mobile device can be routed using the route cache. Mobiware and Cellular IP signaling modules use the IP protocol to communicate with access networks. Mobiware and Cellular IP access networks support the same wireless data link and physical layers (WaveLAN) in our testbed but use different mobility management systems. Future work will include extending reflective handoff to allow access networks to load software radio-based physical and data link layer support into mobile devices.

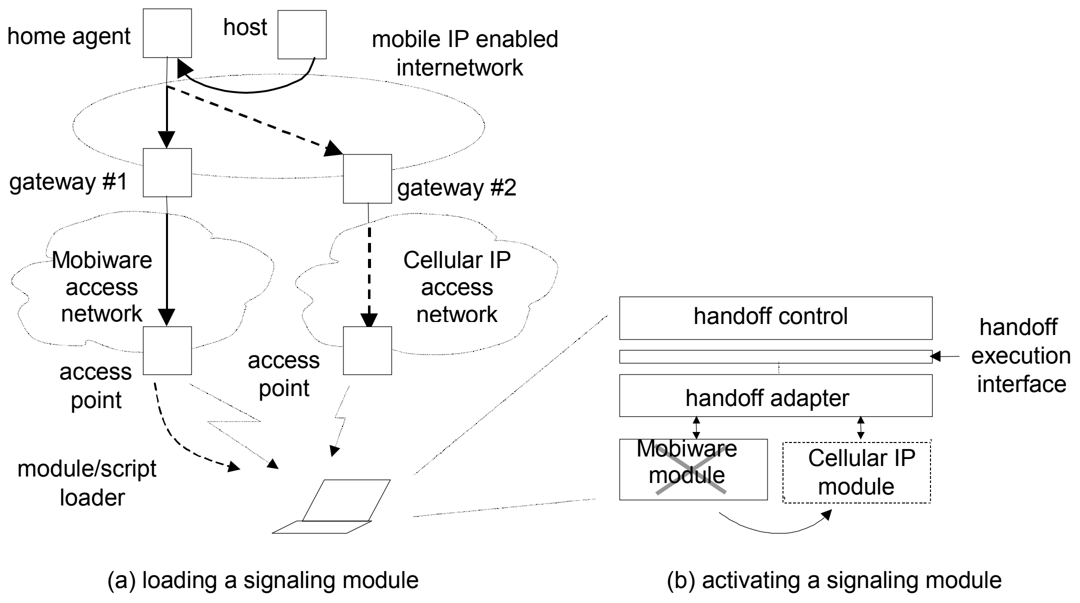


**Figure 27: Reflective Handoff Service**

In our testbed, a Mobile IP enabled internetwork connects the Mobiware and Cellular IP wireless access networks via gateways. Mobile IP is used for managing macro-level mobility between gateways, whereas the Cellular IP and Mobiware wireless access support fast local handoff control. Hierarchical mobility management in IP-based mobile networks has been widely reported in the literature [30, 118]. A mobile device attached to an access network uses the IP address of the gateway as its care-of address. Access networks provide mechanisms for initiating Mobile IP-based inter-gateway handoffs and for establishing datapaths between gateways and access points where mobile devices are attached.

Signaling modules implement generic handoff execution functions as dynamic link libraries using the Windows NT operating system. Three types of handoff are supported: (i) ‘internal’ handoffs, which take place between access points of the same access network; (ii) ‘entry’ handoffs, which take place when a mobile device is attached to a new access network; and (iii) ‘exit’ handoffs, which take place when a mobile device leaves an access network. The handoff execution interface supports method calls for internal, entry and exit handoffs.

Reflective handoff requires that all the signaling modules associated with the handoff process are loaded and activated. Reflective handoff has been implemented as the process of invoking an ‘exit’ handoff on the signaling system of the old access network and an ‘entry’ handoff on the signaling system of the new wireless access network. Access networks realize execution calls in different ways and support mechanisms for registering the care-of address of mobile devices (i.e., gateway IP address) with their corresponding home agents. Care-of address registration has been realized as part of ‘entry handoff’ or ‘exit handoff’. When an entry handoff takes place, access networks check whether the care-of address of a mobile device has been registered with its home agent. If the care-of address is not registered then the access network initiates registration. Registration support during ‘exit’ handoff is optional.

We have programmed our handoff control system to load signaling modules as soon as the mobile device detects that it is inside the coverage area of an access network where the signaling system is not cached. This loading algorithm minimizes the probability of handoff failure due to absence of a signaling module at the mobile device. A soft state mechanism used for managing stored signaling modules is used to avoid having large caches. A timer associated with a module is refreshed while a mobile device remains inside the coverage area of an access network associated with a particular module or set of modules. Mobile devices can permanently cache signaling modules associated with access networks, however.

## 4.4 Evaluation

To evaluate the programmable handoff architecture we use a mixture of testbed implementation and emulation in order to study proof of concept and scalability issues associated with our design. We extend the spawning testbed developed at Columbia University to support the Mobiware [32] and Cellular IP [30] architectures over separate physical topologies in order to test the reflective handoff service. The Mobiware testbed is used to implement and evaluate the multi-handoff access service while the reflective handoff service is evaluated using a combination of Cellular IP and Mobiware wireless access networks. In what follows, we discuss the results from the implementation and evaluation.

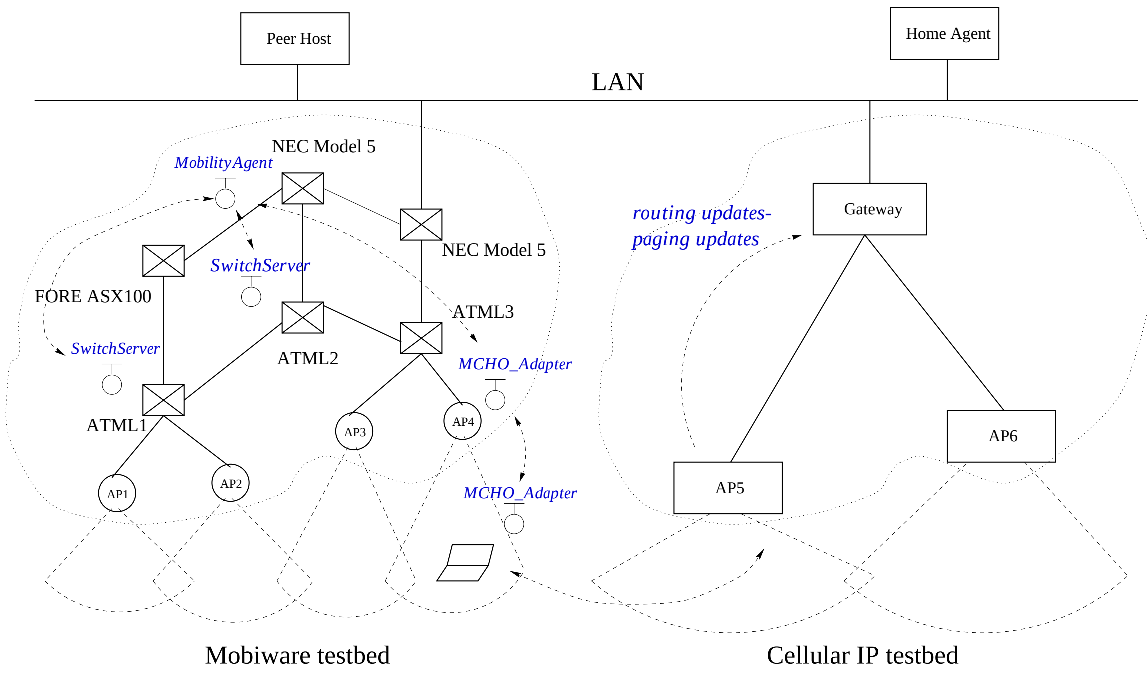


**Figure 28: Experimental Environment**

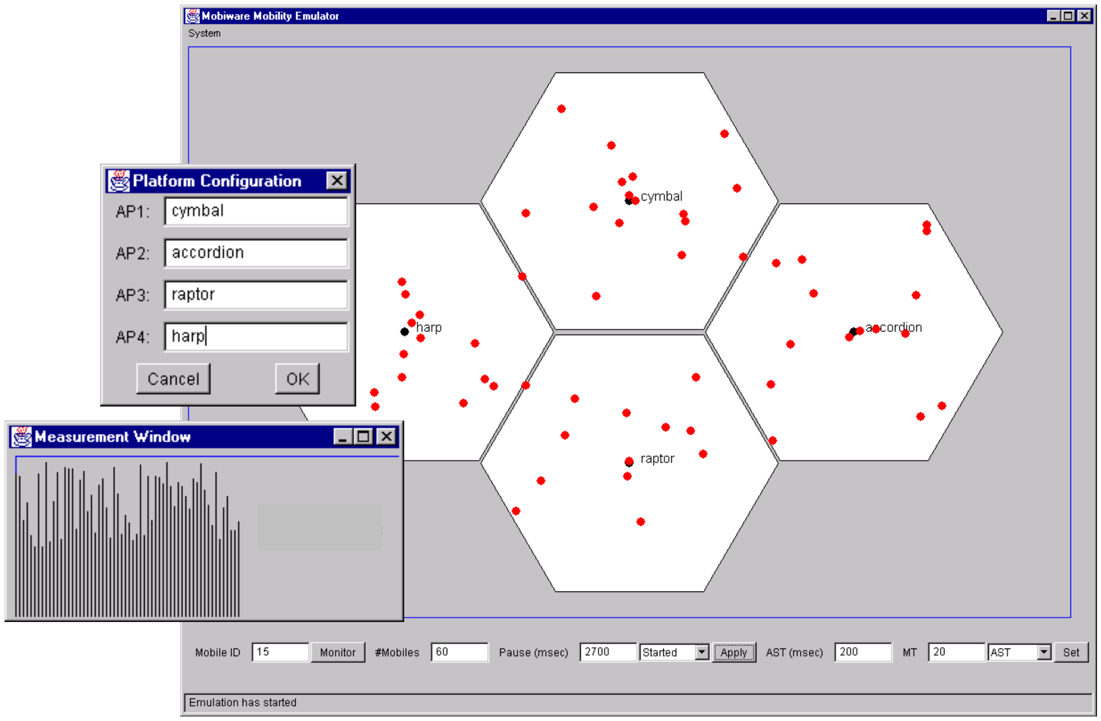
### 4.4.1 Experimental Platform

Our experimental environment is illustrated in Figure 28. The Mobiware testbed provides wireless access to the Internet and comprises four ATM switches (viz. ATML Virata, Fore ASX/100, and NEC model 5 switches) and four wireless access points. To provide a larger network testbed for programmable handoff evaluation, switches allow multiple virtual network elements to be operational within the same physical nodes. Three virtual ATM switches (ATML 1, 2, and 3 shown in Figure 8) in our network are ‘switchlets’ [141] physically co-located at the same physical switch. Each switchlet corresponds to a different CORBA server with a different name space and manages its own resources independently from other co-located switchlets. Access points are multi-homed 300 MHz Pentium PCs that provide radio access to a wireline IP switched access network. High performance notebooks provide support for mobile applications and mobile access to network services. Wireline ATM links operate at OC-3 rates between the switches and at 25 Mbps between the switches and access points. Our testbed radios are based on WaveLAN operating in the 2.4-2.8 GHz ISM band. We use 2 Mbps WaveLAN cards for which we have a low-level radio utility API for programming beacons, getting signal-strength measurements and controlling some radio aspects. The Cellular IP access network consists of three base stations based on multi-homed 300 MHz Pentium PCs. One of the base stations serves as a gateway router to the Internet. Interconnects between Cellular IP base stations are 100 Mbps full duplex links. Mobiware and Cellular IP access networks are connected to a 100 Mbps Ethernet LAN supporting Mobile IP.

Access points and switches of the Mobiware testbed support our service creation environment, programmable handoff control and mobility management systems. For the results provided in this chapter the mobile devices and access points use a version of metabus based on IONA's Orbix v2.0 CORBA running Windows NT and UNIX operating systems. Cellular IP base stations, gateways and mobiles use the Cellular IP protocol [30].

We augmented the experimental testbed platform discussed above with an emulation mode to evaluate the scalability of the programmable handoff architecture to support large numbers of mobile devices. Our aim is to analyze how well our middleware architecture performs in meeting the requirements imposed by different types of mobile environments. An emulation platform has been built consisting of a mobility emulator and the programmable handoff architecture operating in the access points and mobile capable switches of the extended Mobiware testbed.

The mobility emulator, shown in Figure 29, emulates the random movement of mobile devices and supports different levels of mobility and signaling load. The only major modification made to the access network source code used for the experimental evaluation relates to the measurement producer objects, which measure channel quality in wireless access points. Measurement producer objects have been modified to suppress the generation of real channel quality measurements. Rather producer objects receive data from a mobility emulator. Emulated mobile devices have been programmed to remain inside the same cell for an exponentially distributed interval and to move with the same probability to any neighboring cell. The propagation model used by the mobility emulator is based on the path loss component of signal fading only.



AP1: cymbal
AP2: accordion
AP3: raptor
AP4: harp

Mobile ID 15 #Mobiles 60 Pause (msec) 2700 Started AST (msec) 200 MT 20 AST

Emulation has started

**Figure 29: Mobility Emulator**

### 4.4.2 Multi-handoff Access Service Analysis

A number of experiments provide insight into the performance of the multi-handoff access network service. An important objective of the evaluation of service is to measure the latency associated with various handoff styles. For these experiments, we streamed a single video flow (i.e., true_lies.mpg at 350 kbps) from a fixed network server to three mobile devices and performed successive handoffs with these mobile devices between access points AP2 and AP3, as illustrated in Figure 28. The system is lightly loaded and the cross over switch located at the ATML2 switch. The network is programmed to simultaneously support mobile controlled, mobile assisted and network controlled handoff styles with each mobile device being programmed to support a different style of handoff under consideration.

Table 2 summarizes our handoff latency measurement results for each one of the three schemes. Twenty measurements for each handoff style were recorded. The average handoff latency measured for the mobile-controlled handoff is 41 msec. This measurement comprised 22 msec for wireline connection setup and 19 msec for wireless connection setup between the access point and mobile device. Mobiware active filters were not used in the experiments. For an evaluation of the Mobiware active filtering system see [32]. Mobile controlled handoff is associated with the least amount of latency. In this scheme a mobile device periodically takes signal strength measurements and initiates handoff. The average handoff latency measured for the mobile assisted handoff scheme is 750 msec. The greatest portion of the latency (709 msec) is absorbed by the measuring process, which records neighboring access point signal strength from the perspective of the mobile device. The 'hunt' period over which measurements are collected at the mobile device is set to 300 msec, whereas beacons are transmitted by access points and mobile devices every 100 msec. Our detection algorithm initiated handoff if a candidate access point with better SNR was present over two successive hunt periods. The average network controlled handoff latency measured is 683 msec. The measurement collection component of handoff latency is 641 msec. These results indicate that the performance of a multi-service access network is satisfactory when the network is lightly loaded. Our system performs well because binding latencies are eliminated. We experienced higher latencies when bindings between objects were not setup or cached prior to handoff.

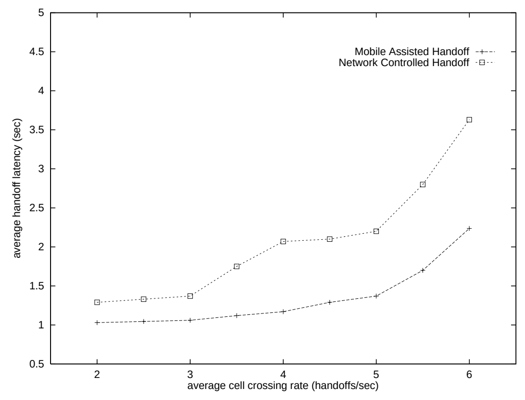
| | *wireline connection latency (msec)* | *wireless connection latency (msec)* | *measurement collection latency (msec)* | *total latency (msec)* |
|---|---|---|---|---|
| *Mobile Controlled Handoff* | 22 ± 1 | 19 ± 1 | - | 41 ± 1 |
| *Mobile Assisted Handoff* | 21 ± 1 | 20 ± 1 | 709 ± 65 | 750 ± 65 |
| *Network Controlled Handoff* | 22 ± 2 | 20 ± 1 | 641 ± 59 | 683 ± 59 |

**Table 2: Handoff Latency Measurement**

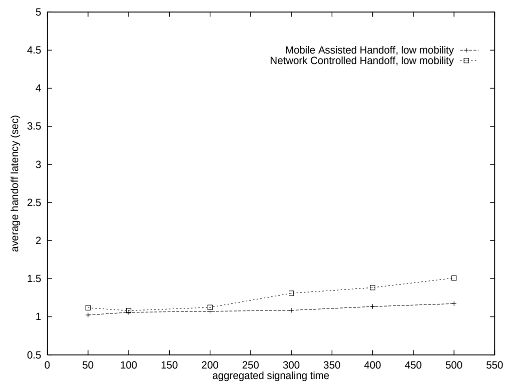
To test the scalability of our system we used the mobility emulator. We emulated the movement of 120 mobile devices and took measurements over a period of 10 min. Half the emulated mobile devices used mobile assisted handoff and the other half network controlled handoff. We measured the average handoff latency observed by each emulated mobile device and varied the average time between successive handoffs, which resulted in different cell crossing rates and signaling loads. The average handoff latency experienced by a single mobile device as a function of the cell crossing rate (characterizing the mobile environment) is presented in Figure 30(a). The figure shows that the average handoff latency for the mobile assisted and network controlled handoff styles is much higher when the system operates under signaling load. The average handoff latency experienced by a mobile device when performing mobile assisted handoff is 750 msec in the case of a lightly loaded testbed. However, the measured latency for the same handoff scheme is between 1.03 and 2.23 sec when the average cell crossing rate ranged between 2 handoffs/sec and 6 handoffs/sec. In the case of a network controlled handoff the average handoff latency measured is between 1.29 and 3.63 sec.

The increase in average handoff latency observed when the system operates under signaling load is partially caused by the fact that signaling functions are implemented as interactions between distributed objects. A common feature of many implementations in CORBA is that remote method invocations are queued until they are served in a first come-first served basis. However, in many distributed systems, such as programmable signaling platforms, service requests occur in bursts. In this case, significant latency can be introduced between the time a request is issued and the time it is served. Latency is introduced during the handoff process in the case of programmable handoff control. This latency is represented as the period between which the handoff execution call is issued by the detection algorithm and the point at which the mobility agent services the call. In the case of a network controlled handoff, this latency is compounded by the fact that access points transmit measurements reporting signal strength associated with mobile devices.

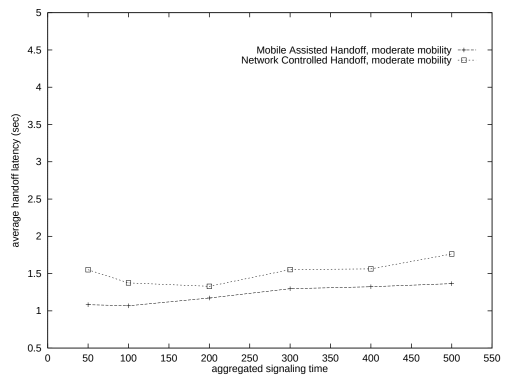
To minimize such latency, we enhanced mobile assisted handoff adapters to transmit aggregated service requests to mobility agents using a single remote method invocation (i.e., signaling interaction). In addition, we enhanced signal strength monitors to transmit aggregated signal strength reports. Aggregated calls carry all service requests that have been issued in the interval between two successive invocations. We call this interval between two successive aggregated calls the aggregated signaling time. We experimented with different values of aggregated signaling time and measured the average handoff latency experienced by mobile devices using mobile assisted and network controlled handoff. We took care that access points were not synchronized when transmitting aggregated calls, avoiding further increase in handoff latency.



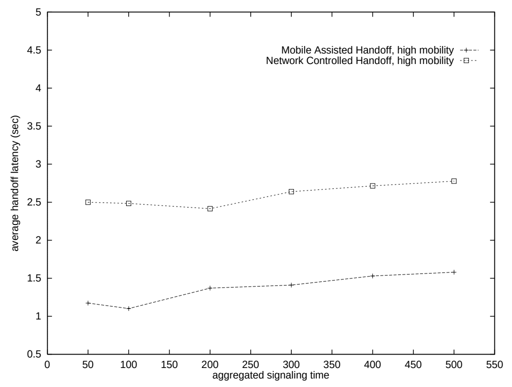
(a) Handoff Latency as a Function of
the Average Cell Crossing Rate



(b) Handoff Latency as a Function of
the Aggregated Signaling Time (Low Mobility
Case)



(c) Handoff Latency as a Function of
the Aggregated Signaling Time (Moderate Mobility
Case)



(d) Handoff Latency as a Function of
the Aggregated Signaling Time (High Mobility
Case)

**Figure 30: Handoff Latency Under Load Conditions**

Three sets of experiments evaluate the benefit of aggregated signaling techniques. In the first experiment, the average cell crossing rate of the mobile environment is 2 handoffs/sec. In the next experiment, the average cell crossing rate of the mobile environment is 4 handoffs/sec corresponding to more frequent mobility. In the third experiment, the average cell crossing rate of the mobile environment is 6 handoffs/sec. The results from these experiments are shown in Figures 30(b)-30(d).

Aggregation of CORBA calls at access points results in reduction in the average handoff latency experienced by mobile devices. Optimal values for the aggregated signaling time ranged between 50 and 200 msec. Higher aggregation signaling times result in increased latencies. Aggregation reduces the average handoff to 980 msec for the mobile assisted handoff scheme and 1.08 sec for the network controlled handoff scheme in the case of lowest cell crossing rate. In the case of the highest cell crossing rate the handoff latency is reduced to 1.10 sec for the mobile assisted handoff scheme and 2.41 sec for the network controlled handoff scheme. While it is difficult to compare results from different signaling systems operating under different load conditions we report for the purpose of qualitative analysis that the average handoff latency in MAHO cellular systems is around 0.9 sec. Network controlled handoff latency varies between 5-10 sec in different cellular systems.

| *signaling module* | *Size* *(Kbytes)* | *Loading time* *(sec)* |
|---|---|---|
| *Mobiware* | 1054 | 7 ± 1 |
| *Cellular IP* | 69 | 0.4 ± 0.1 |

**Table 3: Signaling Modules**

### 4.4.3 Reflective Handoff Analysis

In this section we evaluate the reflective handoff service between Mobiware and Cellular IP wireless access networks. Mobiware and Cellular IP represent rather different wireless access network architectures and their implementation platforms support strikingly different signaling protocols. In this respect supporting seamless handoff between these two networks is good test scenario for reflective handoff. For full details on Mobiware and Cellular IP handoff algorithms see [30] and [32], respectively.

An illustrative example of the performance of our video application is shown in Figure 31. Packet traces are shown for a video stream (i.e., true_lies.mpg) delivered to a mobile device on the downlink during reflective handoff. To study the effect of packet losses on reflective handoff we disable the promiscuous mode of WaveLAN radios. In this way, mobile devices are unable to simultaneously receive data from multiple access points. In the experiments, we successively force handoff between Mobiware and

Cellular IP access networks. Reflective handoff latency ranges between 60 and 100 msec. In the example shown in Figure 31, a Cellular IP to Mobiware handoff takes about 90 msec to complete, whereas a Mobiware to Cellular IP handoff takes approximately 60 msec, including entry and exit handoff. A significant part of the overall handoff latency is absorbed by the Mobile IP signaling component (42 msec over a single hop). Typically, loading times do not affect handoff performance because signaling modules are loaded prior to handoff execution. Activation latencies are 10 msec for the Mobiware and Cellular IP modules. Cellular IP only initiates care-of-address registration during 'entry handoffs'. Mobiware supports care-of address registration during both entry and exit handoffs.

Packet loss during Cellular IP to Mobiware reflective handoff is greater than Mobiware to Cellular IP because the care-of address registration occurs after the wireless radio link transfer takes place (i.e., a change in WaveLAN NWID). Mobiware to Cellular IP handoff is more efficient because care-of address registration occurs first and is initiated by the Mobiware access network during exit handoff. Some forwarding delay is introduced at the gateway that connects the Cellular IP access network with the Mobile IP enabled core network. Forwarding delay is introduced temporarily when compensating against the time required to complete reflective handoff. This forwarding delay is 60 msec in the Mobiware to Cellular IP reflective handoff example as shown in Figure 31. The performance cost of reflective handoff is increased jitter which many adaptive applications are capable of absorbing.

Table 3 summarizes the characteristics of the signaling modules used in our experiments. The Mobiware signaling module has been implemented using CORBA technology (Iona's ORBIX), while the Cellular IP module does not use CORBA. Signaling modules can have varying sizes and memory footprints, which affect their downloading time and performance. The size of the Mobiware and Cellular IP signaling modules are 1054 Kbytes and 69 Kbytes, respectively. Mobiware implements a complex QOS-adaptive mobility management architecture, while Cellular IP implements a simple in-band handoff scheme without QOS support. The implementation of the Mobiware signaling module is based on a commercial Object Request Broker (ORB), which is not customized for wireless environments or for on-demand code loading. This results in a significant footprint and loading time (7 sec). Clearly, this result indicates that reflective handoff is not a practical solution for the current ORB and Mobiware signaling module, especially when one considers a 2 Mbps air interface. Using radios with speeds of 10 and 25 Mbps reduces the loading times to 1.4 sec and 0.56 sec, respectively. Future work will include porting the programmable handoff architecture to a 'light-weight' microORB [107], which will further decrease the footprint of programmable signaling modules.

Our implementation of reflective handoff operates well in our testbed where signaling modules are loaded over slow time scales. Soft state timers associated with module caches operate over tens of minutes. We plan to investigate scalability issues associated with the reflective handoff scheme and its applicability to mobile devices with varying loading requirements and processing capabilities.

## 4.5 Related Work

While the research community has addressed the programmability of the physical layer [20, 21, 98, 99] and quality of service control [32, 94, 95] in mobile networks, little work has been done on using programmability for solving the inter-system handoff problem. On the other hand a variety of 'monolithic' handoff algorithms have been proposed and investigated in the past [123, 138, 145] but these algorithms are mostly tailored toward the needs of some specific type of mobile device or access network.

Programmable mobile networks represent an emerging area of research. In [20, 21, 98, 99] the programmability of the physical layer is addressed based on digital signal processing techniques and wide-band analog-to-digital conversion. A programmable MAC framework is presented in [34] that supports adaptive real-time applications over time-varying and bandwidth-limited networks allowing mobile applications to map their service requirements into groups of 'bearer' traffic classes supported by a programmable scheduler. Active networking services for wireless and mobile networks are discussed in [85].

In this chapter, we investigate technology for connecting mobile devices to programmable mobile networks. Our methodology focuses on a framework for making handoff programmable. Handoff represents an important service in wireless networks characterizing the capability of access networks to respond to mobile user requirements. We believe that the capability of making mobile networks programmable will help accelerate the deployment of new services in mobile and wireless networks and speed innovation.

## 4.6 Summary

In Chapter 4 we address the problem of supporting end-system connectivity focusing on programmable mobile networks. Supporting end-system connectivity in programmable mobile networks is not an easy task because mobile devices may roam across access networks with heterogeneous mobility management architectures. While a variety of handoff algorithms have been proposed and investigated in the past these algorithms are mostly tailored toward the needs of some specific type of mobile device or access network. In this chapter we propose a solution to the intersystem handoff problem where the implementation details of mobility management algorithms are hidden from handoff control systems, allowing the handoff detection state (e.g., the best candidate access point for a mobile device) to be managed separately from handoff execution state (e.g., mobile registration information). The same detection algorithms operating in mobile devices, or access networks can interface with multiple types of mobility management architectures, operating in heterogeneous access networks.

The main results of our work are: (i) we present the design, implementation and evaluation of a 'reflective handoff' service that allows access networks to dynamically inject signaling systems into mobile devices before handoff. Thus, mobile devices can seamlessly roam between wireless access networks that support radically different mobility management systems; and (ii) we show how a 'multi-handoff' access network service can simultaneously support different styles of handoff control over the same wireless access network. This programmable approach can benefit service providers who need to be able to satisfy the mobility management needs of a wide range of mobile devices from cellular phones to more sophisticated palmtop and laptop computers.  To allow a range of mobile devices to connect to programmable mobile networks we further decompose handoff control process into programmable objects, separating the transmission of beacons, from the collection of wireless channel quality measurements and from the handoff detection algorithm.

# Chapter 5

# Programming the Data Path

## 5.1 Introduction

In Chapter 5 we study the performance of the Genesis Kernel programming system, discussed in Chapter 3. In particular, we focus on the problem of efficiently programming the data path. We focus our study on a network processor-based implementation of the Genesis Kernel because network processors are suitable building blocks for software-base routers, comprising multiple processing units for parallel packet processing. Recently, there has been a growing interest in network processor technologies [68-72] that can support software-based implementations of the critical path while processing packets at high speeds. Network processors use specialized architectures that employ multiple processing units to offer high packet-processing throughput. We believe that introducing programmability in network processor-based routers is an important area of research that has not been fully addressed as yet. The difficulty stems from the fact network processor-based routers need to forward minimum size packets at line rates and yet support modular and extensible data paths. Typically, the higher the line rate supported by a network processor-based router the smaller the set of instructions that can be executed in the critical path.

Data path modularity and extensibility requires the dynamic binding between independently developed packet processing components. While code modularity and extensibility is supported by programming environments running in host processors (e.g., high level programming language compilers and linkers), such capability cannot be easily offered in the network. Traditional techniques for realizing code binding, (e.g., insertion of code stubs or indirection through function tables) introduce some overhead when applied to network processors in terms of additional instructions in the critical path. This overhead can be significant in some network processors. One solution to this problem is to optimize the code produced by a binding tool, once data path composition has taken place. Code optimization algorithms can be complex and time-consuming, however. For example, code optimization algorithms may need to process each instruction in the data path code several times resulting in $O(n)$ or higher complexity as a function of the number of instructions in the critical path. Such algorithms may not be suitable for applications that require fast data path composition, (e.g., data path composition on a packet by packet basis). We believe that a binding tool for network processor-based routers needs to balance the flexibility of network programmability against the need to process and forward packets at line rates. This poses significant challenges.

In this chapter, we present the design, implementation and evaluation of NetBind, a high performance, flexible and scalable binding tool for creating modular data paths in network processor-based routers. By "high performance" we mean that NetBind can produce data paths that forward minimum size packets at line rates without introducing significant overhead in the critical path. NetBind modifies the machine language code of components at run time, directing the program flow from one component to another. In this manner, NetBind avoids the addition of code stubs in the critical path.

By "flexible" we mean that NetBind allows data paths to be composed at fine granularity from components supporting simple operations on packet headers and payloads. NetBind can create packet-processing pipelines through the dynamic binding of small pieces of machine language code. A binder modifies the machine language code of executable components at run-time. As a result, components can be seamlessly merged into a single code piece. For example, in [80] we show how Cellular IP [140] data paths can be composed for network processor-based radio routers.

By “scalable” we mean that NetBind can be used across a wide range of applications and time scales. In order to support fast data path composition, NetBind reduces the number of binding operations required for constructing data paths to a minimum set so that binding latencies are comparable to packet forwarding times. In NetBind, data path components export symbols, which are used during the binding process. Each symbol is associated with some specific instruction executed in the critical path (e.g., a branch instruction or an arithmetic logic unit instruction that uses a critical register). Not all instructions in the critical path are exported as symbols, however. The number of symbols exported by data path components, $h$, is typically much smaller than the total number of instructions executed in the critical path, $n$. The NetBind binding algorithm does not inspect every instruction in the data path code but only the symbols exported by data path components. Because of the fact that $h$ is much smaller than $n$, the time it takes to inspect every exported symbol is typically much smaller than the time it takes to inspect every instruction in the data path code. For this reason, the NetBind binding algorithm can compose packet processing pipelines very fast, in the order of microseconds. The NetBind binding algorithm is associated with $O(h)$ complexity.

While the design of NetBind is guided by a set of general principles that make it applicable to a class of network processors, the current implementation of the tool is focused toward the Intel IXP1200 network processor. The IXP1200 network processor targets fast Ethernet, OC-3 and OC-12 line rates. We have not yet investigated forwarding at Gigabit speeds (e.g., OC-48, OC-192). However, there is strong indication that NetBind can be applied to these line rates due to the simplicity and generality of the binding process.

This chapter is structured as follows. In Section 5.2 we present an overview of network processor architectures, and IXP1200 in particular, and discuss issues and design choices associated with dynamic binding. Specifically, we investigate tradeoffs that are associated with different design choices and discuss their implications on the performance of the data path. In Section 5.3, we present the design and implementation of NetBind. We discuss NetBind implementation is Section 5.4. In Section 5.5 we describe how an IPv4 data path can be constructed using NetBind. In Section 5.6, we use a number of NetBind created IPv4 [31] data paths to evaluate the performance of the system. We also evaluate the MicroACE system [71] developed by Intel to support binding between software components running on Intel IXP1200 network processors, and identify pros and cons in comparison to NetBind. Section 5.7 discusses the related work, and finally, in Section 5.8, we provide some concluding remark.

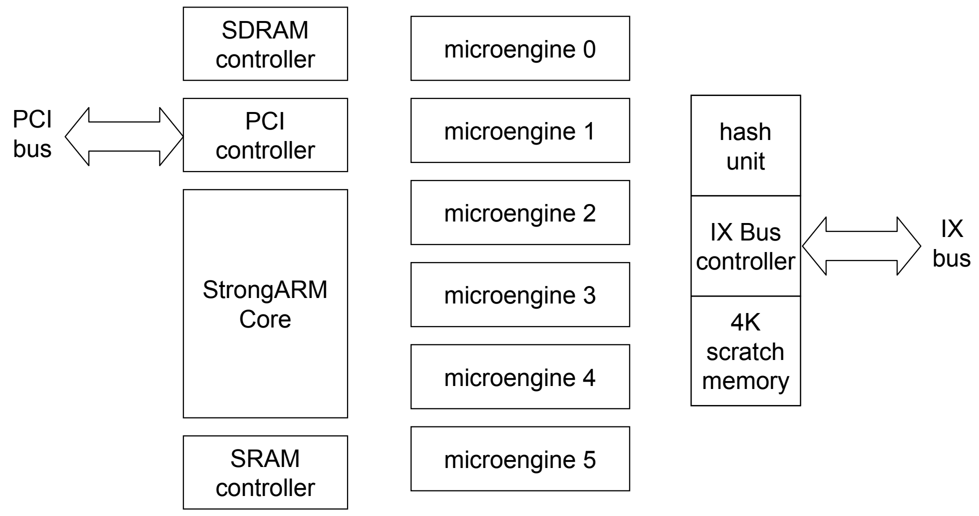


**Figure 31: Internal Architecture of the IXP1200**

## 5.2 Dynamic Binding in Network Processors

### 5.2.1 Network Processors

**Features**

A common practice when designing and building high performance routers is to implement the fast path using Application Specific Integrated Circuits (ASICs) in order to avoid the performance cost of software implementations. ASICs are usually developed and tested using Field Programmable Gate Arrays, which are arrays of reconfigurable logic. Network processors represent an alternative approach to ASICs and FPGAs, where multiple processing units, (e.g., the microengines of Intel's IXP network processors [69-72] or the dyadic protocol processing units of IBM's PowerNP processors [68]) offer dedicated computational support for parallel packet processing. Processing units often have their own on-chip instruction and data stores. In some network processor architectures, processing units are multithreaded. Hardware threads usually have a separate program counter and manage a separate set of state variables. However, each thread shares an arithmetic logic unit and register space. Network processors do not only employ parallelism in the execution of the packet processing code, rather, they also support common networking functions realized in hardware, (e.g., hashing [69], classification [68] or packet scheduling [68]). Network processors typically consume less power than FPGAs and are more programmable than ASICs.

## The IXP1200 Network Processor

IXP1200 network processor incorporates seven RISC CPUs, a proprietary bus (IX bus) controller, a PCI controller, control units for accessing off-chip SRAM and SDRAM memory chips, and an on-chip scratch memory. The internal architecture of the IXP1200 is illustrated in Figure 31. In what follows, we provide an overview of the IXP1200 architecture. The information presented here about the IXP1200 network processor is sufficient to understand the design and implementation of NetBind. For further details about the IXP1200 network processor see [70]. One of the IXP1200 RISC CPUs is a StrongARM Core processor running at 200 MHz. The StrongARM Core can be used for processing slow path exception packets, managing routing tables and other network state information. The other six RISC CPUs, called "microengines" are used for executing the fast path code. Like the StrongARM Core, microengines run at 200 MHz. Each microengine supports four hardware contexts sharing an arithmetic logic unit, register space and instruction store.

Each microengine has a separate instruction store of size 1K instructions (i.e., 4K bytes) called "microstore". Unlike the StrongARM Core processor, microengines do not use instruction or data caches because of their associated performance implications. For example using a data cache in the critical path is effective only when it is easy to determine whether some portion of the data path structures (e.g., classification, forwarding or scheduling data structures) is used more often that others. Each microengine incorporates 256 32-bit registers. Among these registers, 128 registers are General Purpose Registers (GPRs) and 128 are memory transfer registers. The register space of each microengine is shown in Figure 32. Registers are used in the following manner. GPRs are divided into two banks, (i.e., banks A and B, shown in Figure 32), of 64 registers each. Each GPR can be addressed in a "context relative" or "absolute" addressing mode. By context relative mode, we mean that the address of a register is meaningful to a particular context only. Each context can access one fourth of the GPR space in the context relative addressing mode. By absolute mode, we mean that the address of a register is meaningful to all contexts. All GPRs can be accessed by all contexts in the absolute addressing mode.

The memory transfer registers are divided between SRAM and SDRAM transfer registers. SRAM and SDRAM transfer registers are further divided among "read" and "write" transfer registers. Memory transfer registers can also be addressed in context-relative and absolute addressing modes. In the context relative addressing mode, eight registers of each type are accessible on a per-context basis. In the absolute addressing mode, all 32 registers of each type are accessible by all contexts.

The IXP1200 uses a proprietary bus interface called “IX bus” interface to connect to other networking devices. The IX bus is 64-bit wide and operates at 66 MHz. In the evaluation boards we use for experimenting with NetBind, the IXP1200 is connected to four fast Ethernet (100 Mbps) ports. The IX bus controller (shown in Figure 31) incorporates two FIFOs for storing minimum size packets, a hash unit and a 4K “scratch” memory unit. The scratch memory is used for writing or reading short control messages, which are exchanged between the microengines and the StrongARM Core.

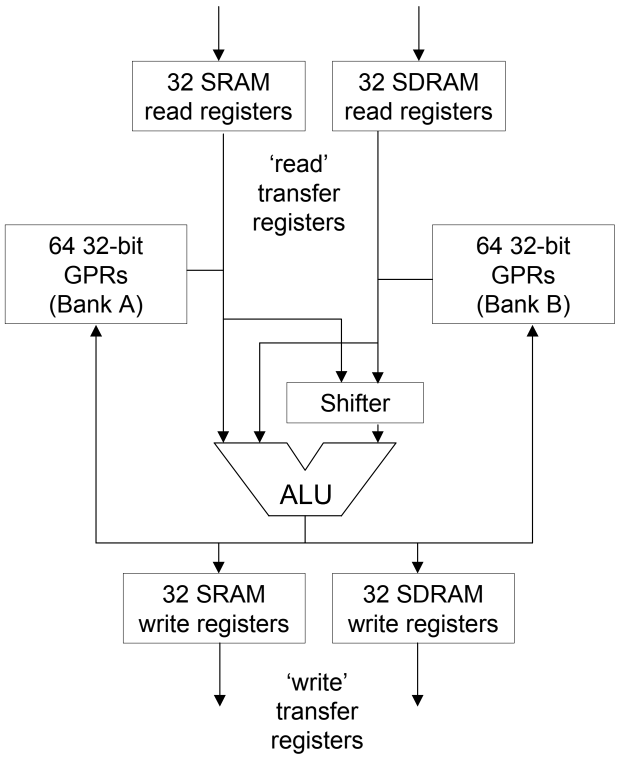


**Figure 32: Microengine Registers**

### 5.2.2 Dynamic Binding Issues

There are many different techniques for introducing new services into software-based routers [45, 46, 133, 135, 148]. At one end of the spectrum, the code that implements a new service can be written in a high level, platform-independent programming language (e.g., Java) and compiled at runtime producing optimized code for some specific network hardware. In contrast, data paths can be composed from packet processing components. Components can be developed independently from each other, creating associations at run time. This dynamic binding approach reduces the time for developing and installing new services, although it requires that algorithmic components are developed and tested in advance. In what follows, we discuss issues associated with the design of a dynamic binding system for network processor-based routers.

The main issues associated with the design of a binding system for network processor-based routers can be summarized as:

- Headroom limitations;
- Register space and state management;
- Choice of the binding method;
- Data path isolation and admission control;
- Processor handoffs;
- Instruction store limitations; and
- Complexity of the binding algorithm.

**Headroom Limitations**

Line rate forwarding of minimum size packets (64 bytes) is an important design requirement for routers. Routers that can forward minimum size packets at line rates are typically more robust against denial-of-service attacks, for example. Line rate forwarding does not mean zero queuing. Rather, it means that the output links of routers can be fully utilized when routers forward minimum size packets.

A necessary condition for achieving line rate forwarding is that the amount of time dedicated to the processing of a minimum size packet does not exceed the packet's transmission or reception times, assuming a single queuing stage. If multiple queuing stages exist in the data path, then the processing time associated with each stage should not exceed the packet's transmission or reception times.

Given that the line speeds at which network processor-based routers operate are high, (e.g., in the order of hundreds of Mbps or Gbps), the amount of instructions that can be executed in the critical path is typically small, ranging between some tens to hundreds of instructions. The amount of instructions that can be executed in the critical path without violating the line rate forwarding requirement is often called headroom.

Headroom is a precious resource in programmable routers. Traditional binding techniques used by operating systems or high-level programming languages are not suitable for programming the data path in network processor-based routers because these techniques waste too much headroom, and, thereby limit the performance of the router. This is because such binding techniques burden the critical path with unnecessary code stubs, or with time-consuming memory read/write operations for accessing function tables. An efficient binding technique needs to minimize the amount of additional instructions introduced into the critical path. The code produced by a good dynamic binding tool should be as efficient, and as optimized, as the code produced by a static compiler or assembler.

**Register Space and State Management**

The exchange of information between data path components typically incurs some communication cost. The performance of a modular data path depends on the manner in which the components of the data path exchange parameters between each other. Data transfer through registers is faster and more efficient than memory operations. Therefore, a well-designed binding tool should manage the register space of a network processor system such that the local and global state information is exchanged between components as efficiently as possible.

The number of parameters which are used by a component determines the number of registers a component needs to access. If this number is smaller than the number of registers allocated to a component, the entire parameter set can be stored in registers for fast access. The placement of some component state in memory impacts the data path's performance. Although modern network processors support a large number of registers, register sets are still small in comparison to the amount of data that needs to be managed by components. Transfer through memory is necessary when components are executed by a separate set of hardware contexts and processing units. A typical case is when queuing components store packets into memory, and a scheduler accesses the queues to select the next packet for transmission into the network.

For fast data path composition, register addresses need to be known to component developers in advance. A component needs to place parameter values into registers so they can be correctly accessed by the next component in the processing pipeline. There are two solutions to this problem. First, the binding mechanism can impose a consensus on the way register sets are used. Each component in a processing pipeline can place parameters into a predetermined set of registers. The purpose of each register, and its associated parameter, can be exposed as a programming API for the component.

A second solution is more computationally intensive. The binding tool can scan all components in a data path at run time and make dynamic register allocations when the data path is constructed or modified. In this case, the machine language code that describes each component needs to be modified reflecting the new register allocations made by the binding tool. Such code optimization algorithm may be time consuming, and not suitable for applications requiring fast data path composition.

**Choice of the Binding Method**

Apart from the manner in which parameters are exchanged between components, the choice of the binding technique significantly impacts the performance of a binding algorithm. There are three methods that can be used for combining components into modular data paths. The first method is to insert a small code stub that implements a dispatch thread of control into the data path code. The dispatch thread of control directs the program flow from one component to another based on some global binding state and on the parameters returned by each module. This method is costly. The minimum amount of state that needs to be checked before the execution of a new component is an identifier to the next module and a packet buffer handle exchanged between components. Checking this amount of state requires at least four compute cycles. If a data path is split between six components, then the total amount of overhead introduced in the critical path is twenty four compute cycles which is a significant part of the network processor headroom in many different network processors. For example in IXP network processors that target the OC-48 and OC-192 line rates, the headroom is equal to 57 compute cycles. In this case the binding overhead accounts for 42% of the headroom. The dispatch loop approach is more appropriate for static rather than dynamic binding and can impact the performance of the data path because of the overhead associated with the insertion of a dispatch code stub.

An enhancement on the first method for dynamic binding adds a small vector table into memory. The vector table contains the instruction store addresses where each component is located. A data path component obtains the address of the next component in the processing pipeline in one memory access time. In this approach, no stub code needs to be inserted in the critical path. When a new component is added, only the content of the vector table needs to be modified. Although this approach is more suitable for dynamic binding, it involves at least one additional memory read operation for each component in the critical path. In commercial network processors that target OC-48 and OC-192 speeds memory access latencies can be several times larger than the network processor headroom (e.g., 100-300 compute cycles). As a result it is difficult to apply the vector table technique in order to support modular data paths at such speeds.

The third binding method is more interesting. Instead of deploying a dispatch loop, or using a vector table, the binding tool can modify the components' machine language code at run time, adjusting the destination addresses of branch instructions. No global binding state needs to be maintained with this approach. Each component can function as an independent piece of code having its own “exit points” and “entry points”. Exit points are instruction store addresses of branch instructions. These branch instructions make the program flow jump from one component to another. Entry points are instruction store addresses where the program flow jumps. The impact of this approach on network processor headroom can be significant since instructions that check global binding state are omitted. The third binding method introduces less overhead in terms of additional processing latency in the critical path. For these reasons, we have used the third binding method in NetBind. In this thesis we refer to this method as ‘code morphing’ method because it modifies the machine language code of components at run time.

**Data Path Isolation and Admission Control**

To forward packets without disruption, data paths sharing the resources of the same network processor hardware need to be isolated. In addition, an admission control process needs to ensure that the resource requirements of data paths are met. Resource assignments can be controlled by a system-wide entity. Resources in network processors include bandwidth, hardware contexts, processing headroom, on-chip memory, register space, and instruction store space.

One way to support isolation between data paths is to assign each data path to a separate processing unit, or a set of hardware contexts, and to make sure that each data path does not execute code that exceeds the network processor headroom. Determining the execution time of programs given some specific input is typically an intractable problem. However, it has been shown that packet processing components can have predictable execution times [109]. One solution to the problem is to allow code modules to carry their developer's estimation of the worst case execution time in their file headers. The time reported in each file's header should be trusted, and code modules should be authenticated using well-known cryptographic techniques. To determine the worst case execution time for components, developers can use reasonable upper bounds for the time it takes to complete packet processing operations.

Bandwidth can be partitioned using packet scheduling techniques. Packet scheduling techniques, (e.g., deficit round robin [75], weighted fair queuing [75], or start time fair queuing [60]), can be implemented either in the network processor hardware, or, as part of a packet-processing pipeline. Hierarchical packet scheduling algorithms [14] can be used for dividing bandwidth between a hierarchy of coexisting data paths. The implementation of packet scheduling algorithms in network processors is beyond the scope of this thesis.

**Processor Handoffs**

Sometimes the footprints of data paths can be too large so that they cannot be placed in the same instruction store. In this case, the execution of the data path code has to be split across two, or more processing units. Another case arises when multiple data paths are supported in the same processor. In this case, the available instruction store space of processing units may be limited. As a result, the components of a new data path may need to be distributed across multiple instruction stores. Third, the execution of a data path may be split across multiple processing units when "software pipelining" [73] is employed for increasing the packet rate that can be achieved in a network processor architecture. Software pipelining is a technique that divides the functionality of a data path into several stages. Pipeline stages can potentially run in different processing units. Multiple stages can be executed at the same time forwarding the packets of different data flows.

We call the transfer of execution from one processing unit to another (that takes place when a packet is being processed), "processor handoff". Processor handoffs impact the performance of data paths and need to be taken into account by the binding system. The performance of processor handoffs depends on the type of memory used for communication between processing units. A dynamic binding system should try to minimize the probability of having high latency processor handoffs in the critical path. This is not an easy task and requires a search to be made on all possible ways to place the code of data paths into the instruction stores of processing units.

### Instruction Store Limitations

Instruction store limitations represent a constraint on the number of data paths or processing functions that can be simultaneously executed in the same network processor. A solution to this problem would be to have the binding system fetch code from off-chip memory units into instruction stores on an on-demand basis. This solution, however, can significantly impact the performance of the critical path because of the overhead associated with accessing memory units.

### Complexity of the Binding Algorithm

The last consideration for designing a dynamic binding system is the complexity of the binding algorithm. In many cases, the complexity of a binding algorithm affects the time scales over which the binding algorithm can be applied. A complex binding algorithm needs time to execute, and is typically, not suitable for applications that require fast data path composition. Keeping the binding algorithm simple while producing high performance data paths is an important design requirement for a good binding system. Applications that require real-time binding include classification, forwarding and traffic management. The performance of such data path algorithms depends on the properties of classification databases [61, 83], routing tables and packet scheduling configurations [119]. These properties typically change at run time calling for advanced service creation environments for programming the data path efficiently. A dynamic binding system should ideally support a wide range of packet processing applications ranging from the creation of virtual networks over slow time scales to the fast creation of customized data paths, after disasters occur. Disasters typically result in rapid changes on the input traffic characteristics and topologies of communication networks. To accommodate increased traffic demands or to reroute traffic to alternate links once some part of the communication infrastructure is physically damaged, communication networks need to be highly adaptive, calling for new techniques and software methodologies for rapid service creation.

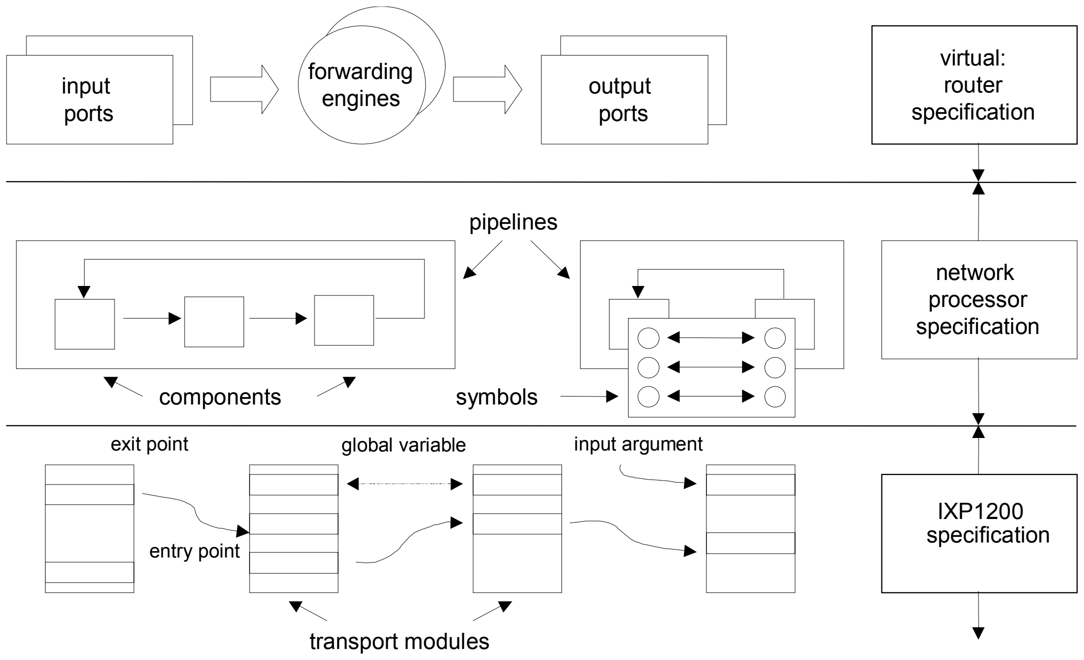


**Figure 33: Data Path Specification Hierarchy**

## 5.3 NetBind Design

NetBind is a binding tool we have developed that offers dynamic binding support for the IXP1200 network processor and is part of the composition controller of the routelet architecture of the Genesis Kernel (see Chatpter 3). NetBind consists of a set of libraries which can modify IXP1200 instructions, create processing pipelines or perform higher-level operations such as data path admission control. Components are written in machine language code called microcode. NetBind groups components into processing pipelines that execute on the microengines of IXP1200.

### 5.3.1 Design Principles

The most fundamental principle in the design of NetBind is that binding is performed by avoiding the use of code stubs or indirection due to their associated performance penalty. As a consequence, NetBind modifies the machine language code of components at run-time in order to construct consistent packet processing pipelines. Since the minimum step required for advancing the program sequence from one component to another is a branch operation, NetBind modifies the destination addresses of branch instructions connecting components. In this way, NetBind allows the program sequence to continue to the next component in a pipeline, after the previous component's code is executed.

NetBind uses no global binding state. Because of this reason, NetBind requires that components expose registers used for inter-component communication as a programming API. Registers are exported as binding symbols. NetBind splits the register space into regions and imposes a consensus on how registers are used. Parameters exchanged between components do not need to be stored as global binding state. In addition, no global biding state needs to be checked every time a new component's code executes. This reduces the overhead introduced in the critical path significantly, as discussed in the evaluation section. Registers are used in three different ways in the components we experimented with. First, components use registers to hold input argument values, as explained in detail below. Second, components use registers to exchange information produced and consumed during the execution of a packet processing pipeline. Third, components use registers to share information between each other. We distinguish registers between 'input argument', 'pipeline' and 'global variable' registers depending on the way registers are used. By exporting registers as input argument, pipeline and global variable, components can communicate with each other with the least possible communication cost.

A second principle followed in the design of NetBind is that binding is performed by inspecting a much smaller number of instructions than the body of code executed in the critical path. This second principle results in fast data path composition. To reduce the number of instructions inspected during binding, NetBind requires from components to allocate register addresses statically. Statically allocated register addresses are exported as part of each component's API. The only exception to this rule concerns the allocation of global variable register addresses. Global variables represent a shared resource because they are accessed by multiple simultaneously running hardware threads and referenced using the absolute addressing mode. As a result, global variables addresses need to be allocated dynamically on an on-demand basis to packet processing pipelines.

### 5.3.2 Data Path Specification Hierarchy

Before creating data paths, NetBind captures their structure and building blocks in a set of executable profiling scripts. NetBind uses multiple specification levels to capture the building blocks of packet forwarding services and their interaction. NetBind uses multiple specification levels in order to offer the programmer the flexibility to select the amount of information that can be present in data path profiling scripts. Some profiling scripts are generic, and thus applicable to any hardware architecture. Some other profiling scripts can be specific to network processor architectures, potentially describing timing and concurrency information associated with components. A third group of scripts can be specific to a particular chip such as the IXP1200 network processor.

Figure 33 illustrates the different ways data paths are profiled in NetBind. First, a virtual router specification can be applied to any hardware architecture. The virtual router specification is part of the compact form of the Genesis profiling script (see Chapter 3). The virtual router specification is generic and can be applied to many different types of programmable routers (e.g., PC-based programmable routers [46, 79], software [148] or hardware [133] plugin-based routers or network processor-based routers [128, 129]).The purpose of the virtual router specification is to capture the composition of a router in terms of its constituent building blocks and their interaction, without specifying the method which is used for component binding. Programmable routers can be constructed using a variety of programming techniques ranging from higher level programming languages (e.g., Java) to hardware plugins based of FPGA technologies. The virtual router specification can be used for network spawning in a heterogeneous infrastructure of programmable routers of many different types. The virtual router specification describes a virtual router as a set of input ports, output ports and forwarding engines. The components that comprise ports and engines are listed, but no additional information is provided regarding the contexts that execute the components and the way components create associations with each other. There is no information about timing and concurrency in this specification.

A network processor specification augments the virtual router specification with information about the number of hardware contexts that execute components and about component bindings. The network processor specification exposes information about the hardware contexts that execute data paths in order to allow the programmer to control the allocation of computational resources (i.e., hardware threads and processing units).

Components are grouped into processing stages. A processing stage is a part of a software pipeline and consists of a set of components that are executed by one or multiple hardware contexts sequentially. Components exchange packets between each other in a “push” or a “pull” manner. Components that sequentially exchange packets in a push manner are grouped into the same stage of a pipeline. A data path software pipeline is split between at least two stages if components perform different operations on packets simultaneously. For example, components may need to place packets (or pointers to packets) into memory, while other components may need to concurrently process or remove packets from memory. In this case, the first set of components should be executed by a separate set of hardware contexts other than those, which execute the second set.

In the network processor related specification, components are augmented with symbols, which are represented as strings. The profiling script specifies one-to one bindings between symbols and between components. Symbols abstract binding properties of components such as registers or instruction store addresses. Symbols are used in the binding process.

The network processor specification is dynamically created from the virtual router specification. The virtual router specification does not include information about symbol bindings. Because of this reason, the virtual router specification is not sufficient to describe the interaction between components at the network specification level. To convert the virtual router specification into the network processor specification, NetBind queries a database which stores information about symbol bindings. Groups of symbol bindings are being stored for each component binding used in the data path. The NetBind binding approach assumes that the number of components and symbols used for constructing data path algorithms is limited (i.e., in the same order of magnitude as the classes and methods found in a higher level programming language API). If this assumption is true, it may not be difficult for component developers to specify exactly which symbol bindings should characterize the interaction between components in the data path.

An IXP1200 specification relates the components of a programmable data path with binding properties associated with the IXP1200 architecture. This type of specification shows how data paths are constructed for a specific network processor. Components are implemented as blocks of instructions (microwords) called transport modules. Each transport module supports a specific set of functions. Transport modules can be customized or modified during the binding process. Symbols are specified as “entry points”, “exit points”, “input arguments” or “global variables”.

Exit points are instruction store addresses of branch instructions. These branch instructions make the program flow jump from one transport module to another. Entry points are instruction store addresses where the program flow jumps. Input arguments are instruction store addresses of “immed” IXP assembler instructions that load GPRs with numeric values. These numeric values, (e.g., Cellular IP timers, IPv4 interface addresses), customize the operation of transport modules. Global variables are GPRs that are accessed using the absolute addressing mode. These GPRs hold numeric values that are shared across pipelines or data paths. For example, the SRAM address of a packet buffer in an IPv4 data path needs to be declared as global variable since this value is shared between the packet buffer and a scheduler. Network processor related specifications are automatically translated into IXP12000 related specifications by the NetBind binding system.

### 5.3.3 Register Allocations

Register allocations realized in NetBind are shown in Figure 34. We observe that in data path implementations we have experimented with, registers are used in three ways. First, registers can hold numeric values used by a specific component. Each component implements an algorithm that operates on numeric values. When a component creates new values, it replaces old numeric values with the new ones in the appropriate microengine registers. We call these registers pipeline registers. The algorithm of each component places some numeric values into pipeline registers, which are used by the algorithm of the next component in the pipeline. In this manner pipeline registers are shared among all components of a pipeline. Pipeline registers are accessed using the context relative addressing mode. Once a component executes it becomes the owner of the entire set of pipeline registers. In this way, the registers used by different contexts are isolated.

Second, registers can hold input arguments. Input arguments are passed dynamically into components when pipelines are created from an external source, (e.g., the control unit of a virtual router [80]). Each component can place its own arguments into input argument registers overwriting the input arguments of the previous component. Similar to pipeline registers, input argument registers are accessed in a context relative addressing manner. Examples of input arguments include the SRAM address where a linked list of packet buffer descriptors is stored, the SDRAM address where a routing table is located, or, the scratchpad address, where a small forwarding MIB is maintained.

Third, some registers are shared among different pipelines or data paths. We call these registers global variable registers. Global variable registers are exported as global variable symbols. These registers need to be accessed by multiple hardware contexts simultaneously. For this reason, global variable registers are accessed using the absolute addressing mode.

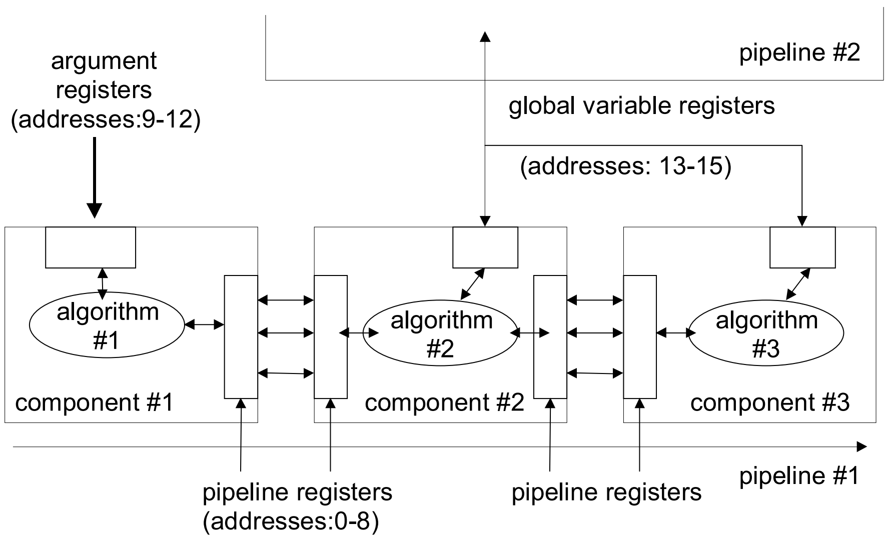


**Figure 34: Register Allocation in NetBind**

To reduce the time required for performing data path composition, NetBind uses static register allocation for pipeline and input argument registers and dynamic allocation for global variable registers. By static register allocation, we mean that register addresses are known to component developers and exported as a programming API for each component. By dynamic register allocation, we mean that register addresses are assigned at run time when data paths are created or modified. An admission controller assigns global variable registers to data paths on-demand. The microcode of components is modified to reflect the register addresses allocated to components in order to hold global variables. NetBind uses static register allocation for pipeline and input argument registers in order to simplify the binding algorithm and to reduce the time needed for the creation a modular data path. Otherwise, the NetBind binding algorithm would have to inspect all instructions in the data path code, locate the instructions where registers are used and assign register addresses dynamically.

In the current implementation of NetBind, we use eighteen GPRs per context as pipeline registers (addresses 0-8 of banks A and B) and eight GPRs per context as input argument registers (addresses 9-12 of banks A and B). Memory transfer registers are all pipeline registers. Each context contributes six GPRs in order to be used as global variable registers. In this manner, a pool of twenty-four global variable registers are shared among pipelines or data paths. Dynamic register allocation is used for global variable registers, because these registers are shared between different pipelines or data paths and it is not possible for their addresses to be known in advance. In our Spawning Networks Testbed, as part of the Genesis Project, we have used NetBind to build IPv4 [115] and Cellular IP [140] virtual routers. Static register allocation is sufficient for programming this set of diverse data paths.

Our characterization of registers as 'input argument', 'pipeline' and 'global variable' reflects register usage in the components we experimented with. Alternative designs may define different types of register usage. What is important in the design of NetBind is that we allow components to communicate without checking global binding state every time a new component is executed. This is accomplished by allowing components to expose registers as a programming API. Registers can be characterized in many different ways in the API depending on the programmer's needs.

### 5.3.4 Binding Algorithm

The NetBind binding algorithm operates on components and modifies their microcode instructions at run time in order to compose packet processing pipelines A description of the binding algorithm is given below.

```
NETBIND-BIND (component-list, binding-list)
1.  for i = 1 to number-of-components do
2.      place each component into the same microstore
3.  for i = 1 to number-of-components do
```

```
4.  for j = 1 to number-of-input-arguments for the i-th component do
5.      j-th-input-argument ← exported-symbol-value for this argument
6.  for j = 1 to number-of-global-variables for the i-th component do
7.      j-th-global variable ← exported-register-address for this variable
8.  for i = 1 to number-of-bindings do
9.      exit-point for the i-th binding ← exported-entry poin for this exit point
```

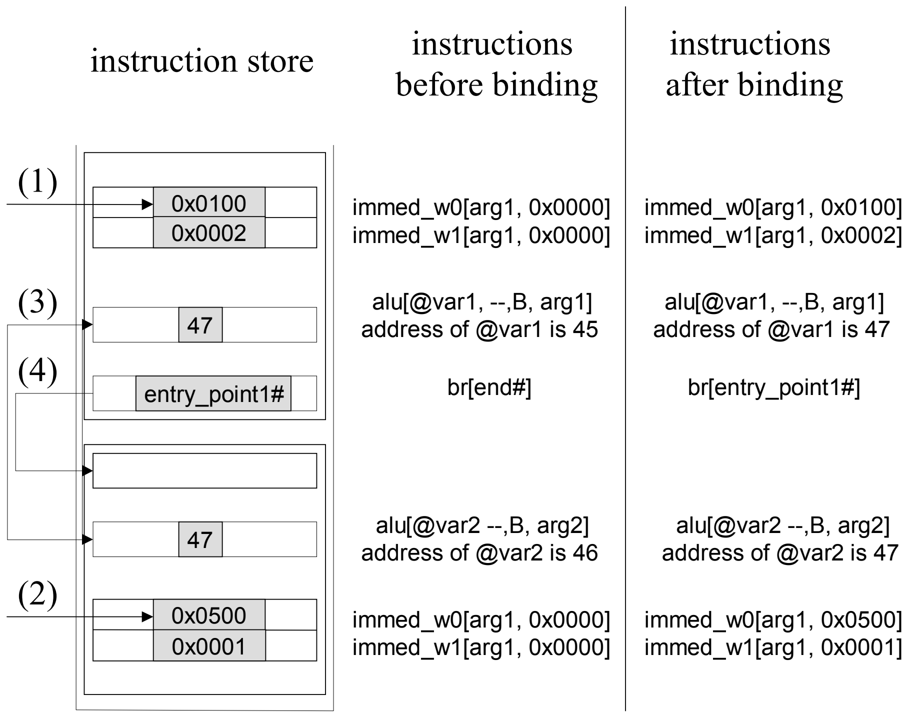


**Figure 35: Dynamic Binding in NetBind**

Steps 1, 2 describe the placement of components into a microstore. Steps 3-7 describe the assignment of input argument values and global variable register addresses. Steps 8, 9 describe the bindings between exit points and entry points. Assuming steps 1 and 2 can be executed in bounded time (e.g., by a DMA transfer unit or other hardware acceleration unit) and that the component list passed as input contains *m* input arguments, *n* global variables, and *l* bindings the complexity of the binding algorithm is O(*m*+*n*+*l*). This is because steps 3-5 are executed *m* times overall, steps 6, 7 are executed *n* times, and steps 8, 9 are executed *l* times. The total number of symbols that are processed during the binding process is h = m+n+l. As a result, the complexity of our binding algorithm is O(h) as stated earlier.

Figure 35 illustrates an example how NetBind performs dynamic binding. In this example, two components are placed into an instruction store. Figure 35 shows the instruction store containing the components and the instructions of components, which are modified during the binding process. The fields of instructions, which are modified by NetBind, are illustrated as shaded boxes in the figure.

First, NetBind modifies the microwords that load input arguments into registers. Input argument values are specified using the NetBind programming API. Input argument values replace the initial numeric values used by the components. In the example of Figure 35, two pairs of “immed_w0” and “immed_w1” instructions are modified at run time, during the steps (1) and (2), as shown in Figure 35. The values of input arguments introduced into the microcode are 0x20100 and 0x10500 for the two components, respectively.

Second, the binder modifies the microwords where global variables are used. The “alu” instructions shown in Figure 35 load the absolute registers @var1 and @var2. The absolute registers @var1 and @var2 are global variable registers. An admission controller assigns the addresses of these global variable registers before binding takes place. The binder then replaces the addresses that are initially used by the programmer for these registers, (i.e., 45 and 46 as shown in Figure 35), with a value assigned by the admission controller (47). In this manner, pipelines can use the same GPR for accessing shared information (step 3 in Figure 35).

Third, the binder modifies all branch instructions that are included in each transport module. The destination addresses of branch instructions are incremented by the instruction store addresses where transport modules are placed. Finally, the microwords that correspond to the exit points of transport modules are modified so that the program flow jumps into their associated entry points (step 4 in Figure 35). By modifying the microcode of components at run time, NetBind can create processing pipelines that are optimized adding little overhead in the critical path. This is a key property of the NetBind system that we discuss in the evaluation section.

### 5.3.4 The Admission Control Algorithm

In NetBind, admission control drives the assignment of system resources including registers, memory, instruction store space, headroom and hardware contexts. Resource management in network processors is a difficult problem. The difficulty stems from the fact that network processors use multi-processor architectures that expose many different types of resources to applications (e.g., multiple types of memory units, registers, packet processing units and hardware contexts).

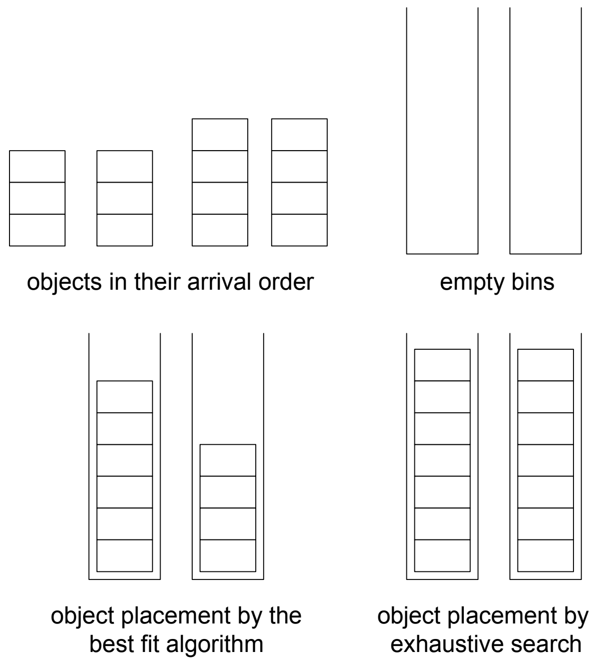

objects in their arrival order
empty bins
object placement by the best fit algorithm
object placement by exhaustive search

**Figure 36: Bin Packing Example**

We have not fully investigated the resource management problem in network processors as yet. Instead, we designed a simple heuristic algorithm for admission control. NetBind applies a “best fit” bin-packing algorithm to determine the most suitable microengines where data path components should be placed. The best fit bin packing algorithm determines the microengine where a packet processing pipeline should be placed by calculating the remaining resources (i.e., leftover) after the placement of the pipeline in each microengine. In order to determine the remaining resources associated with each resource assignment, NetBind calculates a weighted average, taking every type of resource, (i.e., memory and instruction store space, hardware contexts, and global variable registers), into account. Each type of resource weighs equally on the calculation of the leftover resources. The microengine selected is the one that results in the smallest amount of leftover after the placement of the pipeline.

Maintaining records of resource usage is essential to supporting isolation between data paths. If the "best fit" algorithm fails NetBind applies an "exhaustive search" algorithm to determine the microengines where data path components should be placed. Exhaustive search is a technique associated with significant complexity (i.e., O($m$!) as a function of the number of pipelines in the system). However, exhaustive search works effectively for small number of pipelines and can result in efficient resource allocations when the best fit algorithm fails.

Real applications do need exhaustive search. Best fit may fail to find the optimal allocation. In what follows we provide an example that illustrates why exhaustive search is necessary. In the example shown in Figure 36, microengines are abstracted and referred to as 'bins' and pipelines are abstracted and referred to as 'objects' for the sake of simplicity. Let's assume that we have two bins of size seven and two objects which need to be placed into the bins of size three each. The best fit algorithm places the two objects in the first of the bins. If an object of size four needs to be added into the bins as well, the best fit algorithm places the object in the second bin. If a fourth object, also of size four needs to be added into the bins, then the best fit algorithm determines that there is not enough space in the system to accommodate the new object. This happens because the leftover in the first bin is one and in the second bin is three. Exhaustive search in this case determines that there exists a placement configuration which can accommodate all four objects. In the optimal configuration each of the bins contains one object of size four and one object of size three.

Before realizing a data path, NetBind examines whether the candidate data path exceeds the network processor headroom. Each transport module carries an estimate of its worst case execution time. The execution time for each component is determined from deterministic bounds on the time it takes for different microengine instructions to complete. With the exception of memory operations, the execution time of microengine instructions is deterministic and can be obtained using Intel's transactor simulation environment [72].

## 5.4 NetBind Implementation

We have implemented NetBind in C and C++ as a user space process in the StrongARM Core processor of IXP1200 and used it as part of the Genesis Kernel code release for network processors. The

StrongARM Core processor runs a embedded ARM version of Linux. A diagram of the components of the NetBind system is shown in Figure 37. The NetBind binding system consists of:

- a network processor specification converter object, which converts a virtual router specification to a network processor specification. Information about symbol bindings required for the conversion is obtained from a database of transport modules and symbol bindings.
- a data path constructor object, which coordinates the binding process, accepting as input the network processor specification of a data path. The data path constructor parses the transport module (.tmd) files that contain data path components and converts the network processor-related specification of a data path into an IXP1200-related specification.
- an admission controller object, which determines whether the resource requirements of a data path can be met; The admission controller performs resource allocation for every candidate data path. If the resource requirements of a data path can be met, the admission controller assigns a set of microengines, hardware contexts, global variable registers, memory and instruction store regions to the new data path. Once admission control takes place, the data path is created.
- a verifier object, which verifies the addresses and values of each symbol associated with a data path before binding takes place. The verifier object also checks the validity of the resource allocation made by the admission controller. The verifier object is useful for debugging since it makes sure that no incorrect or malicious microcode is written into the instruction stores of IXP units. Using a verifier object one can avoid resetting the system manually every time the data path constructor makes a mistake.
- a binder object, which performs low-level binding functions, such as, the modification of microwords for binding, or, the loading of transport modules into instruction stores. The binder is “plug-and-play” and can either create new data paths, or, modify existing ones at run-time
- a code morphing object, which offers a set of methods that parse IXP1200 microinstructions and modify the fields of these instructions. The code morphing object is used by the binder.
- a database of transport modules and symbol bindings. Transport modules encapsulate component code and are accessed by the data path constructor. Symbol bindings are used during the conversion of the virtual router specification into the network processor specification.

- a microstore object, which can initialize or clear the instruction stores on an IXP1200 network processor. The microstore object can also read from, or write into, any address of the instruction stores.

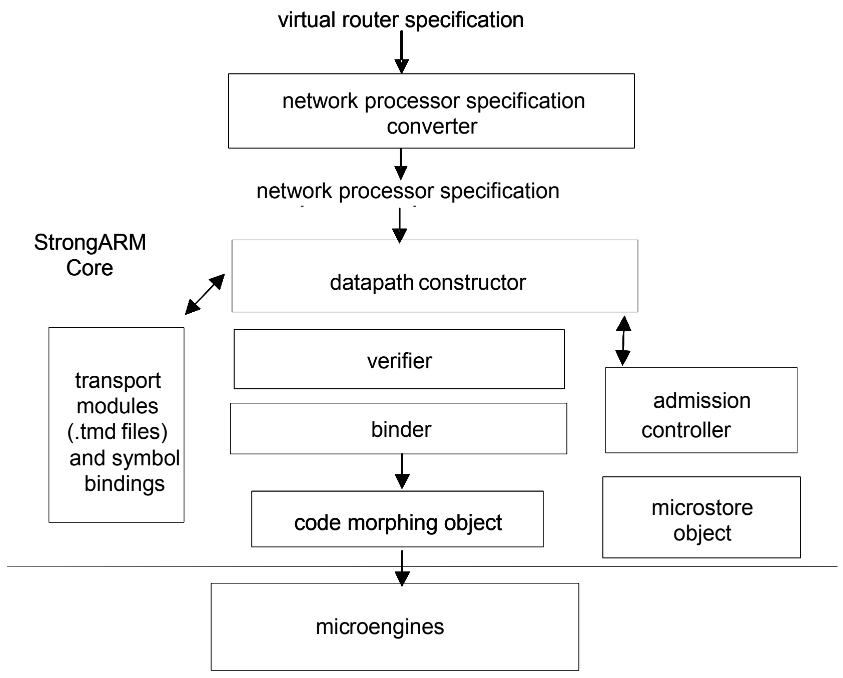

virtual router specification
network processor specification converter
network processor specification
StrongARM Core
datapath constructor
verifier
binder
code morphing object
transport modules (.tmd files) and symbol bindings
admission controller
microstore object
microengines

**Figure 37: The NetBind Binding System**

In order to implement the binder object we had to discover the binary representations for many IXP1200 microassembler instructions. This was not an easy task since the opcodes of microassembler instructions do not have fixed lengths and they are not placed in fixed locations inside each instruction's bit set.

Transport modules can be developed using tools such as the Intel Developer Workbench [72]. The IXP1200 microassembler encapsulates the microcode associated with a component project into a “.uof” file. The UOF file format is an Intel proprietary format. The UOF header includes information about the number of microcode pages associated with a Workbench project, the manner in which the instruction stores are filled and the size of pages associated with a project's microcode. Since we have had no access to the source code of the standard development tools provided by Intel [72], (i.e., the Intel Developer Workbench, the transactor simulation environment and the microassembler), we have created our own microassembler extensions on top of these tools. Our microassembler extensions have been used for creating the transport modules of components.

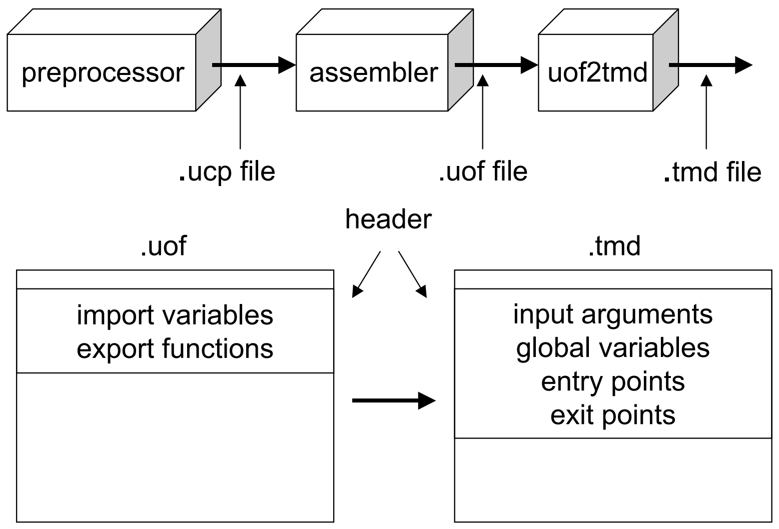


**Figure 38: Microassembler Extensions**

The UOF file format is suitable for encapsulating statically compiled microcode. The UOF file format does not include dynamic binding information in the header. For this reason, we introduced a new file format, the TMD (Transport MoDule) format for encapsulating microcode. Our microassembler extensions are shown in Figure 38. A utility called uof2tmd takes a .uof file as input and produces a ".tmd" file as output. The TMD header includes symbol information about input arguments, global variables entry points and exit points. For each symbol a symbol name, a value and an instruction store address where the symbol is used are provided. Currently, the information included in the TMD header is obtained from the UOF header.

## 5.5 Service Creation Using NetBind

We have used NetBind to create programmable virtual routers that seamlessly share the resources of the communications infrastructure. The dynamic instantiation of a set of routelets across the network hardware, and the formation of virtual networks on-demand are research goals discussed in Chapter 3 and not dealt with in this chapter.

### 5.5.1 IPv4 Data Path

The data path associated with an IPv4 virtual router implemented using NetBind is illustrated in Figure 39. This data path comprises seven transport modules. In the figure, the first six modules run on microengine 0 and are executed by all four contexts by the microengine:

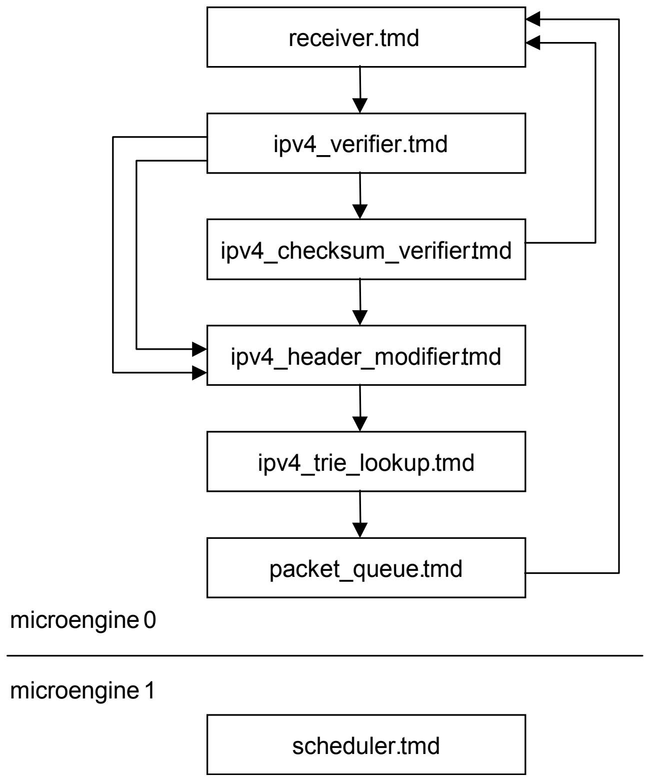


**Figure 39: An IPv4 Data Path**

- a virtual network demultiplexor (receiver.tmd) module receives a packet from the network performs virtual network classification and places the packet into an SDRAM buffer;
- an IPv4 verifier (ipv4_verifier.tmd) module verifies the length, version and TTL fields of an IPv4 header;
- a checksum verifier (ipv4_checksum_verifier.tmd) module verifies the checksum field of an IPv4 header;
- an IPv4 header modifier (ipv4_hdr_modifier.tmd) module decrements the TTL field of an IPv4 packet header and adjusts the checksum accordingly;

- an IPv4 trie lookup (ipv4_trie_lookup.tmd) module performs an IPv4 route lookup; and
- a packet queue (packet_queue.tmd) module dequeues a packet.

In addition, a seventh module runs on a separate microengine (microengine 1). This module is a dynamic scheduler that assigns the transmission of enqueued packets to hardware contexts. Context 0 executes the scheduler, whereas contexts 1, 2 and 3, transmit packets to the network.

```
//declaration of input arguments
decl[symbol, inp_packet_buff_base], 0x20100
decl[symbol, inp_scratchpad_mib], 0x0c0
decl[symbol, inp_scratchpad_pwp], 0x200
decl[symbol, inp_route_base_64k], 0x20000
decl[symbol, inp_route_base_256], 0x30000
decl[symbol, inp_route_base_trie], 0x30100
decl[symbol, inp_ftable_base], 0x8100

//declarations for the packet processing pipeline:
//mpacket receiver:
decl[comp, mpr, mpr, ./receiver.tmd], inp_descriptor_base, inp_packet_buff_base, inp_scratchpad_mib, glo_rdready_sem#,
               glo_rec_sem#, glo_dma_sem#, entry_routerinitialize#, exit_to_verifier#

//ipv4 verifier
decl[comp, ipv4_verifier, ipv4_verifier, ./ipv4_verifier.tmd], inp_descriptor_base, inp_packet_buff_base,
               entry_verificationbegin#, exit_to_checksum#, exit_to_end_verify1#, exit_to_end_verify2#
//ipv4_checksum_verifier
decl[comp, ipv4_checksum_verifier, ipv4_checksum_verifier, ./ipv4_checksum_verifier.tmd], entry_ipv4_cksum_verify#,
               exit_chksum_verify#, exit_to_router#
//ipv4_header_modifier
decl[comp, ipv4_header_modifier, ipv4_header_modifier, ./ipv4_header_modifier.tmd], inp_descriptor_base, inp_packet_buff_base,
               entry_ipv4_header_modifier#, exit_ipv4_header_modifier#
//ipv4_trie_lookup
decl[comp, ipv4_trie_lookup, ipv4_trie_lookup, ./ipv4_trie_lookup.tmd], inp_descriptor_base, inp_packet_buff_base,
               inp_route_base_64k, inp_route_base_256, inp_route_base_trie, inp_ftable_base, entry_routelookup#, exit_routelookup#
//packet_queue
decl[comp, packet_queue, packet_queue, ./packet_queue.tmd], inp_descriptor_base, inp_scratchpad_mib,
               inp_scratchpad_pwp, entry_packet_queue#, exit_packet_queue#

//declaration of bindings for the packet processing pipeline
decl[binding, b0], mpr, exit_to_verifier#, ipv4_verifier, entry_verificationbegin#
decl[binding, b1], ipv4_verifier, exit_to_checksum#, ipv4_checksum_verifier, entry_ipv4_cksum_verify#
decl[binding, b2], ipv4_verifier, exit_to_end_verify1#, ipv4_header_modifier, entry_ipv4_header_modifier#
decl[binding, b3], ipv4_verifier, exit_to_end_verify2#, ipv4_header_modifier, entry_ipv4_header_modifier#
decl[binding, b4], ipv4_checksum_verifier, exit_chksum_verify#, ipv4_header_modifier, entry_ipv4_header_modifier#
decl[binding, b5], ipv4_checksum_verifier, exit_to_router#, mpr, entry_routerinitialize#
decl[binding, b6], ipv4_header_modifier, exit_ipv4_header_modifier#, ipv4_trie_lookup, entry_routelookup#
decl[binding, b7], ipv4_trie_lookup, exit_routelookup#, packet_queue, entry_packet_queue#
decl[binding, b8], packet_queue, exit_packet_queue#, mpr, entry_routerinitialize#

//global variables
decl[gv, var0], mpr, glo_rdready_sem#
decl[gv, var1], mpr, glo_rec_sem#
decl[gv, var2], mpr, glo_dma_sem#

//declaration of the packet processing pipeline that receives and enqueues packets
decl[pline, ipv4_recv, 4], mpr, ipv4_verifier, ipv4_checksum_verifier, ipv4_header_modifier, ipv4_trie_lookup, packet_queue,
               b0, b1, b2, b3, b4, b5, b6, b7, b8, var0, var1, var2

//declaration of the packet transmitting pipeline
decl[comp, scheduler, scheduler, ./scheduler.tmd]
decl[pline, ipv4_xmit, 4], scheduler

//aggregate structures
decl[va, aggr0], ipv4_recv, var0
decl[va, aggr1], ipv4_recv, var1
decl[va, aggr2], ipv4_recv, var2

//declaration of the datapath
decl[dpath, ipv4], ipv4_recv, ipv4_xmit, aggr0, aggr1, aggr2
```

**Figure 40: Network Processor Specification of the Ipv4 Data Path**

The network processor specification for the router presented above is shown in Figure 40. The specification is written in a scripting language called ni. This scripting language has been defined and developed for testing in addition to the XML grammar of the Genesis Kernel. The syntax of ni commands is similar to the syntax of IXP1200 microcode instructions. The first string in a ni command identifies the operation which is executed. A number of operands follow. Operands are declared inside brackets and are separated by commas. The final part of a command consists of one or multiple optional tokens. For a detailed description of the commands and syntax of the ni scripting language see [105].

In the first part of the network processor specification input arguments are declared. Input arguments are declared as symbols. Each symbol is associated with a string name. Optional tokens follow the declaration of input arguments. These tokens specify numerical values assigned to input arguments. For example, the location of a 64K table used in a 16-bit trie lookup is declared as an input argument named 'inp_route_base_64k'. This input argument is used for customizing the IPv4 trie lookup module. The numerical value assigned to this argument is 0x20000, which is an SRAM address location. A different table location could be specified by introducing a different numerical value in the declaration of the symbol inp_route_base_64k.

In the second part of the network processor specification the components that constitute the processing stages and pipelines of the IPv4 data path are declared and their binding symbols specified. Each component is listed using a 'decl' command. Symbols associated with each component are specified as optional tokens. Symbols include input arguments, global variables, entry points and exit points as mentioned earlier. For example the component that performs IPv4 routing lookup (ipv4_trie_lookup) is listed followed by eight symbols as shown in Figure 40. The first symbol (inp_descriptor_base) is an input argument specifying the SRAM location of a list of packet buffer descriptors used by the IPv4 data path.

The second symbol (inp_packet_buff_base) is an input argument specifying an SDRAM location where packet buffers are stored. The next three symbols (i.e., inp_route_base_64k, inp_route_base_256 and inp_route_base_trie) are input arguments specifying SRAM locations of tables constituting a trie data structure. This trie data structure is used for performing a Longest Prefix Matching (LPM) search on the destination IP address of each packet. The next symbol (inp_ftable_base) is an input argument specifying an SDRAM location where next hop information is stored. Finally, the last two symbols (i.e., entry_routelookup# and exit_routelookup#) specify the module's entry and exit points respectively. The entry point symbol (entry_routelookup#) refers to the first instruction of the trie lookup module, where the IPv4 routing lookup process begins. The exit point symbol (exit_routelookup#) refers to a branch instruction executed at the end of the IPv4 routing lookup process, where the sequence of control jumps to another module.

After the declaration of components, bindings are declared. Nine bindings are declared in the example of Figure 40 named b0-b8. The declaration of bindings reflects the sequence of control and inter-module relationship characterizing the IPv4 data path of Figure 41. The execution of the IPv4 data path begins with the execution of a virtual network demultiplexor module (receiver.tmd). After the execution of this module a packet is passed into an IPv4 verifier module (binding b0). If the packet header contains valid version and TTL fields, the sequence of control jumps to the checksum verifier module (binding b1). However, if the packet header does not contain valid header fields, an exception is raised and the sequence of control jumps to the IPv4 header modifier module bypassing the checksum verifier module (bindings b2 and b3). After the checksum verifier module is executed the sequence of control jumps to the IPv4 header modifier module (binding b4). This module decrements the TTL field of the packet header and re-computes the checksum. In case the verification of the checksum field fails an exception is raised and the sequence of control jumps back to the virtual network demultiplexor module (binding b5). Packets with valid checksum fields are passed to the IPv4 header modifier module and then to the IPv4 trie lookup module (binding b6). Once routing lookup takes place, packets are placed in appropriate queues (binding b7) and the sequence of control jumps back to the virtual network demultiplexor (binding b8). Each binding is declared using four tokens. Tokens specify the source component, exit point, destination component and entry point associated with each binding.

Following the declaration of bindings, global variables are declared. Next, packet processing pipelines are declared as collections of components, bindings and global variables. The IPv4 data path described in the specification of Figure 40 consists of two pipelines: An 'ipv4_recv' pipeline which places packets into queues and an 'ipv4_xmit' pipeline which schedules the transmission of packets into the network. In the last statement of the specification the IPv4 data path is declared as a collection of pipelines and global variables.

```
//entry points
.export_func oneway 0123 entry_ipv4_cksum_verify# 0 0
.export_func oneway 0123 entry_ipv4_end_verify# 0 0

//exit points
.export_func oneway 0123 exit_chksum_verify# 0 0
.export_func oneway 0123 exit_to_router# 0 0

//REGISTER ALLOCATIONS:
.areg _exception 3
.areg local_a6 6
.breg local_b8 8
.areg _descriptor_address 0

.$$reg $$xdfer0 0
.$$reg $$xdfer1 1
.$$reg $$xdfer2 2
.$$reg $$xdfer3 3
.$$reg $$xdfer4 4
.$$reg $$xdfer5 5
.$$reg $$xdfer6 6
.$$reg $$xdfer7 7

.xfer_order $$xdfer0 $$xdfer1 $$xdfer2 $$xdfer3 $$xdfer4 $$xdfer5 $$xdfer6 $$xdfer7

entry_ipv4_cksum_verify#:
// Checksum is in the two least significant bytes of
//transfer register 4

alu[local_b8, 0, +16, $$xdfer1]
alu[local_b8, local_b8, +, $$xdfer2]
alu[local_b8, local_b8, +carry, $$xdfer6, >>16]
alu[local_b8, local_b8, +carry, $$xdfer3]
alu[local_b8, local_b8, +carry, $$xdfer4]
alu[local_b8, local_b8, +carry, $$xdfer5]

alu[local_b8, local_b8, +carry, 0]
// add in previous carry
ld_field_w_clr[local_a6, 1100, local_b8]
// get high 16 of the total
alu_shf[local_b8, local_a6, +, local_b8, <<16]
// add low 16 bits to upper 16
alu[local_a6, 1, +carry, local_a6, <<16]
// add last carry +1, local_a6 B op=0
alu[local_b8, local_b8, +, local_a6, <<16]
// add 1<<16 to 0xffff to get zero result

br=0[entry_ipv4_end_verify#]
alu[_exception, --, B, 0x0A] //IP_BAD_CHECKSUM
entry_ipv4_end_verify#:


alu[--, --, B, _exception]
br=0[exit_chksum_verify#]

//else if the packet is not correct
//free the buffer space and start all over
//again

immed_w0[local_b8, 0x0000]
immed_w1[local_b8, 0x0010]

alu_shf[--, local_b8, OR, 0x0000]
// merge ov bit with freelist id/bank

sram[push, --, _descriptor_address, 0, 0], indirect_ref, ctx_swap

exit_to_router#:
br[end#]

nop
exit_chksum_verify#:
br[end#]
end#:
nop
```

**Figure 41: Checksum Verifier Module**

## 5.5.2 Example of a Component

In what follows we describe how one of the components of the IPv4 data path is implemented. We present the checksum verifier module as it is implemented as part of the NetBind source code distribution [105].

The checksum verifier module verifies the checksum field of an IPv4 header. The checksum verifier module consists of twenty three microcode instructions and can be executed by any number of hardware contexts between one and four in a single IXP1200 microengine. The checksum verifier module exports two pipeline registers and can be linked with other modules using two entry points and two exit points. The microcode that implements the checksum verifier module is shown in Figure 41. In the first part of the code, entry and exit points are declared using the '.export_func' directive of the IXP1200 microassembler. Next, pipeline and local registers are declared.

Pipeline register names are distinguished from local register names because pipeline register names begin with the underscore ('_') character. Two pipeline registers are declared and exported as a programming API for the component. The '_descriptor_address' pipeline register holds the address of a packet buffer descriptor. This packet buffer descriptor indicates the location of a buffer where a packet is stored in the SDRAM memory unit. The '_exception' pipeline register holds a flag that indicates errors or failure during the reception of a packet.

After the declaration of pipeline and local registers, memory transfer registers are declared. Register addresses are allocated statically. Eight SDRAM transfer registers are declared in the code (i.e., $$xdfer0 - $$xdfer7). These registers hold the fields of the IPv4 packet header that are examined by the checksum verifier module. The first instruction of the module is marked with the label 'entry_cksum_verify#' indicating that this instruction is the module's entry point. The checksum verify process divides every word in the packet header into two sixteen bit parts and adds the parts of every word producing a sum. The carry resulting from each addition is added into this sum. If the checksum field in the packet header is correct, then the result from all additions is zero. If the value of the checksum field is not correct, then an exception is raised and the buffer space for the packet is de-allocated. The packet is discarded, in this case. The program sequence executes the branch instruction labeled 'exit_to_router#'. This is an exit point leading to the beginning of the packet receiving code. Otherwise, if the value of the checksum field is correct, then program sequence executes the branch instruction labeled 'exit_chksum_verify#', which is an exit point leading to the next component in the packet processing pipeline. Both exit points jump to the same label 'end#' in the microcode of Figure 41. This happens because the component code of Figure 41 reflects the component state before binding. At development time all exit points of components branch to the same instruction which is a 'nop' (i.e., no operation) instruction separating transport modules. During binding the destination addresses of branch instructions are modified. The destination addresses of branch instructions are set to the entry points of the components where the sequence of control jumps to, as explained earlier.

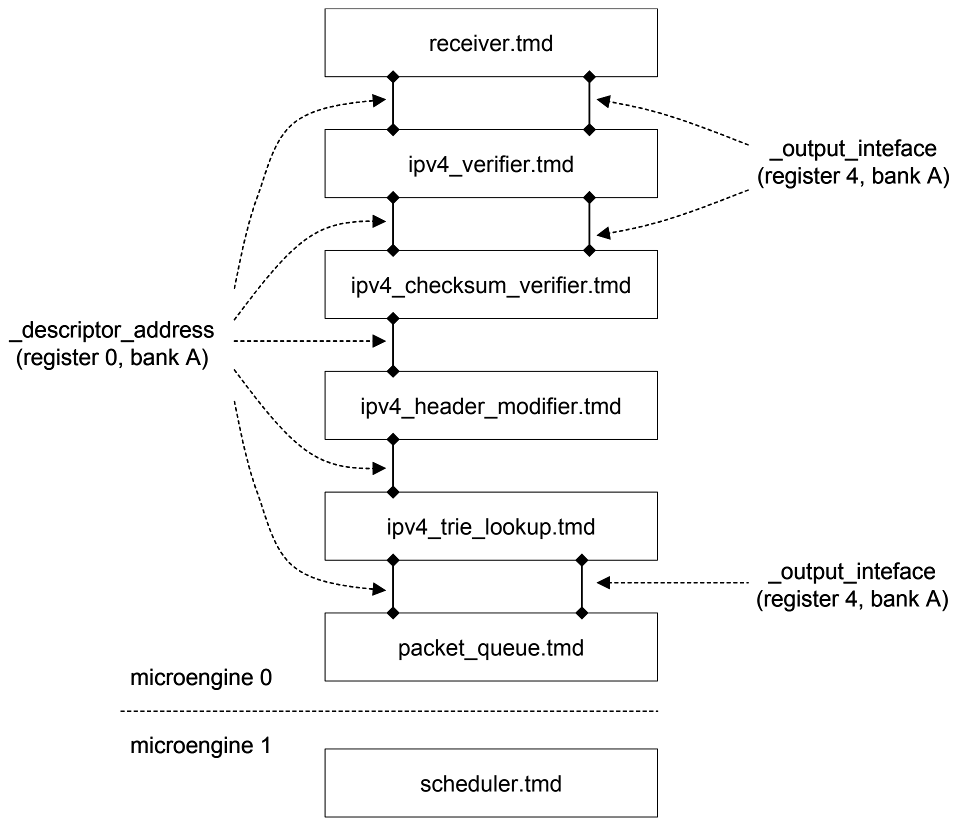


**Figure 42: Pipeline Register Allocations in the Ipv4 Data Path**

## 5.5.3 Hardware Context and Register Allocations

The IPv4 data path of Figure 39 is split into two pipelines as shown earlier. The first pipeline implements the reception of packets from the network. This pipeline is executed by four hardware contexts each serving a different port of the Bridalveil development board (see below). The input FIFO slots of the IXP1200 network processor are evenly divided between the four contexts and assigned to each port in a static manner. On the transmit side, however, the scheduling of contexts is done dynamically. We use three hardware contexts to transmit packets into the network on an on-demand basis and one context to assign packet transmission tasks to the transmit contexts. The same approach has been followed in the IPv4 data path reference design distributed by Intel.

Hardware context management is a difficult problem. There is no single solution that satisfies all types of applications and network processor hardware. Implementing network algorithms in software typically requires sophisticated hardware context assignment and synchronization. For example in a router system where the number of output ports is much larger than the number of hardware contexts available for transmission, static allocation of contexts may not be efficient. In other router systems such as the Bridalveil development boards, static allocation of contexts may result in better performance [128, 129]. Another case where static allocation of contexts is inefficient is packet scheduling. A scheduler may need to select the next eligible packet for transmission from a large number of queues or connections. In this case assigning a single thread for serving each queue may not be possible. In a software implementation of a hierarchical packet scheduler, it is preferable if single-level schedulers are served by different contexts. In this way the dequeuing process at each level of the hierarchy can complete without waiting for packets to be enqueued from previous levels. These contexts need to be assigned dynamically to different levels of the hierarchy.

These examples indicate that it is better if binding systems offer the flexibility to programmers to select hardware context allocation and management policies among a range of choices. We believe that programmers should have the flexibility to select the context allocation and management policies which are more suitable for the algorithms programmed in the data path. In NetBind contexts can be assigned to input ports, output ports and other types of resources (e.g., queues) either statically or dynamically. At the admission control level, the resource requirements of each component are presented as the total number of contexts needed. Port and queue allocations are hidden from the admission controller. The benefit of this approach is that the binding and admission control processes are simplified because port and queue allocations are not taken into account. The disadvantage of this approach is that additional responsibility is placed on the component developer which needs to know details about the way ports and queues operate in different router systems. In addition, a strong trust relationship is assumed between programmers at different levels of the specification hierarchy of Figure 33. The programmers at the virtual router specification level assume that component bindings are successfully resolved at the network specification level. Similarly, programmers at the network processor specification level assume that bindings are successfully resolved at the IXP1200 specification level and that components manage hardware contexts, ports and queues efficiently.

While port and queue allocations are hidden from the binder and admission controller, register allocations are exposed. Pipeline and input argument registers are assigned statically as discussed earlier, while global variable registers are assigned dynamically. In Figure 42 we illustrate the path which a packet follows through the router system of Figure 39, and the pipeline registers used in this path.

## 5.6 Evaluation

### 5.6.1 Experimental Environment

#### Hardware Environment

To evaluate NetBind, we have set up an experimental environment, as part of our spawning testbed, consisting of three "Bridalveil" development boards, interconnecting desktop and notebook computers in our lab. Bridalveil [69] is an IXP1200 network processor development board running at 200 MHz and using 256 Mbytes of SDRAM off-chip memory. The Bridalveil board is a PCI card that can be connected to a PC running Linux. The host processor of the PC and the IXP1200 unit can communicate over the PCI bus. The PC serves as a file server and boot server for the Bridalveil system. PCI bus network drivers are provided for the PC and for the embedded ARM version of Linux running on the StrongARM Core processor. In our experimental environment, each Bridalveil card is plugged into a different PC. In each card, the IXP1200 chip is connected to four fast Ethernet ports that can receive or transmit packets at 100 Mbps.

**Software Environment**

Initially NetBind was developed for an IXP1200 Ethernet evaluation board running at 166 MHz. To evaluate NetBind we ported the NetBind code to the Bridalveil system where we could also execute MicroACE. MicroACE is a systems architecture provided by Intel for composing modular data paths and network services in network processors. The MicroACE adopts a static binding approach to the development of modular data paths based on the insertion of a "dispatch loop" code stub in the critical path. The dispatch loop is provided by the programmer and directs the program flow through the components of a programmable processing pipeline. MicroACE is a complex system that can be used for programming microengines and the StrongARM Core. In what follows we provide an overview of MicroACE.

**MicroACE overview**

MicroACE is an extension of the ACE [71] framework that can execute on the microengines of IXP1200. In MicroACE, an application defines the flow of packets among software components called "ACEs" by binding packet destinations called "targets" to other ACEs in a processing pipeline. A series of concatenated ACEs form a processing pipeline. A MicroACE consists of a "microblock" and a "core component". A microblock is a microcode component running in the microengines of IXP1200. One or more microblocks can be combined into a single microengine image called "microblock group". A core component runs on the StrongARM Core processor. Each core component is the control plane counterpart of a microblock running in the microengines. The core component handles exception, control and management packets.

**Binding in MicroACE**

The binding between microblocks in the ACE framework is static and takes place offline. The targets of a microblock are fixed and cannot be changed at run time. For each microengine, a microcode "dispatch loop" is provided by the programmer, which initializes a microcode group and a pipeline graph. The size of a microblock group is limited by the size of the instruction store where the microcode group is placed. Before a microblock is executed, some global binding state needs to be examined. The global binding state consists of two variables. A "dl_next_block" variable holds an integer identifying the next block to be executed, whereas a “dl_buffer_handle” variable stores a pointer to a packet buffer exchanged between components. Specialized “source” and “sink” microblocks can send or receive packets to/from the network. Since the binding between microblocks takes place offline, a linker can preserve the persistency of register assignments

## 5.6.2 Dynamic Binding Analysis

**Qualitative Analysis**

MicroACE and NetBind have some similarities in their design and realization but also differences. MicroACE offers much higher programming flexibility allowing the programmer to construct processing pipelines in the StrongARM Core and the microengines. NetBind, on the other hand offers a set of libraries for the microengines only. MicroACE supports static linking allowing programmers to use registers according to their own preferences. MicroACE does not impose any constraints on the number and purpose of registers used by components. NetBind on the other hand supports dynamic binding between components that can take place at run time. To support dynamic binding NetBind imposes a number of constraints on the purpose and number of registers used by components, as discussed in Section 5.3.

The main difference between NetBind and MicroACE in terms of performance comes from the choice of binding technique. MicroACE follows a dispatch loop approach where some global binding state needs to be checked before each component is executed. In NetBind there is no explicit maintenance of global binding state. Components are associated with each other at run time through the modification of their microcode. The modification of microcode at run time, which is fundamental to our approach, poses a number of security challenges, which our research has not yet addressed. Another problem with realizing dynamic binding in the IXP1200 network processor is that microengines need to temporarily terminate their execution when data paths are created or modified. Such termination may disrupt the operation of other data paths potentially sharing the resources of the same processing units. We are investigating methods to seamlessly accomplish dynamic binding when multiple data paths share the resources of the same network processor as part of our future work.

**Headroom Analysis**

To compare NetBind against MicroACE we estimate the available headroom for the microengines of the IXP1200 network processor. In what follows, we discuss a methodology on how headroom can be calculated in any multi-threaded network processor architecture. We assume that each port of a network processor can forward packets at line speed *C*, and that a minimum size packet is *m* bits. The maximum time a packet is allowed to spend on the network processor hardware without violating the line rate forwarding requirement is:

$$T = \frac{m}{C} \qquad \text{[Eq. 1]}$$

During this time microengine resources are shared between *n* hardware contexts. It is fair to assume that each thread gets an equal share of the microengine resources on average, and that each packet is processed by a single thread only. Therefore, the maximum time a thread is allowed to spend processing a packet without violating the line rate-forwarding requirement is:

$$\frac{T}{n} = \frac{m}{n \cdot C} \qquad \text{[Eq. 2]}$$

Typically, microengine resources do not remain utilized all the time. For example, there may be cases when hardware contexts attempt to access memory units at the same time. In these cases, all contexts remain idle until a memory transfer operation completes. Therefore, we need to multiply the maximum time calculated above with a utilization factor $\rho$ in order to have a good estimation of the network processor headroom. A final expression for the network processor headroom *H* is given below, expressed in microengine cycles, where $t_c$ is the duration of each cycle:

$$H = \frac{\rho \cdot m}{n \cdot C \cdot t_c} \qquad \text{[Eq. 3]}$$

The utilization factor was measured when our system was running the IPv4 data path discussed in Section 5.3, and was found to be equal to 0.98. We doubled the percentage of idle time to have a worst case estimation of the utilization factor for many different types of data paths, resulting in $\rho = 0.96$. After substituting $C = 100$ Mbps, $m = 64$ bytes, $n = 4$, and $t_c = 5$ ns into Eq. 3, we find that the headroom *H* in the IXP1200 Bridalveil system is 246 microengine cycles.

**Binding Overhead Analysis**

We evaluated the performance of the IPv4 data path discussed in Section 5.3 when no binding is performed (i.e., the data path is monolithic) and when the data path is created using NetBind and MicroACE. We also implemented a simple binding tool based on the vector table binding technique, as discussed in Section 5.2. We used this tool in our experiments in order to compare the three binding techniques (presented in Section 5.2) in a quantitative manner.

| binding technique | per component binding instructions | | | |
|---|---|---|---|---|
| | register operation | conditional branch | unconditional branch | memory (scratch) transfer |
| NetBind | 1 | 1 | N/A | N/A |
| dispatch loop (MicroACE) | 2 | 1 | 2 | N/A |
| vector table | N/A | N/A | 1 | 1 |

**Table 4: Binding Instructions in the Data Path**

| module name | size | input arg. | global var. | entry points | exit points |
|---|---|---|---|---|---|
| receiver (initialization) | 48 | N/A | N/A | N/A | N/A |
| receiver | 100 | 3 | 3 | 3 | 3 |
| ipv4_verfier | 17 | 2 | 0 | 1 | 3 |
| ipv4_checksum_verifier | 16 | 0 | 0 | 2 | 1 |
| Ipv4_header_verifier | 23 | 2 | 0 | 1 | 1 |
| ipv4_trie_lookup | 104 | 6 | 0 | 1 | 1 |
| packet_queue | 44 | 3 | 0 | 1 | 1 |
| aggregate (excluding initialization) | 304 | 16 | 3 | 9 | 10 |

**Table 5: Component Sizes and Symbols for the IPv4 Data Path**

To analyze and compare the performance of different data paths, we executed these data paths on Intel's "transactor" simulation environment and on the IXP1200 Bridalveil cards. Table 4 shows the additional instructions that are inserted in the data path for each binding technique. Our implementation of the IPv4 data path is broken down into two processing pipelines: a "receiver" pipeline consisting of 6 components (from "receiver" to "packet_queue" in Table 2) and a "transmitter" pipeline consisting of single component ("scheduler.tmd" in Figure 39). The receiver and transmitter pipelines are executed by microengines 0 and 1, respectively. Table 5 shows the number of instructions, input arguments, global variables, entry points, and exit points for each component of the receiver pipeline.

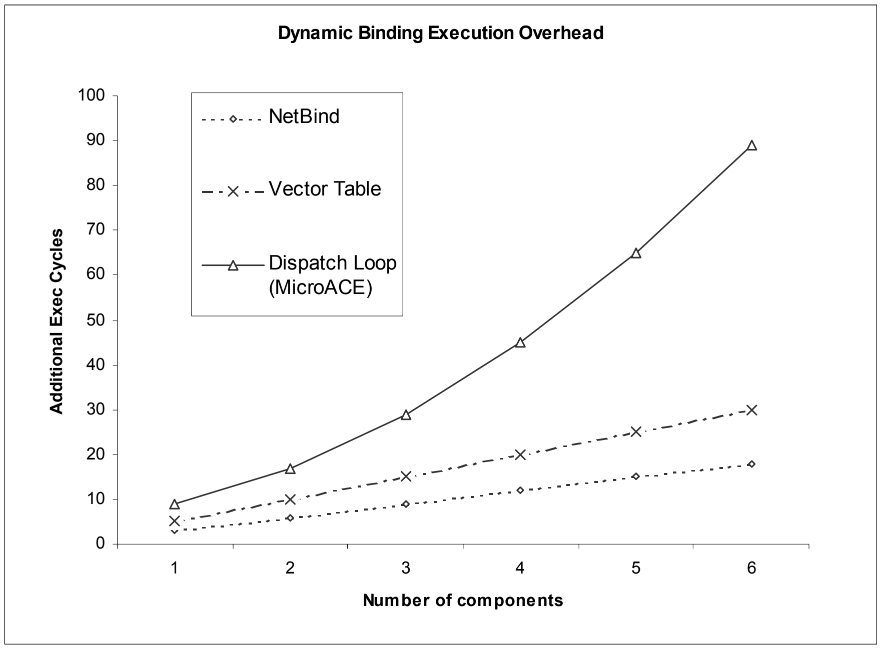


**Figure 43: Dynamic Binding Overhead**

We added a "worst case" generic dispatch loop to evaluate the MicroACE data path. The dispatch loop included a set of comparisons that determine the next module to be executed on the basis of the value of the "dl_next-block" global variable. The loop is repeated for each component. We added 5 instructions for each component in the dispatch loop: (i) an "alu" instruction to set the "dl_next_block" variable to an appropriate value; (ii) an unconditional branch to jump to the beginning of the dispatch loop; (iii) an "alu" instruction to check the value of the "dl_next_block" variable; and (iv) a conditional and an unconditional branch to jump to the appropriate next module in the processing pipeline.

The vector table binding technique works as follows. At the initialization stage, we retrieve the entry point locations using "load_addr" instructions. The vector is then saved into the 4K scratch memory of IXP1200. For each component, we insert a scratch "read" instruction to retrieve the entry point from the vector table and a "rtn" instruction to jump to the entry point of the next module. The performance of the vector table scheme is heavily dependent on the speed of a memory access. If the vector for the next component in the pipeline can be retrieved in advance, the overhead of binding can be reduced drastically to 5 cycles per binding.

The advantage of having a multithreaded network processor is that memory access latencies can be hidden if the processor switches context when performing time consuming memory transfer operations.

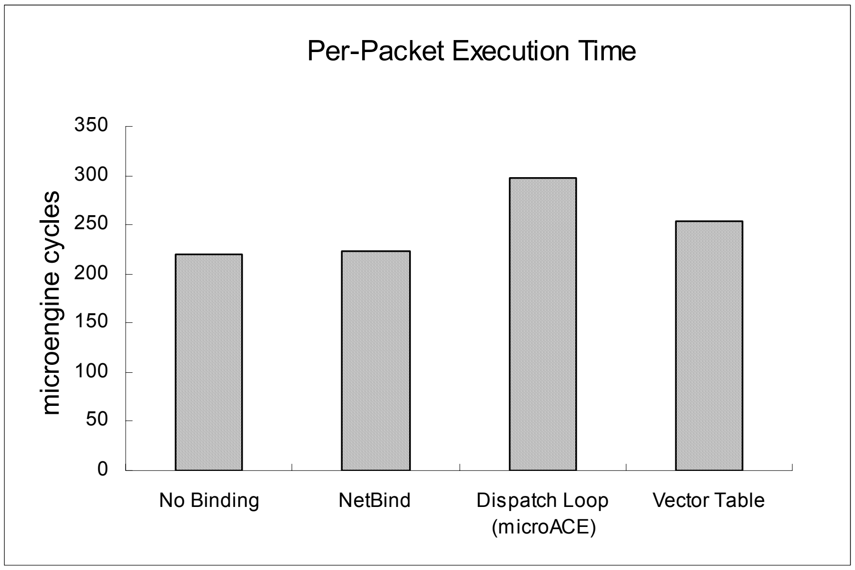


**Figure 44: Per-Packet Execution Time**

Figure 43 shows additional execution cycles for each binding technique. From the figure, we observe that the dispatch loop binding technique, used by MicroACE, introduces the largest overhead, while the NetBind code morphing technique and the vector table technique demonstrate smaller overhead. The overhead of the worst case dispatch loop for a six component data path is 89 machine cycles which represents 36% of the network processor headroom. NetBind demonstrates the best performance in terms of binding overhead adding only 18 execution cycles when connecting six modules to construct the IPv4 data path.

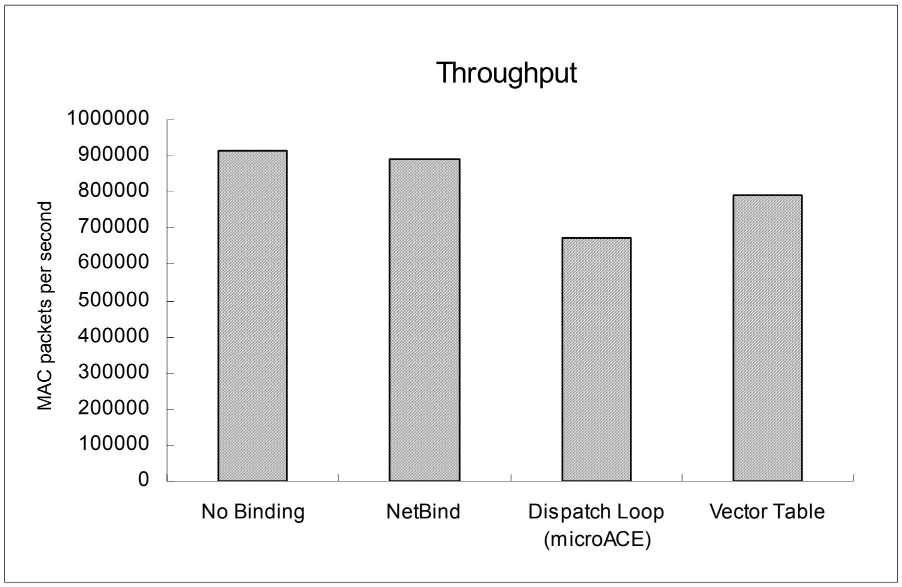


**Figure 45: Packet Processing Throughput**

Figure 43 illustrates that the vector table technique isn't that much worse than NetBind in the IXP1200 network processor. However memory latencies are much larger in network processors that target higher speeds than fast Ethernet. In commercial network processors that target OC-48 and OC-192 speeds memory access latencies can be several times larger than the network processor headroom, as stated earlier. For example memory access latencies can be as large as 100-300 compute cycles in the IXP2400 and IXP2800 network processors. In these processors the headroom for a single microengine is 57 compute cycles. The vector table technique requires at least two times the processing headroom for fetching a single component in these network processors. For this reason it is not easy to support modular line rate forwarding with the vector table technique. Memory access latencies are likely to be even larger in next generation network processors targeting faster line rates (e.g., OC-768). The NetBind approach is suitable for this type of processors because it adds a binding overhead of two compute cycles per component which is insignificant.

Figures 44 and 45 show per-packet execution times and packet processing throughput for the three binding techniques, and a monolithic data path. The term 'MAC packets' used in Figure 45, refers to minimum size Ethernet packets of 64 bytes each. The measurements were taken for the case where the data path is split between 6 components, and packets are forwarded at the maximum possible rate from four input ports to the same output port. To forward packets at the maximum possible rate independent of the input port speed, we applied the technique suggested in [128, 129], where iterations of the input forwarding process forward the same packet from the input FIFO slot avoiding port interaction. NetBind has 2% overhead (execution time) over the monolithic implementation. The vector table and dispatch loop implementation have 12% and 32% overhead, respectively, over the performance of the pipeline created by NetBind. Collecting measurements on an IXP1200 system is more difficult than we initially thought. Since microengines do not have access to an onboard timer, measurements have to be collected by a user program running on the StrongARM core processor. We created a program consisting of a timing loop that is repeated for accuracy. The program stops executing and prints out the time difference measured after a significant number of iterations have taken place.

### 5.6.3 Binding Latency Analysis

We have measured the time to install a new data path using NetBind. Stopping and starting the microengines takes 60 μs and 200 μs respectively. The binding algorithm and the process of writing data path components into instruction stores takes 400 μs to complete. Measurements were taken using the Bridalveil's 200MHz StrongArm core processor.

Loading six modules from a remotely mounted NFS server takes about 60ms to complete. Since the IXP1200 network processor prevents access to its instruction stores while the hardware contexts are running, we had to stop the microengines before binding, and restart them again after the binding was complete. We are currently studying better techniques for dynamic placement without disruption to executing data paths.

### 5.6.4 NetBind Limitations

NetBind offers significant improvement in terms of data path performance and composition time over the alternative choices of dispatch loops and indirection. However, NetBind imposes a number of constraints on the programmer. The most significant limitation of NetBind is that the register APIs exported by consecutive components in a packet processing pipeline need to match with each other. The physical register addresses exported by a component need to be the same as the physical register addresses exported by every subsequent component in packet processing pipelines. This is a significant restriction. For example a routing lookup component needs to place the identifier to an outgoing interface into the register where every subsequent component (e.g., a packet queue) expects to find this value.

While this seams to be a significant restriction, we believe that NetBind allows programmers to effectively write code for a small but significant class of data paths applications. NetBind is most efficient for applications that manipulate packet header fields, such as packet classification, forwarding and traffic management. The number of parameters that need to be exchanged between components for this class of applications is typically small and well defined. For example, parameters exchanged between components may include packet descriptors, interface identifiers, classifier actions, packet time stamps or eligibility times. In order to effectively write code using NetBind component developers need to determine the set of components which interact with their own data path components. Next, component developers need to make sure that the register APIs which their components export match with the register APIs of subsequent components in packet processing pipelines. This may not be difficult if the number of parameters exchanged between components is limited and well defined.

Another limitation of the NetBind approach is that a strong trust relationship is assumed between developers at different levels of the data path specification hierarchy. As mentioned earlier, the programmers at the virtual router specification level assume that component bindings are successfully resolved at the network specification level. Similarly, programmers at the network processor specification level assume that bindings are successfully resolved at the IXP1200 specification level and that components manage hardware contexts, ports and queues effectively.

The design of the NetBind tool is independent of the target line rate. This is because the binding system can construct data paths from many different packet receiving and transmitting components designed for different network interfaces and speeds. Port and queue allocations are hidden from the binder and admission controller. While this practice places extra burden on the programmer it simplifies the binding process and makes the system applicable to a range of network processors. We have not investigated forwarding at Gigabit speeds (e.g., OC-48, OC-192). However, there is strong indication that

NetBind can be applied to these line rates due to the simplicity and generality of the binding process. Porting NetBind to network processors that target the OC-48 and OC-192 line rates is left for future work.

## 5.7 Related Work

Programmable routers represent an active area of research. Click [79] is an extensible architecture for creating software routers in commodity operating systems. A Click router is constructed by selecting components called "elements" and connecting them into directed graphs. Click components are implemented using C++ and, thus, inherit the binding overhead associated with using a higher level programming language to construct data paths. The software [148] and hardware [133] plugins projects are investigating extensibility in programmable gigabit routers. These routers are equipped with port processors, allowing for the insertion of software/hardware components, where, hardware plugins are implemented as reconfigurable logic. The work on Scout OS [100] is addressing the problem of creating high performance data paths using general-purpose processors. The installation of packet forwarders in network processors has been discussed in [129]. Packet forwarders discussed in [129] are rather monolithic in nature and their installation system does not support binding.

Run-time machine language code generation and modification has been proposed as part of the work on the Synthesis Kernel [117]. The Synthesis Kernel aims for improving kernel routine performance in general-purpose processors as well.

## 5.8 Summary

We have investigated solutions to the problem of engineering network programming systems so that they can perform service composition with the least possible overhead. In particular, we presented the design, implementation and evaluation of the NetBind software system, and compared its performance to the Intel MicroACE system, evaluating the binding overhead associated with each approach. While the community has investigated techniques for synthesizing kernel code and constructing modular data paths and services, the majority of the literature has been focused on the use of general-purpose processor architectures. Little work has been done using network processors. Our work on NetBind aims to address this gap. We proposed a binding technique that is optimized for network processor-based architectures, minimizing the binding overhead in the critical path, and, allowing network processors to forward minimum size packets at line rates. NetBind aims to balance the flexibility of network programmability against the need for high performance. We think this a unique part of our contribution. The NetBind source code, described and evaluated in this chapter, is freely available on the Web (comet.columbia.edu/genesis/netbind) for experimentation.

# Chapter 6

# Packet Classification in Programmable Routers

## 6.1 Introduction

Network programmability requires that a major part of network algorithms is implemented in software. Software-based network algorithms usually need to maintain and to navigate through search data structures. Unfortunately, the overhead of navigating through search data structures can often exceed the time and space budget enforced by router systems. Thus, a key challenge is to design network algorithms that impose low memory space and time overhead. In this thesis we focus on the design of data path algorithms because these algorithms operate on the fastest time scale and, as a result, they are associated with smaller time budgets than control and management plane algorithms. Typically a packet data path comprises algorithms for classification, forwarding, and traffic management. While forwarding end traffic management have been investigated in the past [14, 47, 60, 66, 119, 132, 153], we still lack a good solution for packet classification. The classification problem is challenging, particularly in IP networks because forwarding decisions are made based on the values of several different header fields (i.e., source and destination IP addresses, port number of protocol fields) and because classification rules are associated with arbitrary priority levels.

Packet classification involves identifying flows from among a stream of packets that arrive at routers. It is a fundamental building block that enables routers to support access control, Quality of Service differentiation, virtual private networks, and other value added services. To be classified as belonging to a flow, each packet arriving at a router is compared against a set of rules. Each rule contains one or more fields and their associated values, a priority, and an action. The fields generally correspond to specific portions of the TCP/IP header—such as the source and destination IP addresses, port numbers, and protocol identifier. A packet is said to match a rule if it matches every field in that rule. On identifying the matching rules, actions associated with the rules are executed.

Packet classification is often the first packet processing step in routers. It requires network systems to maintain and to navigate through search data structures. Since flows can be identified only after the classification step, to prevent performance interference across flows, network systems must ensure that classification operates at line speeds. Unfortunately, the overhead of navigating through search data structures can often exceed the time budget enforced by the line-speed processing requirement. Thus, a key challenge is to design packet classification algorithms that impose low memory space and access overhead and hence can scale to high bandwidth networks and large databases of classification rules.

In this chapter, we take a step in the direction of designing such efficient classification algorithms. In particular, we study the properties of packet classification rules; our intent is to expose characteristics that can be exploited to design packet classifiers that can scale well with link bandwidths and the sizes of classification rule databases. Since access control is the most common application of packet classification today, we study four databases of classification rules collected from firewalls supported by large ISPs and corporate intranets. Our analysis yields the following key observations:

1. The fields contained in each rule in firewall databases can be partitioned into two logical entities: (1) source and destination IP address pairs that characterize distinct network paths, and (2) a set of transport-level fields (e.g., port numbers, protocol identifier, etc.) that characterize network applications. In most cases, the number of distinct network paths far exceeds the number of network applications.
2. The IP address pairs define regions in the two-dimensional space that can overlap with each other. However, the number of overlaps is significantly smaller than the theoretical upper-bound.

3. Many source-destination IP address pairs share the same set of transport-level fields. Hence, only a small number of transport-level fields are sufficient to characterize databases of different sizes.

We justify these observations based on standard network administration practices; and thereby argue that these findings, although derived from a small number of databases, are likely to hold for most firewall databases. Based on these findings, we provide the following guidelines for designing efficient classification algorithms.

1. The multi-dimensional classification problem should be split into two sub-problems (or two stages): (1) finding a 2-dimensional match based on source and destination IP addresses contained in the packet; and (2) finding a ($n$-2) dimensional match based on transport-level fields. Whereas the first stage only involves prefix matching, the second stage involves the more general range matching.
2. Because of the overlap between IP address filters maintained in a database, each packet may match multiple filters. Identifying all the matching filters is complex. Since the total number of overlaps observed in firewall databases is significantly smaller than the theoretical upper-bound, a design that maintains all of the intersection filters and returns exactly a single filter is both feasible an desirable.
3. Since each IP address filter is associated with multiple transport-level fields, identifying the highest priority rule that matches a packet requires searching through all the transport-level fields associated with the matching IP filter. Since the number of transport-level fields associated with most databases is rather small, it is possible to rely upon a small, special-purpose hardware unit (e.g., a TCAM unit) to perform the (n-2) dimensional searches in parallel.

The chapter is structured as follows. In Section 6.2, we formulate the classification problem and discuss our methodology for studying ACLs. We discuss our findings in Sections 6.3 and 6.4, and expose the implications of our findings in Section 6.5. Finally, Section 6.6 summarizes our contributions.

## 6.2 Problem Formulation

Since access control is the most common application of packet classification today, we focus on the problem of packet classification in firewalls. In a firewall rule database, each rule contains one or more fields and their associated values, a priority, and an action. The fields generally correspond to specific portions of the TCP/IP header—such as the source and destination IP addresses, port numbers, and protocol identifier. Because of the hierarchical nature of IP address allocation, source and destination IP addresses are often specified as prefixes. To accommodate a collection of user or network management applications, port numbers are often specified as ranges. Finally, other protocol attributes, such as the protocol identifier, are specified as exact values. Table 6 shows some examples of classification rules.

| src. IP address | dest. IP address | src. Port | dest port | action | priority |
|---|---|---|---|---|---|
| 128.59.67.100 | 128.* | * | 15 | drop | 2 |
| 128.* | 128.2.3.1 | * | 24 | DSCP 2 | 1 |

**Table 6: Examples of Classification Rules**

The first rule indicates that packets originating from the IP address 128.59.67.100, and destined to any host within the IP address domain beginning with 128 and port number 15 should be dropped. The priority level for this rule is 2. The second rule states that packets originating from any host in the domain beginning with 128, and destined to the host 128.2.3.1 and port number 24 should be forwarded with the Differentiated Services Code Point (DSCP) set to 2. This rule has priority level of 1.

In this context, the packet classification problem can be stated as follows: Given a set—often referred to as an Access Control List (ACL)—of access control rules, determine the action A associated with the highest priority rule that matches packet p. To reduce the overhead of identifying rules that may match each packet, most packet classification algorithms employ search data structures for organizing classification rules. These data structures occupy memory space. Furthermore, navigating on these data structures incurs several memory accesses. In what follows, we first discuss several existing packet classification algorithms and argue that they do not scale well with increase in network bandwidth or ACL sizes. We then argue that understanding the structure and properties of ACLs is crucial in designing efficient, scalable algorithms. Finally, we describe our methodology for studying the properties of ACLs.

### 6.2.1 State of the Art

Existing packet classification algorithms [9, 10, 25, 53, 86, 116, 130, 131, 139, 161-163] can be grouped into four classes: trie-based algorithms, hash-based algorithms, parallel search algorithms, and heuristic algorithms. Throughout this discussion, we use *n* to denote the number of rules in a classification database, *k* to denote the number of fields (i.e., dimensions), and *w* to denote the maximum length of the fields (in bits).

1. Trie-based Algorithms: Trie-based algorithms [10, 25, 131, 139] build hierarchical radix tree structures where once a match is found in one dimension a search is performed in a separate tree linked into the node representing the match. Examples of such algorithms are the Grid-of-tries [131] and Area-based Quad Tree (AQT) [25] algorithms. Trie-based algorithms require, in worst case, as many memory accesses as the number of bits in the fields used for classification. Multi-bit trie data structures are more efficient from the perspective of the number of memory accesses required. However, these data structures incur significantly higher memory space overhead. In general, trie-based schemes work well for single-dimensional searches. However, the memory requirement of these schemes increases significantly with increase in the number of search dimensions.
2. Hash-based Algorithms: Hash-based algorithms [130] group rules according to the lengths of the prefixes specified in different fields. The groups formed in this manner are called 'tuples'. Hash-based algorithms perform a series of hash lookups one for each tuple to identify the highest priority matching rule. Tuple space search has $O(n)$ storage and time complexity. Hash-based algorithms, in the worst case, require as many memory accesses as the number of hash tables, and the number of hash tables can be as large as the number of rules in a database. As a result, hash-based techniques do not scale well with the number of rules. An optimized hashing technique, referred to as rectangle search [130], reduces the lookup time complexity from $O(n)$ to $O(w)$ in two dimensions. However, to support lookups in more than two dimensions, the algorithm still requires a significant number of memory accesses. A lower bound on the complexity of rectangle search is discussed in [130]. It is proven than tuple probes can be at least $w(k-1)/k!$
3. Parallel Search Algorithms: These algorithms formulate the classification problem as an n-dimensional matching problem and search each dimension separately. In some algorithms [9, 86], when a match is

found in a dimension, a bit vector is returned identifying the matches. The logical AND of the bit vectors returned from all dimensions identifies the matching rules. Such bit-vector techniques are associated with $O(n)$ memory accesses in the lookup process. Fetching a single bit vector or an aggregate bit vector (as described in [9]) can be memory access intensive, especially in cases where the ACL contains more than a few thousand rules. Another parallel search technique called Cross-Producting Table [131] reduces the lookup time complexity to ($O(kw)$) where k is the number of fields and w is maximum length of the fields. However, this technique increases the worst case storage complexity to ($O(n^k)$) making it impractical.

4. Heuristic Algorithms: A fourth category of algorithms includes heuristic algorithms that exploit the structure and redundancy in the rule set [61, 62]. The algorithms proposed to-date are associated with very low lookup time complexity ($O(k)$); however, they impose significant memory space requirements ($O(n^k)$). Hence, these algorithms are suitable for single- or two-dimensional searches, but their space requirement makes them unsuited for the more common five-dimensional searches.
5. From the above discussion, it is apparent that exploiting the structure and properties of ACLs is a promising direction for designing packet classification algorithms that can scale well with link bandwidth and ACL sizes. Unfortunately, the literature contains no detailed studies of ACL properties. This is in-part because ISPs and enterprises, for privacy and security reasons, protect access to their rule databases. Recently, we have obtained access to four firewall databases from ISPs and corporate intranets. Hence, in this chapter, we conduct a careful study to expose the structure and properties of these ACLs, and postulate how these properties can be used to design efficient classification algorithms. The design of specific packet classification algorithms, however, is beyond the scope of this chapter.

### 6.2.2 Experimental Methodology

We analyze four firewall databases; three of these databases are from large ISPs, whereas one is from a corporate intranet. Table 7 summarizes the basic statistics of these ACLs.

As Table 7 indicates, the ISP ACLs are generally much larger than those of the enterprise intranets. Further, it shows that the fields specified in ACLs can be partitioned into two logical entities: (1) source and destination IP address pairs that characterize distinct network paths represented in ACLs, and (2) a set of transport level fields (e.g., port numbers, protocol identifier, etc.) that characterize network applications. In most cases, the number of distinct network paths far exceeds the number of network applications represented in the ACLs.

In what follows, we first analyze IP address pairs and then study the characteristics of transport-level fields. We justify our findings based on standard practices for creating ACLs used by network administrators. Hence, we argue that although our observations are derived from a small number of rule databases, our conclusions are likely to be valid across a large number of such rule databases.

| | type | number of rules | number of unique source/destination IP address fields | protocol types | unique port number fields |
|---|---|---|---|---|---|
| ACL1 | ISP | 754 | 426 | 4 | 140 |
| ACL2 | ISP | 607 | 527 | 5 | 30 |
| ACL3 | ISP | 2399 | 1588 | 5 | 192 |
| ACL4 | Intranet | 157 | 98 | 4 | 36 |

**Table 7: Summary of ACLs**

## 6.3 IP Prefix Pair Analysis

Each rule in an ACL contains a specification of source and destination IP address pairs (also referred to as IP address filters). These addresses are specified as wildcards, prefixes, or exact values. Based on these specifications, the filters represent rectangles, lines or points in the two-dimensional IP address space. Further, the filters may overlap with each other. In what follows, we first conduct a structural analysis of the filters; this allows us to characterize ACLs as a composition of different types of filters (i.e., filters that represent a different shape in the two-dimensional space). We find that only a small number of filters contain wildcards in the source or the destination dimensions in the ISP ACLs. Further, for most filters that do not contain any wildcards, the destination field contains complete IP addresses (representing individual hosts), while the source field contains prefixes (representing IP address domains). Second, we analyze the overlaps among the filters. This allows us to characterize the number of filters that may match a packet, as well as the overhead of maintaining in the ACL a unique filter representing each of the overlaps such that the maximally matching filter can be uniquely identified for each packet. We find that overlaps are created mostly by filters that contain a wildcard in their source or destination fields. Since only a small number of filters contain wildcards, the actual number of overlaps observed in ACLs is significantly smaller than the theoretical upper bound.

| | partially-specified filters | fully-specified filters | total number of filters |
|---|---|---|---|
| ACL1 | 4 (1%) | 421 (99%) | 426 |
| ACL2 | 68 (13%) | 458 (87%) | 527 |
| ACL3 | 160 (10%) | 1427 (90%) | 1588 |
| ACL4 | 83 (86%) | 14 (14%) | 98 |

**Table 8: Partially- and Fully-Specified Filters**

| | wildcard in source address | wildcard in destination address |
|---|---|---|
| ACL1 | 2 | 2 |
| ACL2 | 36 | 32 |
| ACL3 | 112 | 48 |
| ACL4 | 12 | 71 |

**Table 9: Breakdown of Partially-Specified Filters**

### 6.3.1 Structural Analysis

The source-destination IP address pairs can be classified into two types: Partially-specified filters and fully-specified filters. Partially-specified filters contain at least one wildcard (*) in the source or in the destination IP address dimension; these filters capture traffic sent to/from designated servers or subnets of ISP networks. Fully-specified filters, on the other hand, contain an IP address prefix in both the source and destination IP address dimensions. These filters identify the traffic exchanged between specific IP address domains of ISP networks. In most cases, the traffic handled by fully-specified filters is exchanged between important servers (e.g., web, e-mail, NTP, or streaming servers) and clients.

Each IP address filter can be represented geometrically as a point, a line, or a rectangle in a two dimensional IP address space. Whereas partially-specified filters of the form (*,*) cover the entire two dimensional address space, filters of the form (x, *) and (*, y) can be represented either as a line or a rectangle in the 2-D space depending on the values of x and y. If x and y represent IP address domains (i.e., IP prefixes of length smaller than 32), then these filters are represented as rectangles; on the other hand, if x and y denote hosts (i.e., full 32-bit IP addresses), then the corresponding filters are represented as lines. Similarly, depending the lengths of x and y, fully-specified IP address filters of the form (x, y) represent lines, points, or rectangles in the two dimensional space.

The first two columns of Table 8 show the breakdown of partially- and fully-specified filters in our firewall ACLs. The third column of Table 8 shows the total number of filters, which is equal to the sum of the number of partially- and fully-specified filters plus one more filter representing the wildcard pair (*, *). Table 8 illustrates that, whereas partially-specified filters represent a small percentage of the total number of filters in large ISP databases; they constitute a significant percentage of the relatively small-size enterprise intranet firewall ACL. This is because large ISPs often describe administrative policies between specific IP address domains within their network. Examples of such policies include the admission of all HTTP traffic between a server and a client subnet, or the blocking of all RTSP traffic between two specific IP address domains. In intranets, on the other hand, administrators do not specify cross-domain traffic management policies, since such policies are often enforced by their ISP. Instead, most of the rules in intranet firewalls refer to specific sources or destinations, but not both.

We further analyze the partially-specified filters to determine the relative occurrence of the wildcard in the source or the destination IP address fields, as well as the lengths of specified IP addresses. We find that in the intranet ACL, which is the smallest in size, filters with the wildcard in the destination address are the majority. In the first two ACLs, which are of medium size, there is a balance between the filters that have the wildcard in the source and destination address fields. In the third ACL, which has the largest size, most filters have the wildcard in the source address field.

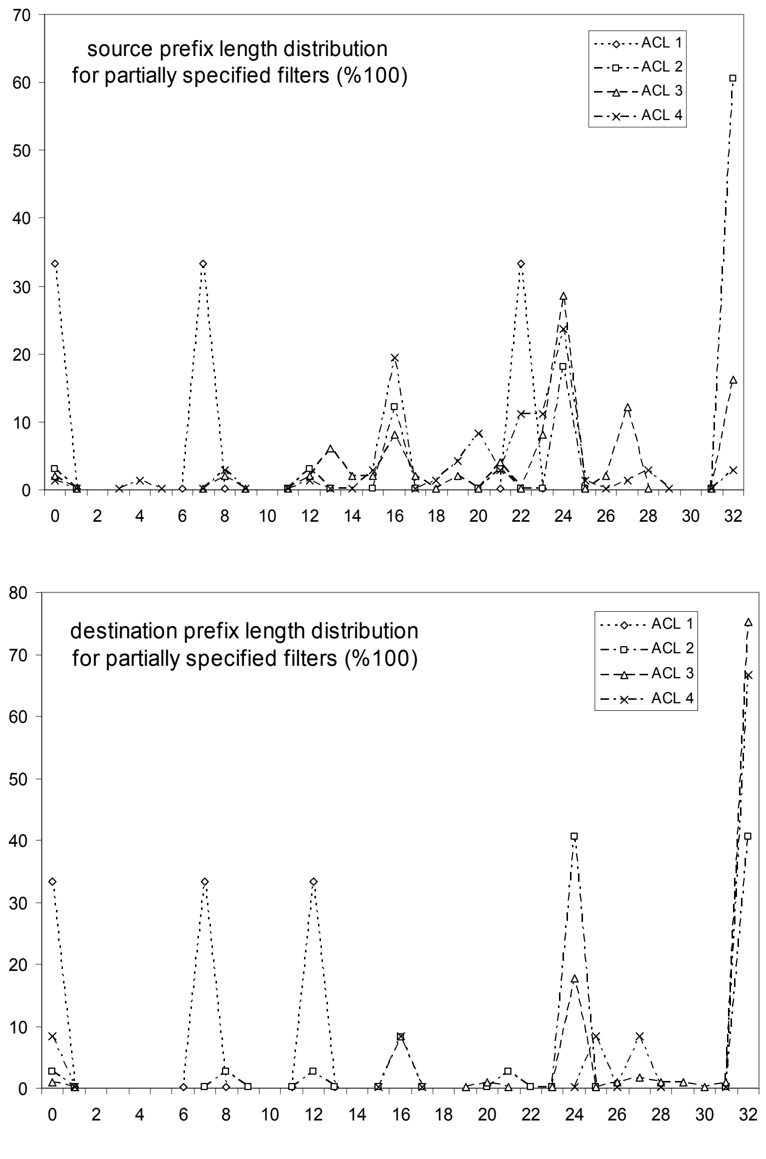


**Figure 46: Distribution of prefix lengths for partially-specified filters**

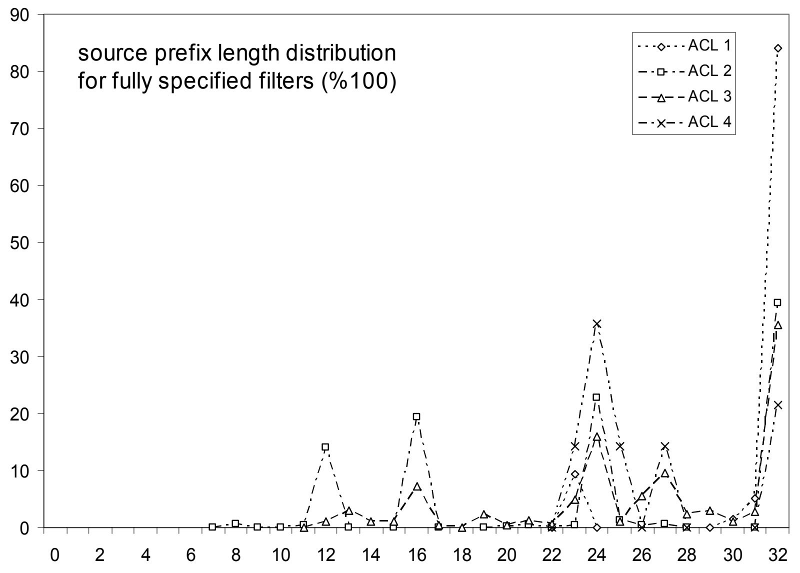
From the results of table 9 it appears as if there is a dependency between the size of an ACL and the numbers of filters that have the wildcard in the source or destination IP address fields. Typically, the smaller an ACL is the closer to client networks the firewall is located. The intranet ACL in our example describes policies that block the traffic form many specific client subnets of the intranet and thus contains many rules having the wildcard in the destination dimension. Larger ACLs on the other hand are closer to the Internet core and describe higher-level policies for connecting to important servers or networks. Such policies are typically expressed as rules having the wildcard in the source dimension.

| | domain-domain filters | host-domain filters | domain-host filters | host-host filters |
|---|---|---|---|---|
| ACL1 | 30 | 31 | 37 | 323 |
| ACL2 | 124 | 99 | 154 | 81 |
| ACL3 | 165 | 18 | 755 | 489 |
| ACL4 | 9 | 0 | 2 | 3 |

**Table 10: Breakdown of fully-specified filters**

Figure 46 shows the distribution of prefix lengths for partially-specified filters. It shows that the source and destination IP address specifications are spread across the entire range of prefix lengths, with 8-bit, 16-bit, 24-bit and 32-bit prefixes constituting the majority. Geometrically, this indicates that most partially-specified filters represent lines or rectangles characterized by a few standard width values in the two-dimensional space.

There are four types of fully-specified filters: (1) filters that characterize traffic exchanged between two domains, (2) filters that characterize traffic originating within a domain but destined to a host, (3) filters that characterize traffic originating from a host but destined to an IP domain, and (4) filters that characterize traffic exchanged between a specific pair of hosts. In these filters, a host is represented using a 32 bit address (IPv4 address) while a domain is represented by a shorter prefix. Table 10 shows the breakdown of these four types of filters in our ACLs. It shows that majority of the fully-specified filters in ISP databases represent communication where either the sender or the receiver is a host. In many cases these hosts are servers representing important resources of large networks. On the other hand, in the intranet ACL the majority of fully-specified filters represent Domain-Domain filters.



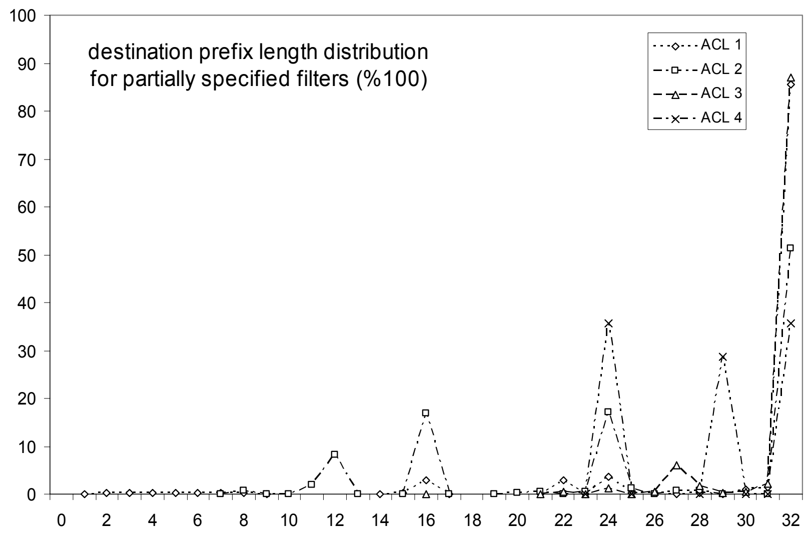


**Figure 47. Distribution of source and destination prefix lengths for fully-specified filters**

Figure 47 shows the distribution of source and destination prefix lengths for fully-specified filters. Geometrically, this indicates that most fully-specified filters represent lines or points in the two-dimensional space. The spatial distribution of IP prefixes is a very important property, especially to analyze requirements to store the IP prefixes. We have created 4-bit trie data structures for both the source and destination IP addresses, measured the number of trie blocks required to store IP prefixes, and compared this number with the theoretical maximum for number of trie blocks. The results are shown in Table 11. We find that the total number of trie blocks needed to represent source and destination prefixes is much less than the theoretical upper bound in real world data bases.

From the above analyses, we derive the following general conclusions:

1. Filters in real world ACLs are either fully-specified or partially-specified. Partially-specified filters represent a small percentage of the total number of filters in medium and large size ACLs.
2. The breakdown of partially-specified filters between filters having the wildcard in source and destination IP addresses may depend on the size of the ACL. Careful study of more ACLs would help investigating the existence of such dependency.
3. Most fully specified filters are segments of straight lines or points in medium and large size ACLs.
4. Trie data structures representing source and destination prefixes require much fewer blocks than the theoretical upper bound.

| | number of unique source prefixes | observed number of source trie blocks | theoretical bound on the number of source trie blocks |
|---|---|---|---|
| ACL1 | 97 | 29 | 759 |
| ACL2 | 182 | 231 | 1439 |
| ACL3 | 431 | 496 | 3256 |
| ACL4 | 79 | 127 | 615 |
| | number of unique destination prefixes | observed number of destination trie blocks | theoretical bound on the number of destination trie blocks |
| ACL1 | 205 | 383 | 1623 |
| ACL2 | 207 | 243 | 1639 |
| ACL3 | 516 | 620 | 3855 |
| ACL4 | 20 | 60 | 155 |

**Table 11: Trie-Block Analysis**

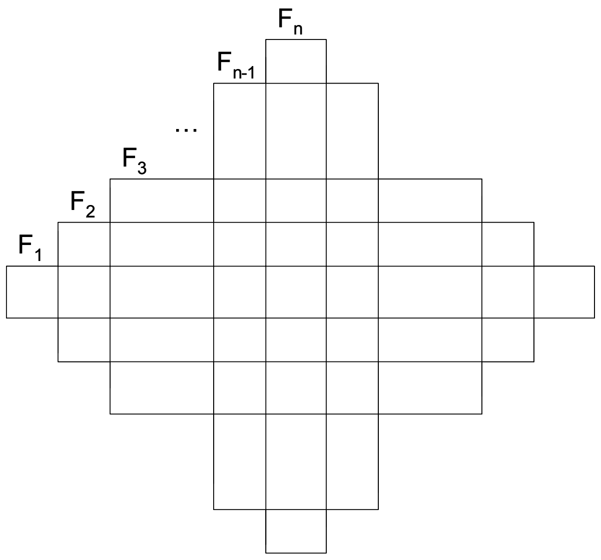


**Figure 48: Worst-case filter structure**

### 6.3.2 Overlap Analysis

The geometrical objects representing filters may overlay in the two dimensional space. Since each packet represents a point in the two dimensional space, it may be contained within the geometrical space defined by one or more filters in the ACL. In such an event, a packet may match multiple filters within the ACL; hence, identifying the highest priority rule requires comparing transport-level fields associated with all the matching filters with the appropriate fields contained in the packet. Clearly, the larger the number of filters that a packet may match with, the greater is the complexity of identifying the highest priority rule that matches the packet. In the worst-case, if all filters within an ACL overlap with each other (as shown in Figure 48), then identifying the highest priority rule for a packet that represents a point in the intersection of these filters may require a search on all filters. Thus, the complexity of packet classification depends on the amount of overlap between filters (which in turn determines the number of filters that may match a packet).

In what follows, we analyze our ACLs for their overlap properties. Table 12 and Figure 49 show the distribution of the number of filters that may match a packet. Figure 49 illustrates that for all the ACLs, on an average, about 4 filters match every packet. Although this is not a very large number, identifying these filters imposes significant overhead. The navigation on the data structures that store two-dimensional filters (e.g., hierarchical singe-bit or multi-bit tries) typically requires significantly more memory accesses than the number of filters matching a packet. For instance, the Extended Grid of Tries (EGT) algorithm reported in [10] requires 137 accesses to classify packets from a database of 2799 rules.

| | average | standard deviation | maximum |
|---|---|---|---|
| ACL1 | 4.00 | 0.36 | 5.00 |
| ACL2 | 3.96 | 0.73 | 7.00 |
| ACL3 | 3.75 | 0.84 | 7.00 |
| ACL4 | 3.71 | 0.90 | 7.00 |

**Table 12: Number of Filters that May Match a Packet**

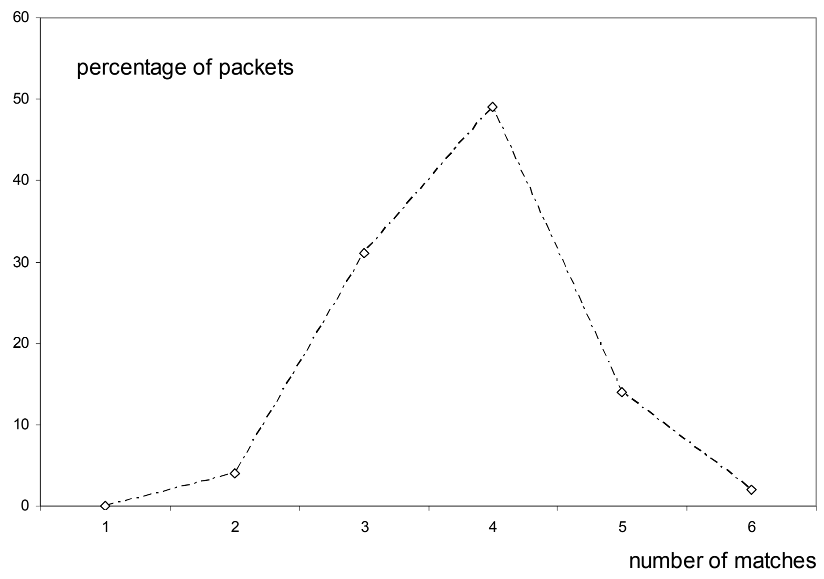


**Figure 49: Distribution of the number of filters that may match a packet**

An alternative architecture involves maintaining a filter that represents each overlap in the ACL. We observe that overlaps between filters can be complete or partial. In the event that one filter is completely contained in another, the overlap between the filters is represented exactly by the contained filter. In such a case, no additional filter needs to be stored. On the other hand, if filters overlap partially, then the overlap can be identified uniquely by a filter that represents the intersection region between the two filters; hence, each partial overlap introduces a new filter in the ACL. If all such filters are maintained, then the classifier can determine the most refined filter for each packet. In the worst case, if each filter overlaps with all the other filters in the ACL, then maintaining all the intersection filters would incur an $O(n^2)$ overhead , where $n$ is the number of distinct IP prefix pairs in the ACL. The worst-case scenario is one where each filter in the ACL overlaps with all the other filters. In such a case, the number of filters that represents the overlaps can be bounded by $(n-1) + (n-2) +\ldots+ 1 = n(n-1)/2$. However, as we have illustrated earlier, most ACLs contain filters that can be represented as points, lines or small rectangles. Hence, we can expect the number of additional filters required for real ACLs to be much smaller than the theoretical worst-case.

| | number of rules | number of filters | observed number of partial overlaps | upper bound on the number of partial overlaps |
|---|---|---|---|---|
| ACL1 | 754 | 426 | 4 | 90,525 |
| ACL2 | 607 | 527 | 2,249 | 138,601 |
| ACL3 | 2,399 | 1,588 | 6,138 | 1,260,078 |
| ACL4 | 157 | 98 | 852 | 4,753 |

**Table 13: Observed Filter Overlaps**

To validate this hypothesis, we determine the number of such overlap observed in our ACLs. The results are shown in Table 13. Table 13 indicates that the number of filters representing intersections that may need to be stored is, in fact, several orders of magnitude smaller than the theoretical upper bound.

Our analyses of the ACLs show that the organization of filters in real-world ACLs is significantly different from the worst-case structure shown in Figure 48. A more realistic structure of filters is shown in Figure 50. The filters in the structure of Figure 5 are either-fully specified or partially- specified as explained in the previous section. Some fully-specified filters form ‘clusters’ as shown in Figure 50. A cluster is a set of filters where every filter overlaps either partially or completely with at least one other filter in the cluster. A closer analysis of the ACLs reveals that there are three cases that create partial overlap between filters.

1. Overlaps between partially-specified filters. Each filter having the wildcard in the source IP address dimension creates a unique partial overlap with all the filters having a wildcard in the destination IP address dimension. Since IP addresses are specified as prefixes, filters with a wildcard in the same dimension do not create partial overlaps between each other; such filters are either disjoint or completely overlapping. The number of partial overlaps created only by partially-specified filters is equal to the product of the number of partially-specified filters in each of the two dimensions.
2. Overlaps between fully-specified filters. Fully-specified filters may overlap with each other either fully or partially.
3. Overlaps between fully- and partially-specified filters.

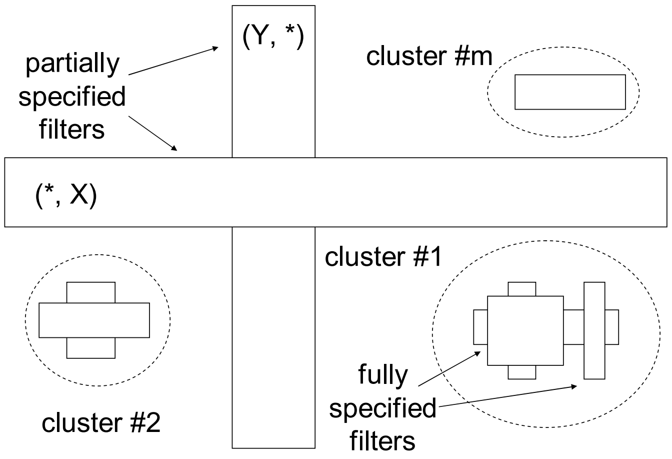


**Figure 50: A realistic structure of filters in ACLs.**

Table 14 shows the breakdown of the number of partial overlaps created in each of the four ACLs. It shows that the overlaps created by partially-specified filters represent the majority in all ACLs, ranging from 51% in ACL 2 up to 100% in ACL 1 and ACL 4. We also observe that the overlaps created between partially and fully-specified filters represent a significant percentage (45%) of the total number of overlaps in ACL 2. In all the ACLs, fully-specified filters create an insignificant number of overlaps (it turns out that most clusters have size equal to one). These results indicate that partially-specified filters are the main source of overlaps in all ACLs. Further, as we had demonstrated earlier, partially-specified filters generally represent only a small percentage of the total number of filters in large databases. These two observations together justify why the total number of partial overlaps is significantly less than the theoretical upper bound. In Appendix A, we derive a tighter upper bound on the number of partial overlaps.

| | number of overlaps | overlaps formed by partially specified filters only | overlaps formed by fully specified filters only | overlaps formed by between fully ad partially specified filters |
|---|---|---|---|---|
| ACL 1 | 4 | 100% | 0% | 0% |
| ACL 2 | 2249 | 51% | 4% | 45% |
| ACL 3 | 6138 | 88% | 1% | 11% |
| ACL 4 | 852 | 100% | 0% | 0% |

**Table 14: Breakdown of Overlaps**

## 6.4 Transport-Level Field Analysis

The Internet supports thousands of routes but relatively only a few, commonly used applications. Hence, as indicated in Table 7, only a small number of unique transport-level fields (consisting of port numbers and protocol types) are usually present in ACLs. Further, many source-destination pairs share the same transport-level fields. In what follows, we first analyze the transport-level fields associated with individual source-destination pairs (or IP address filters) and then expose the sharing of these transport-level fields across multiple IP filters.

### 6.4.1 Analysis of Transport-Level Fields for Individual IP Filters

ACLs generally contain several rules with the same IP address filter (i.e., source-destination IP address pair) but with different combination of transport-level fields. To understand this phenomenon carefully, we analyzed the sets of such transport-level fields associated with the same IP filters.

Figure 51 depicts the distribution of the set sizes observed in the four ACLs under consideration. It shows that for all the ACLs, most (about 90%) transport-level field sets are small (1-4 entries); the remaining 10% of the sets have sizes between 5 and 40. This is mainly because most ACLs contain rules that identify explicitly only a small number of the most popular applications; in today's Internet the number of these applications is very small.

We observe that the highest percentage of transport-level fields in our ACLs specify TCP and UDP protocols. This is because most data traffic in today's Internet uses TCP and a smaller percentage of traffic uses UDP. Further, most transport-level fields specify a destination port or port range. The source port field is generally unspecified (i.e., a wildcard specification). This is because most classification rules apply to packets that request the establishment of TCP connections. These packets are sent to servers that are listening to well-known non-ephemeral or ephemeral ports. Table15 depicts the distributions of the source and destination port numbers observed for the four ACLs.

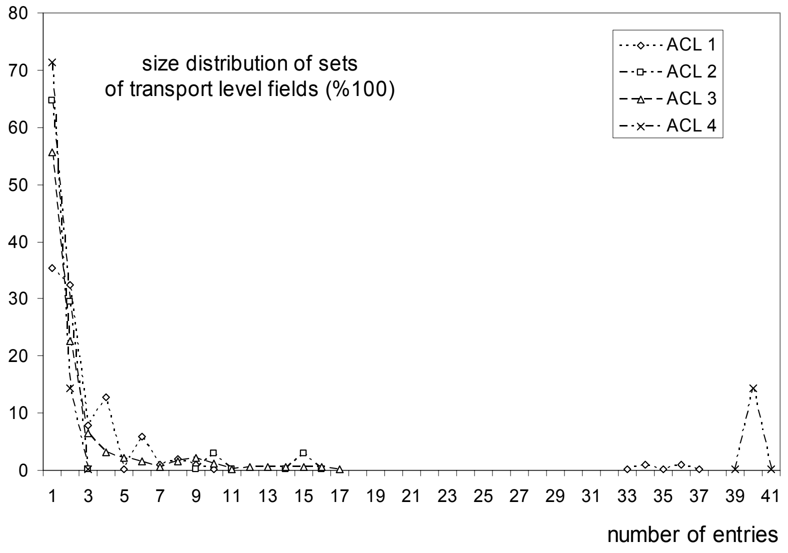


**Figure 51: Distribution of sizes of transport level field sets**

### 6.4.2 Sharing Transport-Level Fields Across Multiple IP Filters

To analyze the sharing of transport-level fields across multiple IP filters, we derive the total number of transport-level field entries with and without any sharing across filters. Table 16 summarizes our findings. It shows that for all ACLs, many source-destination IP prefix pairs share the same sets of transport-level fields. The relative priority and corresponding actions of fields are the same in different occurrences of each set. In addition the number of unique entries characterizing the shared sets of transport level fields is also small. This number is much smaller than the total number of entries in the unique sets.

| | number of unique transport-level fields | source port number | | | destination port number | | |
|---|---|---|---|---|---|---|---|
| | | wildcard | range | exact value | wildcard | range | exact value |
| ACL1 | 146 | 146 | 0 | 0 | 4 | 74 | 68 |
| ACL2 | 40 | 40 | 0 | 0 | 8 | 29 | 3 |
| ACL3 | 202 | 200 | 2 | 0 | 5 | 157 | 40 |
| ACL4 | 43 | 42 | 1 | 0 | 8 | 32 | 3 |

**Table 15: Distribution of Source and Destination Port Number in Transport-Level Fields**

| | number of transport-level fields | number of transport-level fields in unique sets | number of unique transport-level fields |
|---|---|---|---|
| ACL1 | 754 | 316 | 146 |
| ACL2 | 607 | 67 | 40 |
| ACL3 | 2399 | 442 | 202 |
| ACL4 | 157 | 48 | 43 |

**Table 16: Sharing Transport-Level Fields Among IP Filters**

## 6.5 Implications

Our evaluation of ACLs leads us to the following main conclusions.

1. The fields contained in each rule in ACLs can be partitioned into two logical entities: (1) source and destination IP address pairs that characterize distinct network paths represented in ACLs, and (2) a set of transport level fields (e.g., port numbers, protocol identifier, etc.) that characterize network applications. In most cases, the number of distinct network paths far exceeds the number of network applications.
2. The IP address filters are either partially-specified or fully-specified. Partially-specified filters represent a small percentage of the total number of filters in databases. Furthermore, most of the overlap between filters is caused by partially-specified filters. Fully-specified filters create only a few partial overlaps with each other. Thus, the total number of overlaps is significantly smaller than the theoretical bound.
3. Many source-destination IP address pairs share the same set of transport-level fields. Hence, only a small number of transport-level fields are sufficient to characterize databases of different sizes.

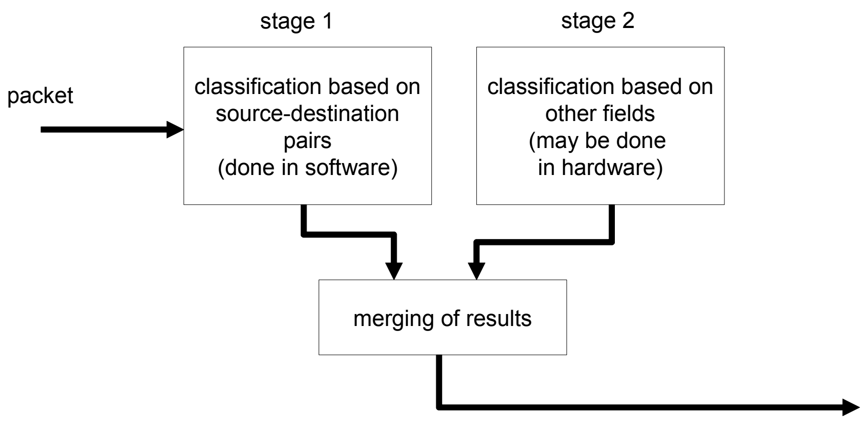

stage 1
stage 2
packet
classification based on source-destination pairs (done in software)
classification based on other fields (may be done in hardware)
merging of results

**Figure 52: Two stage classification architecture**

Based on these findings, we provide the following guidelines for designing efficient classification algorithms.

1. The multi-dimensional classification problem should be split into two sub-problems (or two stages): (1) finding a 2-dimensional match based on source and destination IP addresses contained in the packet, and (2) finding a (*n*-2) dimension match based on transport-level fields (see Figure 52). Whereas the first stage only involves prefix matching, the second stage involves the more general range matching.
2. Because of the overlap between IP address filters maintained in an ACL, each packet may match multiple filters in stage 1. Identifying all the matching filters is complex. Since the total number of overlaps observed for ACLs is significantly smaller than the theoretical upper-bound, a design that maintains all of the intersection filters and returns exactly a single match from stage 1 is both feasible an desirable. For example, we believe that a new two-dimensional scheme can be thought of that searches the source and destination IP address dimensions in parallel, avoiding the memory explosion problem associated with Cross Producting [131]. The memory explosion problem can be solved in this case by taking into account the fact that IP address filters create an insignificant amount of filter overlaps with each other.
3. Since each IP address filter is associated with multiple transport-level fields, identifying the highest priority rule that matches a packet requires searching through all the transport-level fields associated with the matching IP filter. Since the number of transport-level fields associated with most ACLs is rather small, it is possible to rely upon a small, special-purpose hardware unit (e.g., a TCAM unit) to perform the (*n*-2) dimensional search in parallel.

The combination of a fast software algorithm for finding a 2-dimensional match in stage 1 and a specialized hardware acceleration unit for performing (*n*-2) dimensional match in stage 2 can result in a classification system capable of meeting stringent space time constraints.

## 6.6 Summary

To classify a packet as belonging to a flow often requires network systems—such as routers and firewalls—to maintain large data structures and perform several memory accesses. Network processor-based programmable routers, on the other hand, are generally configured with only a small amount of memory with limited access bandwidth. Hence, a key challenge is to design packet classification algorithms that can be implemented efficiently. We argue that the design of such algorithms will need to exploit the structure and characteristics of packet classification rules.

In this chapter, we analyze several databases of classification rules found in firewalls and derive their statistical properties. Our analysis yields three main conclusions: (1) the rules found in ACLs contain two types of fields—source-destination IP address pairs that identify network paths and transport-level fields that characterize network applications; further, these rules refer to many more network paths than applications. (2) IP address pairs identify regions that overlap with each other; however, the number of overlaps is significantly smaller than the theoretical upper-bound. (3) Only a small number of transport-level fields are sufficient to characterize ACLs of different sizes. We justify our findings based on several standard practices employed by network administrators, and thereby argue that although our findings are for specific databases, the properties are likely to hold for most databases. Based on these findings, we suggest that a hybrid, two-stage classification architecture that combines a software scheme for matching in 2-dimensions (IP address pairs) with a hardware unit that performs efficient ($n$-2) dimensional searches has the potential of scaling well with link speeds and ACL sizes.

## Appendix A: A Tighter Bound on the Number of Partial Filter Overlaps

From the analyses of ACLs, we have shown that the number of overlaps between IP filters is significantly smaller than the theoretical upper-bound of $n(n$-1)/2. In this appendix, we derive a tighter upper bound on the number of partial filter overlaps. The derivation of the upper bound is based on properties that characterize medium size and large ISP ACLs. Therefore the analysis presented in this appendix applies to the first three of our ACLs only.

There are three factors that produce intersections between IP filters in ACLs. First, partially-specified filters create intersections with each other. The number of such overlaps $O_1$ is exactly equal to $S \cdot D$ where $S$ is the number of partially-specified filters that specify the source IP address dimension and D is the number of partially-specified filters that specify the destination IP address dimension. Since partially-specified filters represent a small percentage of the total number of filters in all databases (1%- 13%) we expect their overlaps to be bounded by the square of the number of filters divided by a large constant. In fact, the majority of partial overlaps (51%-100%) are created by partially-specified filters in databases.

Second, partial overlaps result from the intersections between fully-specified filters in the same cluster. However, clusters with more than one element are only but a few in our ACLs. Fully specified filters form an insignificant amount of overlaps between each other. This happens mainly because server and client subnets are characterized by disjoint IP address domains in rules. As a result the number of partial overlaps $O_2$, created by fully-specified filters, is also much less than the theoretical upper bound.

Third, partial overlaps are created between fully and partially-specified filters. Each fully-specified filter may create partial overlaps with one or more partially-specified filters. In most medium size and large ACLs the total number of servers specified per IP address domain is bounded. These servers represent the prefixes of partially specified filters. As a result the total number of overlaps formed between fully-and partially-specified filters $O_3$ is bounded by the product of the number of filters times a constant. The detailed derivation of an upper bound is given below:

$$O = O_1 + O_2 + O_3 \qquad \text{[Eq. 4]}$$

$$O_1 = S \cdot D \qquad \text{[Eq. 5]}$$

$$O_2 \leq \sum_{i=1}^{q} \frac{C_i \cdot (C_i - 1)}{2} \qquad \text{[Eq. 6]}$$

$$O_3 = \sum_{j=1}^{f} F_j \qquad \text{[Eq. 7]}$$

Equation 4-7 result in:

$$O \leq S \cdot D + \sum_{i}^{q} \frac{C_i \cdot (C_i - 1)}{2} + \sum_{j=1}^{f} F_j \qquad \text{[Eq. 8]}$$

$S$ and $D$ are the number of partially-specified filters that specify the source and destination IP address dimensions respectively, $q$ is the number of clusters that contain more than one filter, $C_i$ is the number of filters in cluster $i$, $f$ is the number of fully-specified filters that create partial overlaps with partially-specified filters, and $F_j$ is the number of partially-specified filters that overlap with filter $j$. To complete the derivation of an upper bound we need to understand the relation between the parameters $S$, $D$, $q$, $f$, $C_i$, $F_j$ and the number of filters in a database, $n$.

Let $r_1$, be the ratio of the total number of filters in a database over the number of partially-specified filters. Then $S+D = n/r1$. The ration $r_1$ is expected to be greater than one with very high probability in many different databases. The number of overlaps formed by partially-specified filters $O_1$, is equal to the product $S \cdot D$. This product is maximized when $S = D = n/(2 \cdot r_1)$. Therefore:

$$O_1 \leq \frac{n^2}{4 \cdot r_1^2} \qquad \text{[Eq. 9]}$$

Let $r_2$, be the ratio between the number of fully-specified filters in a database and the number of fully-specified filters that create partial overlaps with each other. Such filters participate in clusters having more than one element. The ratio $r_2$ is also expected to be greater than one with very high probability. The number of fully-specified filters that create partial overlaps with each other is equal to $(r_1 - 1) \cdot n/(r_1 \cdot r_2)$.

As a result:

$$O_2 \leq \frac{(r_1 - 1) \cdot n}{2 \cdot r_1 \cdot r_2} \left( \frac{(r_1 - 1) \cdot n}{r_1 \cdot r_2} - 1 \right) \qquad \text{[Eq. 10]}$$

The values of $F_j$ refer to overlaps between partially and fully-specified filters. The number of such overlaps per fully-specified filter is independent of n as a property of a pair of IP address domains. As a result we can consider that each $F_j$ is bounded by some value $F$. Therefore:

$$O_3 \leq n \cdot F \qquad \text{[Eq. 11]}$$

Eq. 4, 9-11 result in:

$$O \leq \frac{n^2}{4 \cdot r_1^2} + \frac{(r_1 - 1) \cdot n}{2 \cdot r_1 \cdot r_2} \left( \frac{(r_1 - 1) \cdot n}{r_1 \cdot r_2} - 1 \right) + n \cdot F \qquad \text{[Eq. 12]}$$

Even though the result of Eq. 12 is also $O(n^2)$ this bound is much tighter than the worst case. This happens because the number of filters n is divided by the parameters $r_1$ and $r_2$ in Eq. 12. The parameters $r_1$ and $r_2$ are expected to be greater than one. Another difference is that the worst case bound is a deterministic bound whereas the bound of Eq. 12 is a stochastic bound, since the parameters $r_1$, $r_2$ and $F$ are random variables.

The random variables $r_1$, $r_2$ and $F$ have unknown distributions. However, we expect with very high probability that $r_1$ and $r_2$ are greater than one and $F$ is a small number. Estimations on the upper bound of Eq. 12 can be derived by selecting the values with the highest frequency from the limited number of databases we experimented with, for $r_1$, $r_2$ and $F$. The values for $r_1$, $r_2$ and $F$ used in our calculations are $r_1 = 8.75$, $r_2 = 4.3$ and $F = 15$. More accurate results would require the parameters to be estimated from a greater number of samples. Our results are shown in Table 17. The worst case estimations derived from Eq. 12 are compared against the worst case estimations in this table.

| database number | number of filters | observed number of partial overlaps | upper bound from Eq. 9 | worst case upper bound |
|---|---|---|---|---|
| 1 | 426 | 4 | 10,790 | 90,525 |
| 2 | 527 | 2,249 | 14,651 | 138,601 |
| 3 | 1,588 | 6,138 | 85,393 | 1,260,078 |

**Table 17: Upper Bounds on the Number of Partial Filter Overlaps**

# Chapter 7

# Conclusion

In the thesis we address the problem of programming network architectures. Programming Network Architectures is not a trivial problem. The difficulty stems from the fact that it is hard to define a unifying programmable networking model and a set of programming interfaces that encompass services as diverse as routing, signaling, and access control/forwarding. Another challenging issue is related to the computational efficiency and performance of programmable network architectures. Programmable networks require more computational resources than existing networks in order to support the introduction of new services in software. In addition, today's router systems are generally configured with only a small amount of memory with limited access bandwidth. Hence, a key challenge is to design programming systems and network algorithms that can operate efficiently under stringent space-time constraints.

This thesis makes several contributions. First, we propose a programmable networking model that provides a common framework for understanding the state-of-the-art in programmable networks. A number of projects are reviewed and discussed against a set of programmable network characteristics. We present a simple qualitative comparison of the surveyed work and make a number of observations about the direction of the field.

Next, we present the design, implementation and evaluation of a programming system that automates the life cycle process for the creation, deployment, management, and architecting of network architectures. We discuss our experiences in building a “spawning” network testbed that is capable of creating distinct network architectures on-demand. Network architectures are created as programmable virtual networks. Our programming system is based on a methodology that allows a “child” network to operate on top of a subset of its “parent’s” network resources and in isolation from other spawned virtual networks. We show through experimentation how a number of diverse network architectures can be spawned and architecturally refined.

Third, we discuss how we can support end-system connectivity with programmable network architectures. We focus on wireless cellular network architectures, where the connectivity problem is more challenging due to host mobility. We describe a ‘reflective handoff’ service that allows access networks to dynamically inject signaling systems into mobile devices before handoff. Thus, mobile devices can seamlessly roam between wireless access networks that support different mobility management systems. We also show how a ‘multi-handoff’ access network service can be introduced that simultaneously supports different styles of handoff control over the same wireless access network. This programmable approach can benefit service providers who need to be able to satisfy the mobility management needs of a range of mobile devices from cellular phones to palmtop and laptop computers.

Next, we study the performance of network programming systems. In particular, we focus on problem of efficiently programming the data path. Programming the data path is challenging because data path algorithms operate on the fastest time scale and are associated with the small time budgets. We focus on an implementation of a network programming system in network processor-based routers. Network processors comprise multiple processing units for parallel packet processing and constitute suitable building blocks for software-base routers. We propose the design of a binding tool that balances the flexibility of network programmability against the need to process and forward packets at line speeds. To support dynamic binding of components with minimum addition of instructions in the critical path, our tool modifies the machine language code of components at run time. To support fast data path composition, our tool reduces the number of binding operations required for constructing data paths to a minimum set so that binding latencies are comparable to packet forwarding times.

Finally, we study the realization of performance critical algorithms such as packet classification in programmable networks. Performance critical algorithms include classification, forwarding and traffic management. While forwarding and traffic management are problems that have been investigated in the past we still lack a good solution for the packet classification problem. We conjecture that the design of classification algorithms will need to exploit the structure and characteristics of packet classification rules. We study the properties of several classification data bases and, based on these findings, we suggest a classification architecture that can be implemented efficiently in programmable networks.

We believe that the Genesis Kernel programming system can be used for solving many interesting networking problems, which were not addressed as part of this thesis. The first problem, which we aim to solve as part of future work is the problem of network architecture analysis. This problem can be stated as follows: Given a network architecture defined as a set of protocols for transport, control and management, a network topology and a set of network resources (i.e., link and processing capacities), what is the cost and performance of the architecture when deployed over the given topology and set of resources? To address this problem we plan to develop an analytical model for network architectures and use the Genesis Kernel to visualize spawned network architectures. We plan to define and visualize parameters capturing the cost and performance of network architectures and investigate how these parameters change when the distributed algorithms of network architectures are modified.

A second problem we plan to investigate is the problem of network architecture synthesis. The synthesis problem can be stated as follows: Given the cost and performance requirements of a network architecture, a network topology, and a set of network resources what is the minimum cost network architecture that can meet the specified cost and performance requirements, when operating over the given topology and resources. To solve this problem we plan to develop a methodology for network synthesis based on experimentation with the Genesis Kernel. We plan to spawn different network architectures, observe their cost and performance and discover ways to build better networks. Our goal is to develop a methodology for finding optimal network architectures for given network topologies, resources and input traffic.

We would like to investigate whether an optimal network architecture exists or if it can be found for a given communications infrastructure. The design of the optimal network architecture may depend on properties of the input traffic, topology and network resources. Knowing the input traffic before an architecture is deployed is not possible. To ensure that a running architecture operates as optimally as possible, a network architect may need to monitor changes in the input traffic and topology periodically. Once the input traffic is estimated and current topology discovered a minimum cost architecture that satisfies the estimated network operating conditions could be determined. Then, the currently executing network architecture could be modified so that it becomes more optimal. We call this periodic cycle of estimating the network operating conditions and modifying a network architecture so that it satisfies its input traffic demands 'architecting'.

We believe that architecting represents an important challenge in networking. For example the problem of designing more resilient networks that respond to sudden changes in the traffic demand or topology (e.g., after disasters occur) can be consider as a special case of the more generic architecting problem as described above. To solve the architecting problem we need to have a good understanding of how network algorithms affect the performance and cost of network architectures. We lack such understanding today. Our research aims to fill this gap. Currently, we have completed the implementation and evaluation of Genesis Kernel v1.0. Our Genesis Kernel prototype runs on top of a programmable infrastructure of PC-based and IXP1200 network processor-based routers. We plan to develop an analytical model for network architectures and use the Genesis Kernel to investigate the impact of network algorithms on the cost and performance of network architectures.

# Chapter 8

# My Publications as a Ph.D Candidate

My Publications as a Ph.D Candidate (1998-2003) are listed below. This list also includes research that is indirectly related to the work presented in this thesis including, the design and implementation of QOS-aware and event-driven middleware platforms, and the specification of network programming interfaces for programming QOS in the internet.

## 8.1 Patents

- M. E. Kounavis, A. Kumar, H. Vin, and R. Yavatkar, "Method and Apparatus for Two Stage Packet Classification using Most Specific Filter Matching Transport Level Sharing", *United States Patent Application,* filed.

## 8.2 Journal Publications

- M. E. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. T. Campbell, "Directions in Packet Classification for Network Processors", *Network Processor Design: Issues and Practices*, Volume 2, to appear, 2003.

- M. E. Kounavis, A. T. Campbell, S. Chou, and J. Vicente "Programming the Data Path in Network Processor-based Routers", *Software Practice and Experience*, to appear 2003.
- M. E. Kounavis, A. T. Campbell, S. Chou, F. Modoux, J. Vicente and H. Zhuang "The Genesis Kernel, A Programming System for Spawning Network Architectures", *IEEE Journal on Selected Areas in Communications*, Vol. 19, No. 3, pg. 511-526, March 2001.
- M. E. Kounavis, A. T. Campbell, G. Ito, and G. Bianchi, "Design, Implementation and Evaluation of Programmable Handoff in Mobile Networks", *ACM/Kluwer Journal on Mobile Networks and Applications*, September 2001.
- A. T. Campbell, M. E. Kounavis, and R. R.-F. Liao, "Programmable Mobile Networks", *Computer Networks*, Vol. 31, No. 7, pg. 741-765, April 1999.
- A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. Vicente, and D. Villela, "A Survey of Programmable Networks", *ACM SIGCOMM Computer Communications Review*, Vol. 29, No 2, pg. 7-23, April 1999.
- A. T. Campbell, M. E. Kounavis, D. Villela, J. Vicente, H. G. De Meer, K. Miki, and K. S. Kalaichelvan, "Spawning Networks", *IEEE Network*, Vol. 13, No. 4, pg. 16-29, July/August 1999.
- O. Angin, A. T. Campbell, M. E. Kounavis, and R. R.-F. Liao, "The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking", *IEEE Personal Communications, Special Issue on Adapting to Network and Client Variability*, Vol. 5, No. 4, pg. 32-43, August 1998.
- A. T. Campbell, G. Coulson, and M. E. Kounavis "Managing Complexity: Middleware Explained", *IT Professional Magazine*, Vol.1, No. 5, pg. 22-28, September/October 1999.
- A. T. Campbell, M. E. Kounavis and J. Vicente, "Programmable Networks", Reinhard Wilhelm (ed.), *Dagstuhl 10 Year Anniversary Proceedings*, Springer-Verlag, Lecture Notes in Computer Science 2000, pp. 34-49, 2001.

## 8.3 Conference Publications

- M. E. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. T. Campbell, "Directions in Packet Classification for Network Processors", *Second Workshop on Network Processors (NP2)*, Anaheim, CA, February 2003.
- M. E. Kounavis, A. T. Campbell, S. Chou, and J. Vicente "A Programming Environment for Network Processors", *Network Processors Conference West,* San Jose, CA, October 2002.
- A. T. Campbell, S. Chou, M. E. Kounavis, V. D. Stachtos and J. Vicente, "Netbind: A Binding Tool for Constructing Data Paths in Network Processor-based Routers", *Fifth International Conference on Open Architectures and Network Programming (OPENARCH' 02)*, New York, June 2002.
- V. D. Stachtos, M. E. Kounavis, and A. T. Campbell, "Sphere: A Binding Model and Middleware for Routing Protocols", *Fourth International Conference on Open Architectures and Network Programming (OPENARCH' 01)*, Anchorage, Alaska, April 2001.
- M. E. Kounavis, A. T. Campbell, G. Ito, and G. Bianchi, "Accelerating Service Creation and Deployment in Mobile Networks", *Third International Conference on Open Architectures and Network Programming (OPENARCH' 00)*, Tel-Aviv, Israel, March 2000.
- A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. Vicente, and D. Villela, "The Genesis Kernel: A Virtual Network Operating System for Spawning Network Architectures", *Second International Conference on Open Architectures and Network Programming (OPENARCH' 99)*, New York, March 1999.
- K. Tanaka, M. E. Kounavis, and A. T. Campbell, "Automating the Creation of Personalized Mobile Multimedia Services Using Cellware", *Tenth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV' 00)*, Chapel Hill, North Carolina, USA, June 2000.
- O. Angin, A. T. Campbell, M. E. Kounavis, and R. R.-F. Liao, "Open Programmable Mobile Networks", *Eighth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV' 98)*, Cambridge, U.K., July, 1998.

- A. Balachandran, A. T. Campbell, and M. E. Kounavis, "Active Filters: Delivering Scaled Media to Mobile Devices", *Seventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV' 97)*, Saint Louis, May, 1997.
- J. Vicente, M. E. Kounavis, D. Villela, M. Lerner, and A. T. Campbell, "Programming Internet Quality of Service", *Third International Conference on Trends toward a Universal Service Market (USM' 2000)*, Munich, Germany, September, 2000.
- A. T. Campbell, and M. E. Kounavis, "Toward Reflective Network Architectures", *Workshop on Reflective Middleware*, New York, NY, April 2000.
- M. E. Kounavis, A. T. Campbell, G. Ito, and G. Bianchi, "Supporting Programmable Handoff in Mobile Networks", *Sixth International Workshop on Mobile Multimedia Communications (MoMuC' 99)*, San Diego, CA, November 1999.
- R. R.-F. Liao, M. E. Kounavis, and A. T. Campbell, "The Design Implementation and Evaluation of the Mobiware Toolkit", *Fifth Internation Workshop on Mobile Multimedia Communications (MoMuC'98)*, Berlin, Germany, October, 1998.
- S. Chou, M. E. Kounavis, A. T. Campbell, and J. Vicente, “Genesis Kernel on the IXP1200”, *Concepts and Applications of Programmable and Active Networking Technologies*, Dagstuhl Seminar Series, February 2002.
- M. E. Kounavis, S. Chou, V. D. Stachtos and A. T. Campbell, "Routelets and Network Processors", *Next Generation Network Programming (OPENSIG' 01)*, London, UK, September 2001.
- A. T. Campbell, S. Chou, M. E. Kounavis and V. D. Stachtos, "Implementing Routelets: Virtual Router Support for the IXP1200 Network Processor", *IXA Univeristy Program Workshop*, Portland, Oregon, June 2001.
- M. E. Kounavis, "Implementing Spawning Networks", *Programming the Internet (OPENSIG' 00)*, Napa, CA, October 2000.
- M. E. Kounavis, and A. T. Campbell, "Reflective Handoff", *Open Signaling for ATM, Internet, and Mobile Networks (OPENSIG' 99)*, Pittsburg, PA, October 1999.

## 8.4 Documents in Preparation and Technical Reports

- M. E. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. T. Campbell, "Packet Classification under Stringent Space-Time Constraints”, *Technical Report Submitted for Publication*, February 2003.
- M. E. Kounavis, and A. T. Campbell "Programming the Ether”, *Technical Report*, October 2000.

## References

[1] ABONE, Active network Backbone, http://www.isi.edu/abone/

[2] C. M. Adam, A. A. Lazar, K-S. Lim, and F. Marconcini, “The Binding Interface Base Specification Revision 2.0”, *OPENSIG Workshop on Open Signalling for ATM, Internet and Mobile Networks*, Cambridge, UK, April 1997.

[3] D. S. Alexander, B. Braden, C. A. Gunter, W. A. Jackson, A. D. Keromytis, G. A. Milden, and D. A. Wetherall, “Active Network Encapsulation Protocol (ANEP)”, *Active Networks Group Draft*, July 1997

[4] D. S. Alexander, W. A. Arbaugh, M. A. Hicks, P. Kakkar, A. Keromytis, J. T. Moore, S. M. Nettles, and J. M. Smith, “The SwitchWare Active Network Architecture”, *IEEE Network Special Issue on Active and Programmable Networks*, vol. 12 no. 3, 1998.

[5] E. Amir, S. McCanne, and R. Katz, “An Active Service Framework and its Application to real-time Multimedia Transcoding”, *Proceedings ACM SIGCOMM’ 98,* Vancouver, Canada

[6] N.G. Aneroussis, and , A.A Lazar., "Virtual Path Control for ATM Networks with Call Level Quality of Service Guarantees", *IEEE Transactions on Networking*, Vol. 6, No. 2, April 1998, pp. 222-236.

[7] O. Angin, A. T. Campbell, M. E. Kounavis, and R. R.-F. Liao, “The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking”, *IEEE Personal Communications Magazine*, Special Issue on Adaptive Mobile Systems, August 1998.

[8] The ARRCANE Project, http://www.docs.uu.se/arrcane/

[9] F. Baboescu, G. Varghese, “Scalable Packet Classification”, *Proceedings of ACM Sigcomm*, pages 199-210, August, 2001.

[10] F. Baboescu, S. Singh, and G. Varghese, “Packet Classification for Core Routers: Is there an alternative to CAMs?”, *Technical Report*, University of California, San Diego, 2003.

[11] P. Bagwat, C. Perkins, and S. Tripathi, “Network Layer Mobility: an Architecture, and Survey”, *IEEE Personal Communications Magazine*, June 1996.

[12] A. Balachandran, A. T. Campbell, and M. E. Kounavis, “Active Filters: Delivering Scalable Media to Mobile Devices” , *Proc. Seventh International Workshop on Network and Operating System Support for Digital Audio and Video*, St Louis, May, 1997.

[13] A. Barabasi, R. Albert, "Emergence of Scaling in Random Networks", *Science*, Vol. 286, pp. 509-512, 1999.

[14] J. C. R. Bennett, and H. Zhang, “Hierarchical Packet Fair Queueing Algorithms”, *IEEE/ACM Transactions on Networking*, 5(5):675-689, Oct 1997.

[15] B. N. Bershad, et al., “Extensibility, Safety and Performance in the SPIN Operating System”, *Fifth ACM Symposium on Operating Systems Principles*, Copper Mountain, December 1995.

[16] G. Bianchi, and A. T. Campbell, "A Programmable Medium Access Controller for Adaptive Quality of Service Control", *IEEE Journal of Selected Areas in Communications (JSAC)*, Special Issue on Intelligent Techniques in High Speed Networks, March 2000.

[17] J. Biswas, et al., " The IEEE P1520 Standards Initiative for Programmable Network Interfaces” *IEEE Communications Magazine*, Special Issue on Programmable Networks, October, 1998.

[18] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, “An Architecture for Differentiated Services” *Request for Comments 2475*.

[19] S. Borst, O. Boxma and P. R. Jelenkovic, "Generalized Processor Sharing with Long-Tailed Traffic Sources", *In Teletraffic Engineering in a Competitive World*, Proc. ITC-16, Edinburgh, UK, eds. P. Key, D. Smith (North-Holland, Amsterdam), pp. 345-354, 1999.

[20] V. Bose, M. Ismert, W. Wellborn, and J. Guttag, “Virtual Radios”, *IEEE Journal on Selected Areas in Communications*, Special Issue on Software Radios, 1998.

[21] V. Bose, D. Wetherall, and J. Guttag, “Next Century Challenges: Radioactive Networks”, *Fifth ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM’99)*, Seattle, Washington, 1999.

[22] R. Braden, “Active Signaling Protocols”, *Active Networks Workshop*, Tucson AZ, March 1998.

[23] R. Braden, D. Clark, and S. Shenker, "Integrated services in the internet architecture: an overview," *Request for Comments (Informational) 1633*, Internet Engineering Task Force, June 1994.

[24] K. Buchanan, R. Fudge, D. McFarlane, T. Phillips, A. Sasaki, and H. Xia, “IMT-2000: Service Provider’s Perspective”, *IEEE Personal Communications Magazine*, August 1997

[25] M. M. Buddhikot, S. Suri, and M. Waldvogel. “Space decomposition techniques for fast layer-4 switching,” *Proceedings of Conference on Protocols for High Speed Networks*, pages 25-41, August 1999.

[26] R. Caceres, and V. Padmanabhan, “Fast and scalable wireless handoffs, in support of mobile Internet audio”, *Mobile Networks and Applications*, 1998.

[27] K. Calvert, et. al, “Directions in Active networks”, *IEEE Communications Magazine*, Special Issue on Programmable Networks, October 1998.

[28] A.T. Campbell, J. Vicente and D. A. M. Villela, "Virtuosity: Performing Virtual Network Resource Management", *International Workshop on Quality of Service (IWQOS'99)*, London, June 1999.

[29] A. T. Campbell, M. E. Kounavis, J. Vicente, D. A. M. Villela, K. Miki and H. G. De Meer, “A Survey of Programmable Networks”, *ACM SIGCOMM Computer Communication Review*, Vol. 29, No. 2, pp. 7-24, April 1999.

[30] A. T. Campbell, J. Gormez, J., S. Kim, A. Valko, C. Wan, Z. Turanyi, “Design Implementation, and Evaluation of Cellular IP”, *IEEE Personal Communications*, vol. 7 No. 4, pg. 42-49, Aug 2000

[31] A. T. Campbell, S. Chou, M. E. Kounavis, V. D. Stachtos, and J. Vicente, "NetBind: A Binding Tool for Constructing Data Paths in Network Processor-based Routers", *IEEE OPENARCH 2002*, to be presented.

[32] A. T. Campbell, M. E. Kounavis, and R. R.-F. Liao, "Programmable Mobile Networks", *Computer Networks,* Vol. 31, No. 7, pg. 741-765, April 1999.

[33] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela, “The Genesis Kernel: A Virtual Network Operating System for Spawning Network Architectures”, *Second International Conference on Open Architectures and Network Programming (OPENARCH)*, New York, 1999.

[34] A.T. Campbell and G. Bianchi "A Programmable MAC Framework for Utility-based Adaptive Quality of Service Support", *IEEE Journal of Selected Areas in Communications (JSAC)*, Special Issue on Intelligent Techniques in High Speed Networks, Vol. 18, No. 2, pp. 244-256, February 2000.

[35] CANEs: Composable Active Network Elements", http://www.cc.gatech.edu/ projects/canes/

[36] The Cellular IP Project Home Page and Source Code Distribution, comet.columbia.edu/cellularip.

[37] M. C. Chan, A. A. Lazar and R. Stadler, "Customer Management and Control of Broadband VPN Services", *Proc. Fifth IFIP/IEEE International Symposium on Integrated Network Management*, San Diego, CA, May 1997.

[38] M. C. Chan, J.-F. Huard, A. A. Lazar, and K.-S. Lim, "On Realizing a Broadband Kernel for Multimedia Networks", *3rd COST 237 Workshop on Multimedia Telecommunications and Applications*, Barcelona, Spain, November 25-27, 1996.

[39] M. C. Chan, and A. A. Lazar, "Designing a CORBA-based High Performance Open Programmable Signaling System for ATM Switching Platforms", *IEEE Journal on Selected Areas in Communications*, September 1999.

[40] P. Chandra, et al., "Darwin: Customizable Resource Management for Value-added Network Services", *Sixth IEEE International Conference on Network Protocols (ICNP'98)*, Austin, October 1998.

[41] K. G. Coffman and A. M. Odlyzko "Growth of the Internet", *In Optical Fiber Telecommunications* IV, I. P. Kaminow and T. Li, eds. Academic Press, 2001.

[42] G. Coulson, et al., "The Design of a QOS-Controlled ATM-Based Communications System in Chorus", *IEEE Journal of Selected Areas in Communications*, vol.13, no.4, May 1995.

[43] DARPA Active Network Program, http://www.sds.lcs.mit.edu/darpa-activenet/

[44] S. Da Silva, Y. Yemini, and D. Florissi, "The NetScript Active Network System", *IEEE Journal on Selected Areas in Communications*, Vol. 19, No 3, March 2001.

[45] D. Decasper, G. Parulkar, B. Plattner, "A Scalable, High Performance Active Network Node", *IEEE Network*, January 1999.

[46] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, “Router Plugins: A Software Architecture for Next Generation Routers”, *Proc. ACM SIGCOMM’98* Vancouver Canada, 1998.

[47] M. Degermark, A. Brodnik, S. Carlsson, and St. Pink, "Small forwarding tables for fast routing lookups," *in Proc. ACM SIGCOMM*, September 1997, pp. 3—14

[48] L. Delgrossi. and D. Ferrari, “A Virtual Network Service for Integrated-Services Internetworks”, *7th International Workshop on Network and Operating System Support for Digital Audio and Video*, St. Louis, May 1997.

[49] N. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, K. Van der Merwe, “A Flexible Model for Resource Management in Virtual Private Networks”, *Proc. ACM SIGCOMM'99*, Cambridge MA, 1999.

[50] D. R. Engler, M. F. Kaashoek and J. O’Toole, “Exokernel: An Operating System Architecture for Application-Level Resource Management”, *Fifth ACM Symposium on Operating Systems Principles*, Copper Mountain, December 1995.

[51] The Expat XML Parser Toolkit, expat.sourceforge.net

[52] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On Power-Law Relationships of the Internet Topology", *Computer Communication Review*, Volume 29, Number 4, October 1999.

[53] A. Feldman and S. Muthukrishnan. “Tradeoffs for packet classification,” *Proceedings of Infocom*, vol. 3, pages 1193-202, March 2000.

[54] D. C. Feldmeier, at al. “Protocol Boosters”, *IEEE Journal on Selected Areas in Communications*, Special Issue on Protocol Architectures for the 21st Century, 1998The Genesis Project Home Page. Available at http://www.comet.columbia.edu/ genesis

[55] P. Ferguson, and G. Huston, “What is a VPN?”, *OPENSIG'98 Workshop on Open Signalling for ATM, Internet and Mobile Networks*, Toronto, October 1998.

[56] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, *Addison-Wesley Professional Computing Series*, 1995.

[57] The Genesis Project Home Page, http://www.comet.columbia.edu/genesis/

[58] B. Gleeson, A. Lin, J. Heinanen, “A Framework for IP Based Virtual Private Networks”, *draft-gleeson-vpn-framework-00.txt*, internet-draft, work in progress, February 1999.

[59] D. J. Goodman, "Wireless Personal Communications Systems", *Addison-Wesley Wireless Communications Series,* 1997.

[60] P. Goyal, P., H. Vin, and H. Cheng, "Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks", *IEEE/ACM Transactions on Networking*, Vol. 5, No. 5, pp. 690-704, October 1997.

[61] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields", *Proc. Sigcomm, Computer Communication Review*, vol. 29, no. 4, pp 147-60, September 1999, Harvard University.

[62] P. Gupta and N. McKeown, "Packet Classification using Hierarchical Intelligent Cuttings", *Proc. Hot Interconnects VII*, August 99, Stanford. This paper is also available in IEEE Micro, pp 34-41, vol. 20, no. 1, January/February 2000.

[63] P. Gupta and N. McKeown "Algorithms for Packet Classification", *IEEE Network Magazine*, 2001

[64] J. Hartman, et al., "Liquid Software: A New Paradigm for Networked Systems", *Technical Report* 96-11, Dept. of Computer Science, Univ. of Arizona, 1996.

[65] M. Hicks, et al., "PLAN: A Programming Language for Active Networks", *Proc ICFP'98*, 1998.

[66] N.-F. Huang, S.-M. Zhao, and J.-Y.Pan, "A Fast IP Routing Lookup Scheme for Gigabit Switching Routers", *Infocom*, 1999.

[67] X. W. Huang, R. Sharma, and S. Keshav, "The ENTRAPID Protocol Development Environment", *Proc. Eighteenth IEEE International Conference on Computer Communications (INFOCOM'99)*, New York, 1999.

[68] IBM Corporation, IBM PowerNP NP4GS3 Network Processor Datasheet, May 2001.

[69] Intel IXP1200, http://www.intel.com/IXA

[70] Intel Corporation, IXP1200 Network Processor Datasheet, Dec 2000.

[71] Intel Corporation, Intel IXA SDK ACE Programming Framework Developer's Guide, June 2001

[72] Intel Corporation, IXP1200 Network Processor Development Tools User's Guide, Dec 2000.

[73] E. J. Johnson, and A. R. Kunze, "IXP1200 Programming", Chapter 13, *Intel Press*, 2002.

[74] S. Karlin, and L. Peterson, "VERA: An Extensive Router Architecture", *In Proceeding of the 4th International Conference on Open Architectures and Network Programming*, pg 3-14, April 2001

[75] S. Keshav "An Engineering Approach to Computer Networking", Addison Wesley, 1997

[76] S. Keshav and R. Sharma, "Issues and Trends in Router Design", *IEEE Communications Magazine,* May 1998

[77] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, A. S. Tomkins, "The Web as a Graph: Measurements, Models, and Methods", *Proceedings of the 5th Annual International Conference on Computing and Combinatorics*, 1999.

[78] Korilis, Y. A., Lazar, A.A. and Orda, A., "Achieving Network Optima Using Stackelberg Routing Games*", IEEE Transactions on Networking*, Vol. 5, No. 1, February 1997, pp. 161-173.

[79] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. Kaashoek, "The Click Modular Router", *ACM Transactions on Computer Systems* 18(3), Aug 2000, pg 263-297.

[80] M. E. Kounavis, A. T. Campbell, S. Chou, F. Modoux, J. Vicente and H. Zhuang, "The Genesis Kernel: A Programming System For Spawning Network Architectures", *IEEE Journal on Selected Areas in Communications*, Vol. 19, No 3, pg. 511-526, March 2001.

[81] M. E. Kounavis, A. T. Campbell, G. Ito, and G. Bianchi, "Accelerating Service Creation and Deployment in Mobile Networks", *Third International Conference on Open Architectures and Network Programming, (OPENARCH'00)*, Tel-Aviv, Israel, March 2000.

[82] M. E. Kounavis, A. T. Campbell, G. Ito, and G. Bianchi, "Supporting Programmable Handoff in Mobile Networks", *Sixth International Workshop on Mobile Multimedia Communications (MoMuC'99)*, San Diego, CA, November 1999.

[83] M. E. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. T. Campbell, "Directions in Packet Classification for Network Processors", *Second, Workshop on Network Processors (NP2),* Anaheim, California, 2003.

[84] A. B. Kulkarni, G. J. Minden, R. Hill, Y. Wijata, A. Gopinath, S. Sheth, F. Wahhab, H. Pindi, and A. Nagarajan, "Implementation of a Prototype Active Network", *First International Conference on Open Architectures and Network Programming (OPENARCH)*, San Francisco, 1998.

[85] 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.

[86] T.V. Lakshman and D. Stiliadis. “High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching”, *Proceedings of ACM Sigcomm*, pages 191-202, September 1998.

[87] A. A. Lazar, S. Bhonsle, and K. S. Lim, "A Binding Architecture for Multimedia Networks", *Journal of Parallel and Distributed Computing*, Vol. 30, No. 2, November 1995, pp. 204-216.

[88] A. A. Lazar and A. T. Campbell, “Spawning Network Architectures”, *White Paper*, Center for Telecommunications Research, Columbia University, comet.columbia.edu/genesis, January 1998.

[89] A.A. Lazar, "Scaling in Networks", available at http://comet.columbia.edu/courses/ee_e9701/2001/overview.html

[90] A.A. Lazar, K.S Lim, and F. Marconcini, "Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture", *Journal of Selected Areas in Communications*, Vol.14, No.7, September 1996, pp. 1214-1227.

[91] A. A. Lazar, “Programming Telecommunication Networks”, *IEEE Network*, vol.11, no.5, September/October 1997. .

[92] W. E. Leland, M. S. Taqqu, W. Willinger and D. V. Wilson, "On the self-similar nature of Ethernet traffic", *IEEE/ACM Trans. Networking* 2 (1994), 1--15.

[93] R.R.-F. Liao, and A.T Campbell,. "Dynamic Core Provisioning for Quantitative Differentiated Service", *9th International Workshop on Quality of Service (IEEE/ACM/IFIP IWQOS 2001)*, Karlsruhe, Germany, June 2001.

[94] R. R.-F. Liao, and A.T. Campbell, "On Programmable Universal Mobile Channels in a Cellular Internet", *Fourth ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM'98)*, Dallas, October 1998.

[95] R. R.-F. Liao, P. Bocheck, A. T. Campbell, and S.-F. Chang, “Utility-based network adaptation in MPEG-4 systems”, *Ninth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV’99)*, Basking Ridge, New Jersey, 1999.

[96] L. K. Lim, J. Gao, T. S. E. Ng, P. Chandra, P. Steenkiste, H. Zhang “Customizable Virtual Private Network Service with QoS“, Computer Networks, 1999

[97] S- L Lo, and D. Riddoch, “The OmniORB2 version 2.7.1 User’s Guide”, *Technical Report*, AT&T Laboratories Cambridge, 1999.

[98] J. Mitola, “Technical Challenges in the Globalization of Software Radio”, IEEE Communications Magazine, February 1999.

[99] J. Mitola, “Cognitive Radio for Flexible Mobile Multimedia Communications”, *Sixth International Workshop on Mobile Multimedia Communications (MoMuC’99)*, San Diego, CA, November 1999.

[100] A. Montz, D. Mosberger, S. O’Malley, L. Peterson, and T. Proebsting, “Scout, A Communication Oriented Operating System”, *Operating System Design and Implementation*, 1994.

[101] Mobiware Toolkit v1.0 source code distribution http://www.comet.columbia.edu/ mobiware

[102] M. Nandikesan, “On the Foundations of Network Programmability”, *Ph.D Thesis*, Comet Group, 2001.

[103] Network Processing Forum,http://www.npforum.org

[104] P. Newman, W. Edwards, R. Hinden, E. Hoffman, C. F. Liaw, T. Lyon, and G. Minshall, "Ipsilon's General Switch Management Protocol Specification," *Request For Comments 1987*, Aug. 1996

[105] The NetBind Project home page, available at: http://www.comet.columbia.edu/genesis/ netbind

[106] The Object Management Group, www.omg.org.

[107] Object Management Group, “Minimum CORBA”, Joint Revised Submission*, OMG Document*, orbos/98-08-04 ed., August 1998.

[108] Open Signaling Working Group comet.columbia.edu/opensig/

[109] P. Pappu P. and T. Wolf. “Scheduling processing resources in programmable routers”, *Proceedings of the Twenty-First IEEE Conference on Computer Communications (INFOCOM),* New York, NY, June 2002.

[110] A. K. Parekh and R. G. Gallager. "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case", *IEEE/ACM Transactions on Networking*, 1(3):344--357, June 1993.

[111] V. Paxson, and S. Floyd, "Wide-Area Traffic: The Failure of Poisson Modeling", *IEEE/ACM Transactions on Networking*, Vol. 3 No. 3, pp. 226-244, June 1995.

[112] V. Paxson and S. Floyd, "Difficulties in Simulating the Internet", *IEEE/ACM Transactions on Networking*, February, 2001.

[113] D. Pendarakis, "On the Tradeoff between Signaling and Transport in Broadband Networks", *Ph.D Thesis* COMET Group, Columbia University, 1996.

[114] L. Peterson, "NodeOS Interface Specification", *Technical Report*, Active Networks NodeOS Working Group, February 2, 1999

[115] J. Postel, Editor, "Internet Protocol", *Request For Comments 791*, September 1981.

[116] A. Prakash, and A. Aziz, "OC-3072 Packet Classification Using BDDs and Pipelined SRAMs", *Hot Interconnects*, 2001

[117] C. Pu, C., H. Massalin and J. Ioannidis, "The Synthesis Kernel", *Springer Verlag*, 1988.

[118] R. Ramjee, T. La Porta, S. Thuel, K. Varadhan, "HAWAII: A Domain-based Approach for Supporting Mobility in Wide-area Wireless Networks", *Seventh International Conference on Network Protocols*, Toronto, Canada, 1999.

[119] J. Rexford, A. Greenberg and F. Bonomi, "Hardware Efficient Fair Queuing Architectures for High-Speed Networks" *In Proceedings, IEEE INFOCOM,* March 1996.

[120] J. P. Redlich, M. Suzuki, and S. Weinstein, "Virtual Networks in the Internet", *In Proceedings, Second International Conference on Open Architectures and Network Programming (OPENARCH)*, New York, 1999.

[121] D. Schmidt, and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware", *IEEE Communications Magazine*, Special Issue on Design Patterns, April, 1999.

[122] B. Schwartz, W. A. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge, "Smart Packets for Active Networks", *Second International Conference on Open Architectures and Network Programming (OPENARCH)*, New York, 1999.

[123] S. Seshan, H. Balakrishnan, and R. H. Katz, "Handoffs in Cellular Networks: The Daedalus Implementation and Experience", *Kluwer International Journal on Wireless Communication Systems*, 1996.

[124] N. Semret, R. Liao, A.T. Campbell, and A.A. Lazar, "Pricing, Provisioning and Peering: Dynamic Markets for Differentiated Internet Services and Implications for Network Interconnections", *IEEE Journal on Selected Areas in Communications*, Vol. 18, Number 12, December 2000, pp. 2499-2513.

[125] F. Shafai, K.J. Schultz, G.F. R. Gibson, A.G. Bluschke and D.E. Somppi. "Fully parallel 30-Mhz, 2.5 Mb CAM," *IEEE Journal of Solid-State Circuits*, vol. 33, no. 11, November 1998.

[126] Signal Engines Project, comet.columbia.edu/signalingengines

[127] B. C. Smith, "Procedural Reflection in Programming Languages", *PhD Thesis*, Massachusetts Institute of Technology, 1982.

[128] T. Spalink, S. Karlin and L. Peterson, "Evaluating Network Processors in IP Forwarding", *Technical Report* TR-626-00, Nov 15, 2000

[129] T. Spalink, S. Karlin, L. Peterson and Y. Gottlieb, "Building a Robust Software-Based Router Using Network Processors", *In Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pg. 216-229, Oct 2001

[130] V. Srinivasan, S. Suri, and G. Varghese. "Packet Classification using Tuple Space Search", *Proceedings of ACM Sigcomm*, pages 135-46, September 1999.

[131] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel. "Fast and Scalable Layer four Switching," *Proceedings of ACM Sigcomm*, pages 203-14, September 1998.

[132] I. Stoica, H. Zhang, and T. S. E. Ng, "A Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Service, *in Proceedings of SIGCOMM'97*, Cannes, France, 1997, pp. 249-262.

[133] D. Taylor, J. Turner and J. Lockwood "Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware for High-Performance Programmable Routers", *IEEE Open Architectures and Network Programming*, April 2001.

[134] D. Tennenhouse, and D. Wetherall, "Towards an Active Network Architecture", Proceedings, *Multimedia Computing and Networking*, San Jose, CA, 1996.

[135] D. Tennenhouse, et al., "A Survey of Active Network Research", *IEEE Communications Magazine*, January 1997.

[136] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska, “Implementing Network Protocols at User Level”, *IEEE/ACM Transactions on Networking*, October 1993.

[137] J. Touch, J. and S. Hotz, "The X-Bone", *Third Global Internet Mini-Conference in conjunction with Globecom '98* Sydney, Australia, November 1998.

[138] N. D. Tripathi, J. H. Reed, and H. F. Vanlandingham, “Handoff in Cellular Systems”, *IEEE Personal Communications Magazine*, December 1998.

[139] P. Tsuchiya. “A search algorithm for table entries with non-contiguous wildcarding,” *unpublished report*, Bellcore

[140] A. G. Valko, J. Gomez, S. Kim, and A. T. Campbell, "On the Analysis of Cellular IP Access Networks", *Sixth IFIP International Workshop on Protocols for High Speed Networks* , Salem, 25-27 August 1999.

[141] K. Van der Merwe, and I. M. Leslie, “Switchlets and Dynamic Virtual ATM Networks”, *Proc Integrated Network Management V*, May 1997.

[142] K. Van der Merwe, S. Rooney, I. M. Leslie and S. A. Crosby, “The Tempest - A Practical Framework for Network Programmability”, *IEEE Network*, November 1997

[143] A. Veres, Z. Kenesi, S. Molnar, G. Vattay, "On the Propagation of Long-Range Dependence in the Internet", Proc. ACM SIGCOMM 2000, Stockholm, Sweden, Sep. 2000.

[144] S. Vinoski,“CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments”, *IEEE Communications Magazine*, Vol. 14, No. 2, February, 1997.

[145] H. J. Wang, R. H. Katz, and J. Giese, "Policy-Enabled Handoffs Across Heterogeneous Wireless Networks," *Second IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '99)*, New Orleans, LA, February 1999.

[146] D. J. Watts and S. H. Strogatz, "Collective Dynamics of Small-World Networks", *Nature*, Vol. 393, pp. 440-442, 1998.

[147] D. Wetherall, J. Guttag and D. Tennenhouse, “ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols”, *Proc. IEEE OPENARCH'98*, San Francisco, CA, April 1998.

[148] T. Wolf and J. Turner, “Design Issues for High- Performance Active Routers”, *IEEE Journal on Selected Areas in Communications*, March 2001.

[149] xbind code http://comet.columbia.edu/xbind

[150] Xbind Inc., www.xbind.com

[151] Y. Yemini, and S. Da Silva "Towards Programmable Networks", *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, October, 1996.

[152] A. Yun, A. Leon-Garcia, and M. Jaseemuddin, “Virtual Networks: A Divide-and-Conquer Approach to Network Resource Management”, *Proc. Open Signaling for ATM, Internet and Mobile Networks (OPENSIG) Workshop*, New York, October 1997.

[153] L. Zhang, "Virtual clock: A new traffic control algorithm for packet switched networks," *Proc. ACM Trans. on Comp. Systems*, May 1991, pp. 101-125.