The Genesis Kernel: A Programming System for Spawning Network Architectures

Michael E. Kounavis, Andrew T. Campbell, Stephen Chou, Fabien Modoux, John Vicente and Hao Zhuang

***Abstract*—Currently, the design, deployment and refinement of new network architectures is a manual, ad-hoc and time-consuming process. We present the design, implementation and evaluation of the Genesis Kernel, 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 that is capable of creating distinct virtual network architectures on-demand. The Genesis Kernel is based on a methodology that allows a child virtual 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. These spawned network architectures include a parent network that supports IP forwarding, and interior and exterior routing. We discuss how two child networks based on Cellular IP and Mobiware architectures can be spawned on the parent network to support wireless access to data and continuous media services, respectively.**

***Keywords*—programmable virtual networking, spawning, service creation**

I. INTRODUCTION

Existing network architectures (e.g., Internet, mobile, Telephone, ATM) exhibit a lack of intrinsic architectural flexibility in adapting to new user needs and requirements. We broadly define network architecture as a set of transport, signaling, control and management mechanisms that are governed by operational time-scales and state information. Typically, network architectures are realized as a set of distributed network algorithms that offer services to end-systems. In what follows, we make a number of observations about the limitations encountered when designing and deploying new network architectures.

First, current network architectures are deployed on top of a multitude of technologies such as land-based, wireless, mobile and satellite for a wide array of voice, video and data applications. Because these architectures offer a limited capability to match the evolving needs of new applications and environments, the deployment of these architectures has predictably met with various degrees of success.

Second, the interface between the network and the service architecture responsible for the basic communication services (such as connection setup procedures in Telephone and ATM networks) is rigidly defined and can not be replaced, modified, nor supplemented. In other cases, for example the Internet, end user connectivity abstractions provide little support for quality of service (QOS) guarantees and accounting for usage of network resources (billing).

Third, the creation and deployment of network architectures is a manual, time consuming and costly process. At its most advanced, the creation process utilizes off-line tools for network planning, emulation and simulation. These tools are, however, invariably narrow in scope, primitive in use and fail to highlight architectural shortcomings. To the network architect the creation process is typically ad-hoc in nature, based on hand crafting small-scale prototypes that evolve toward wide scale deployment. We believe that there is a need to design architectures based on solid theoretical foundations that call for clearly reasoned system level models. Fourth, multiple parameterizations of the network design space are needed and should be used for systematic exploration before the final realization of the architecture is deployed. Such capabilities hardly exist and as a result the deployment cycle is typically a blind iterative process based on 'design, deploy and analyze'.

In response to these limitations, we argue that there is a need to propose, investigate, and evaluate alternative network architectures to the existing ones. This challenge goes beyond a proposal for yet another experimental network architecture. Rather, it calls for new approaches to the way we design, develop, deploy, observe and analyze new network architectures in response to future needs and requirements. We believe that the design, deployment and management of new network architectures should be automated and built on a foundation of *spawning networks*, a new class of open programmable networks. In [1] we describe the process of automating the creation and deployment of new network architectures as 'spawning'. The term spawning finds a parallel with an operating system spawning a child process. We envision spawning networks as having the capability to spawn not processes but complex network architectures.

Spawning networks support the deployment of programmable virtual networks. We call a virtual network

Michael E. Kounavis, Andrew T. Campbell and Stephen Chou ({mk, campbell, schou}@comet.columbia.edu) are affiliated with the COMET Group, Center for Telecommunications Research, Columbia University. Fabien Modoux (modoux@comet.columbia.edu) works for Voicemate Inc., New York, USA. John Vicente (John.Vicente@intel.com) is affiliated with Intel Corporation, USA. Hao Zhuang (hzhuang@celoxnetworks.com) is a member of the technical staff at Celox Networks, USA.

This work is supported in part by the NSF CAREER Award No. ANI 9876299 and with support from the Intel Corporation, Nortel Networks and Hitachi Limited.

installed on top of a set of network resources a 'parent virtual network'. We propose the realization of parent virtual networks[1] with the capability of creating 'child virtual networks' operating on a subset of network resources and topology, as illustrated in Figure 1. 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, as illustrated in Figure 1. 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 paper, 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. The Genesis Kernel is an enabling technology for spawning networks that automates the virtual network life cycle process, which comprises *profiling*, *spawning*, *management* and *architecting*. 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 [4] 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 networks 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 [2] and Mobiware [3] 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 [13] and [14] 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 paper is structured as follows. In Section II we describe the Genesis Framework and discuss the principles that underpin spawning networks. Following this, in Section III we describe our prototype implementation. In Section IV we present our experiences with using the Genesis Kernel, focusing on the dynamic creation, deployment and management of the baseline network architectures. In Section V, we present related work in the area of programmable networks. Finally, we present some concluding remarks in Section VI.

[1] We abbreviate the terms parent virtual network and child virtual networks to parent network and child network, respectively. The term virtual network and spawned virtual network are synonymous and refer to parent or child networks.

II. The Genesis Kernel

A. Genesis Framework

Three distinct levels of the Genesis Kernel support spawning, as illustrated in Figure 2. 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 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 2. 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* [6] 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:

- *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

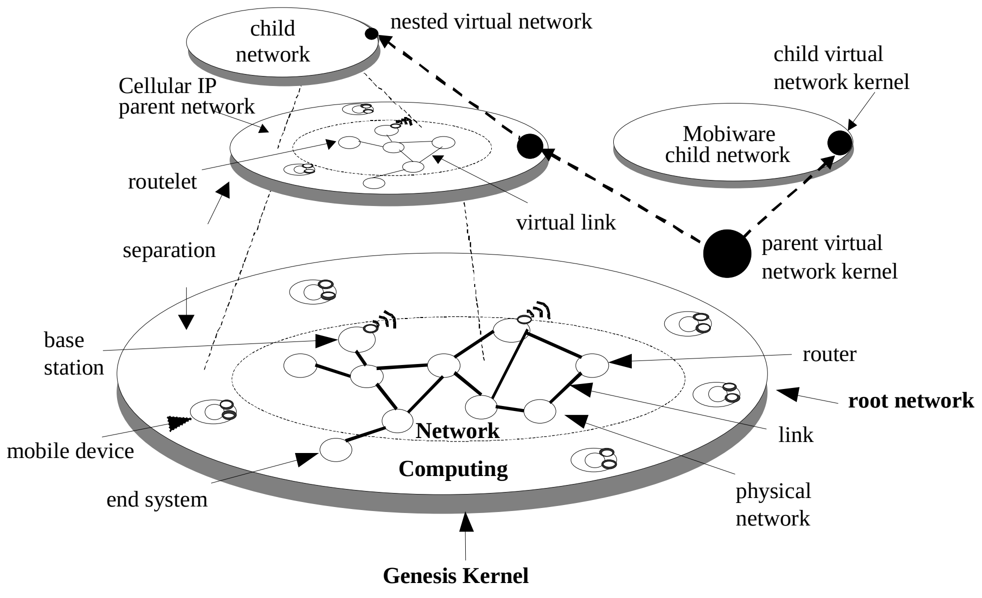


Figure 1: Spawning Networks

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 2, 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).

B. 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.
- *Nesting Principle:* A child network inherits the capability to spawn other virtual networks creating the notion of 'nested virtual networks' within a virtual network, as illustrated in Figure 1. 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' [4], as illustrated in Figure 1. In this spawning scenario, the inheritance tree is formed by the virtual network at the 'root' of the tree, which spawns two child networks. One child network (i.e., the Cellular IP virtual network) is a parent to its own child network. We call the virtual network at the root of the inheritance tree the *root network*, as illustrated in Figure 1.
- *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

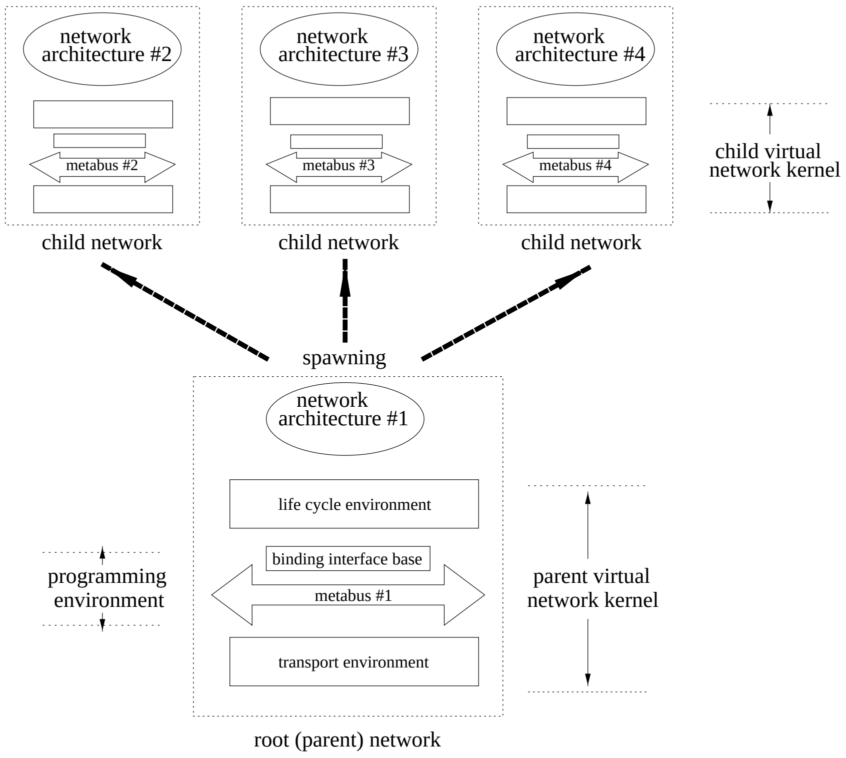


Figure 2: The Genesis Framework

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.

C. *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 3. 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.

1) *Ports and Engines*

Ports and engines, shown in Figure 3, 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 datapaths.

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 [5] 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.

2) *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).

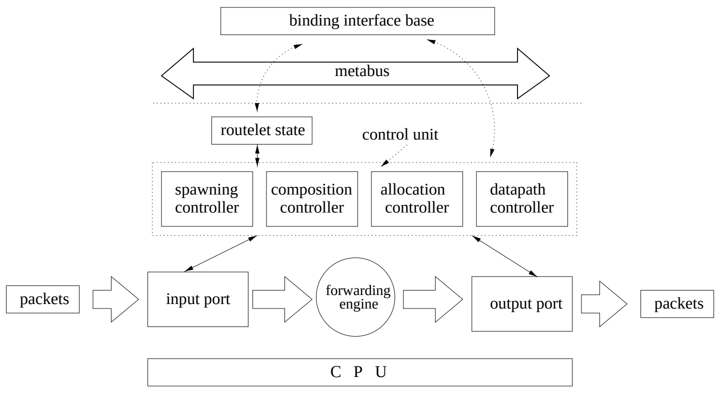


Figure 3: Routelet Architecture

Routelets generalize the concept of partitioning physical switch resources introduced in [6] and [7]. 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.

3) *Nested Routelets*

Nested routelets operate on the same physical node and maintain their structure according to a virtual network inheritance tree, as discussed in Section II-B. 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 4. 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’ [4] 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.

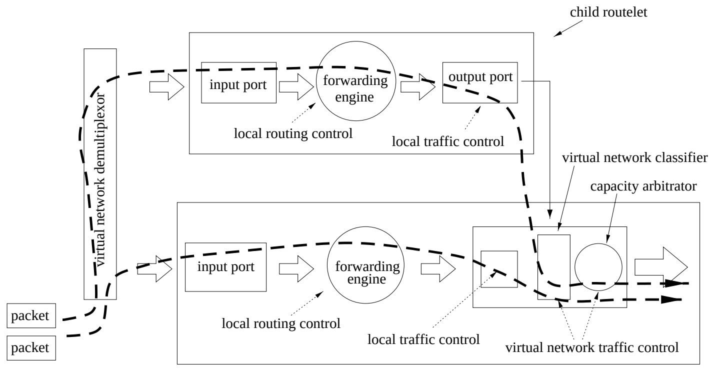


Figure 4: Nested Routelets

4) *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. 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).

D. Programming Environment

Each network architecture comprises a set of distributed controllers that realize communication algorithms (e.g., routing, control, management), as discussed in Section II. 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 5 and discussed below.

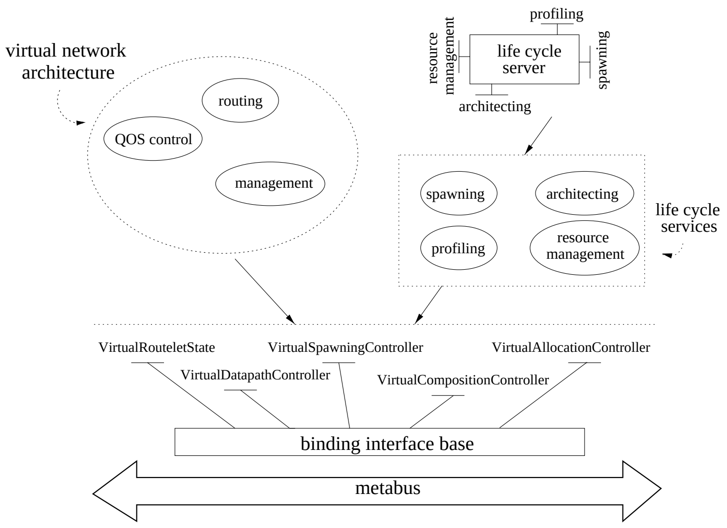


Figure 5: Programming Environment

1) 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 [8] 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 [3] [6] that use CORBA technology for service creation, signaling and management in previous projects. 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 III-B 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.

2) 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.

E. *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.

1) *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 definition of the communication services and protocols that characterize a network architecture. 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 virtual networks. 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. The network architect can 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. Similarly, QOS architectures based on well-founded models (e.g., integrated services [10] and differentiated services [5]) can be dynamically selected and used for QOS provisioning in virtual networks. Transport protocols (e.g., TCP, RTP, UDP) and network management components (e.g., SNMP, CMIP) made available as software building blocks can be instantiated on-demand.

2) *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.

3) *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.

The Genesis virtual network resource management system called *virtuosity* [4] leverages the benefits of the kernel's hierarchical model of inheritance and nesting. Virtual network resources are provisioned based on 'policy' and slow time-scale resource re-negotiation. As a result, parent networks manage child traffic in an aggregated and scalable manner using general purpose capacity classes. Virtual network resources are controlled on slow performance management timescales (e.g., possibly in the order of tens of minutes). We argue that this is a suitable timescale for the resource management system to operate over while allowing virtual networks to perform dynamic provisioning as needed. A full description of virtuosity is outside the scope of this paper. For details on the virtuosity framework and its performance evaluation see [4] and [21], respectively.

4) *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.

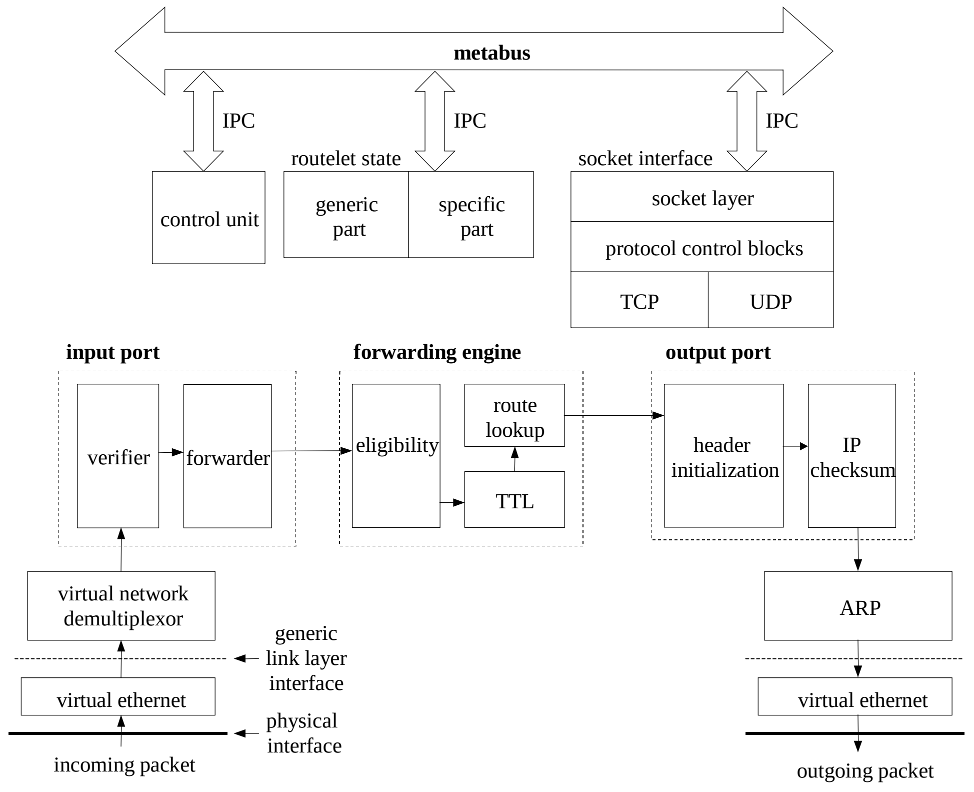


Figure 6: IPv4 Routelet Implementation

III. Implementation

We have been developing the kernel since the Spring of 1998 and have completed the implementation of Genesis Kernel v1.0 [35]. Using a set of foundation objects and services, we have been able to profile, spawn, manage and architect a simple set of baseline architectures. The kernel represents a partial implementation of the Genesis Framework discussed in the previous section. The transport and programming environments have been implemented using commodity operating systems and distributed systems technology. There remain a number of technical barriers to realizing the lifecycle service capability, particularly in the areas of profiling and architecting virtual networks. In addition, virtuosity is not implemented as part of the current kernel release. However, we have implemented virtuosity and evaluated its virtual network resource management capability as extensions to the *ns* simulator. For full details on the virtuosity ns extensions and performance see [21].

A. *Transport Environment*

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 [12] 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 [16]. In addition, we modified the Mobiware and Cellular IP software distributions [13] [14] to create child network architectures. Figure 6 illustrates the implementation of an IPv4 routelet.

1) *Link Layer Support*

The Genesis Kernel separates the link and internetworking layers through a generic link layer interface, as illustrated in Figure 6. 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 a 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 6, 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. We have also implemented virtual WaveLAN and ATM modules.

2) *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 specifying the manner in which packets should be received. For example, a virtual network demultiplexor can configure a virtual ATM module to receive packets from a specific set of PVCs.

3) *Routelet Components*

Routelet components are shown in Figure 6. 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.

4) *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.

5) *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.

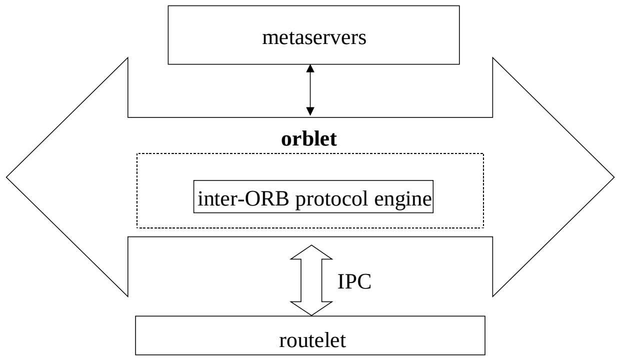


Figure 7: Metabus Architecture

B. *Programming Environment*

1) *Metabus*

The metabus comprises an orblet component and set of metaservers, as illustrated in Figure 7. 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 [19] 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.

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 [18] 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.

2) *Binding Interface Base.*

The binding interface base shown in Figure 5 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 Genesis Kernel v1.0.

The *VirtualRouteletState* and *VirtualDatapathController* interfaces illustrated in Figure 5 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.

C. *Life cycle environment*

1) *Profiling Service*

The Genesis Kernel v1.0 only supports a subset of the virtual network requirements discussed in Section II. 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 8. 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.

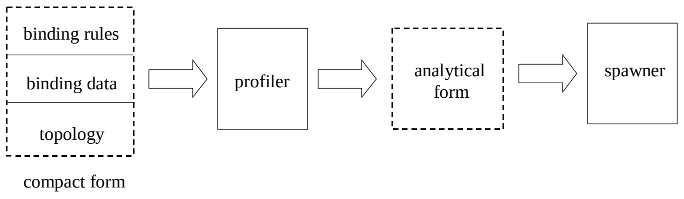


Figure 8: Profiling Process

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 to spawn 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.

```
<?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 9: Profiling Script: A Snippet of the Binding Rules for the Cellular IP Network Architecture. See [35] for a Full Specification.

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 virtuosity system [4] to allocate link resources for child networks. Currently, we have not addressed topology conversion and resource management issues in the Genesis Kernel v1.0. 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).

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 XML4C from IBM and Alphaworks [20].

Figure 9 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.

2) *Spawning*

The spawning process 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 [4] and represents a distributed controller, which performs admission testing for child networks. If the admission test is successful the child network is spawned. For details on virtual network admission control see [21].

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 [4]. Virtual network capacity scheduling is currently being investigated [21]. 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.

IV. 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.

A. Spawning Networks Testbed

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 10. Once the root (parent) network was boostrapped onto the network hardware, we were able spawn the Cellular IP and Mobiware 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.

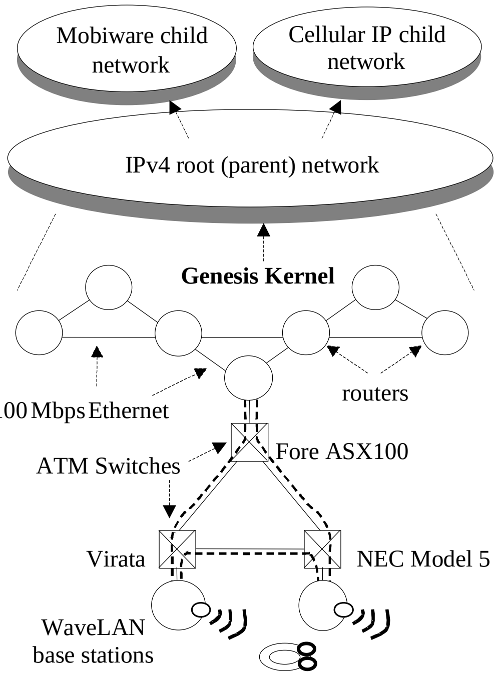


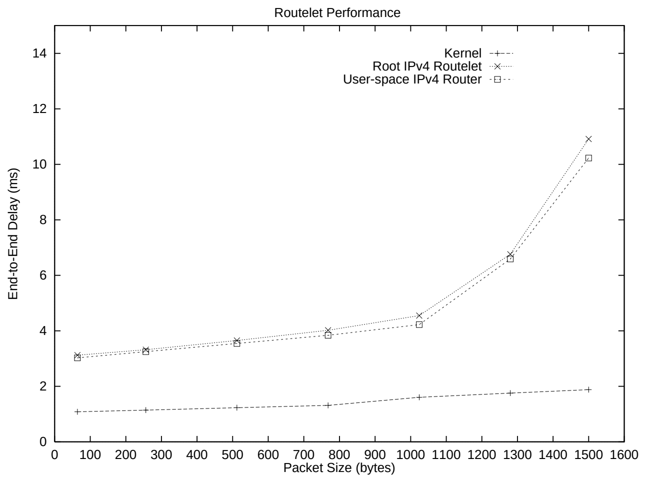
Figure 10: Spawning Networks Testbed

The spawning networks testbed comprises heterogeneous link layer technologies that interconnect routers, switches and base stations, as illustrated in Figure 10. 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 over the 10 Mbps cards because the older cards support a low-level radio utility API for programming beacons.

The spawning capability is currently implemented in the network and not in end-systems. The Genesis Kernel v1.0 code release has been designed to run on Windows NT and FreeBSD operating systems.

B. An IP 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 III-A 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. In order to spawn the IP routing architecture we deviate from the standard spawning procedure described in Section III. 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.



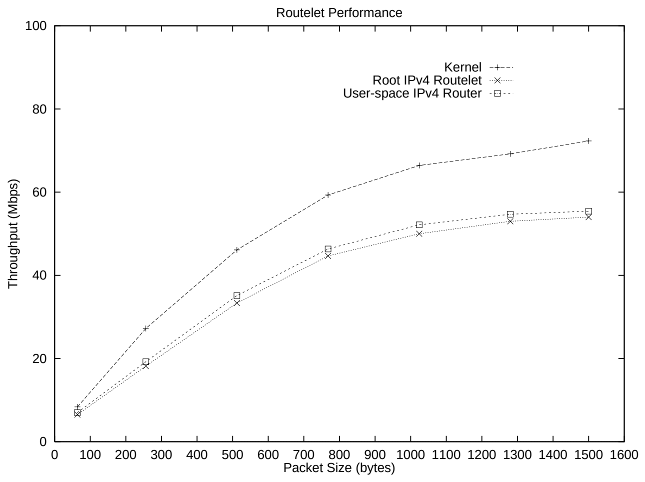


Figure 11: Routelet Performance

In Figure 11, 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 11 shows the end-to-end delay and throughput results across a single hop. We observe that there is little difference in performance between the IPv4 routelet and the standalone user-space IPv4 router. This is a very encouraging result. 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 11 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. Our current work includes porting the Genesis Kernel to the Intel IXA router [38] where the routelet implementation will gain performance from executing on the network processor IXP1200.

C. *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 12.

Mobiware [3] 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 [13]. 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 12. 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 [2] 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 Figure12. Forwarding engines update paging and routing caches inserting a pointer to the downlink path on behalf of the mobile host that sends the packet.

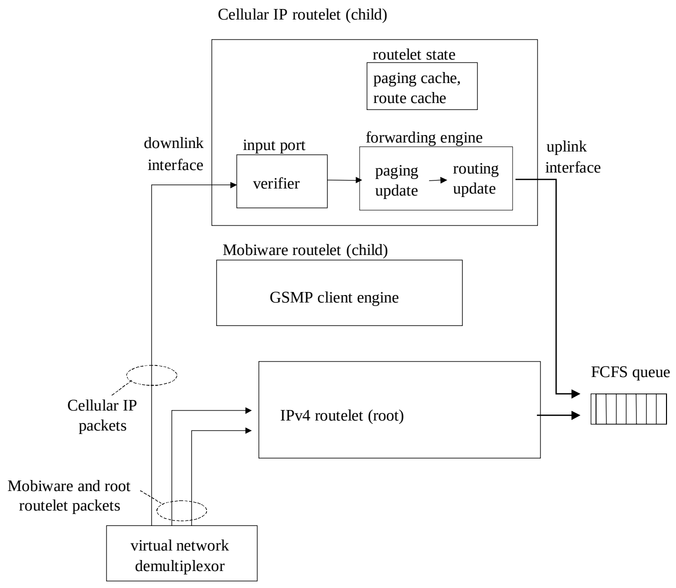


Figure 12: Mobiware and Cellular IP Datapaths

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 paper. For full details of their specification, performance and source code release see [14] and [13], respectively.

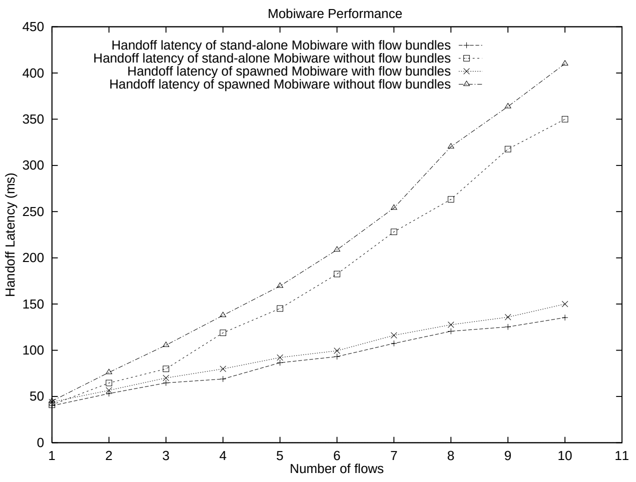
We have conducted a set of tests that compare the performance of the Mobiware and Cellular IP child networks against their 'standalone' counterparts. In all cases the spawning and standalone testbeds were lightly loaded during the experiments. In order to evaluate the Mobiware child network we streamed a number of video flows to a mobile host and performed continuous handoffs, as shown in Figure 13. 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 13. 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.

Throughout measurements are taken for the 'hard' and 'semisoft' Cellular IP handoff modes [2], as shown in Figure 13. 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 13 that the child network achieves similar performance to the standalone network architecture over a wide range of handoff rates.

D. *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.



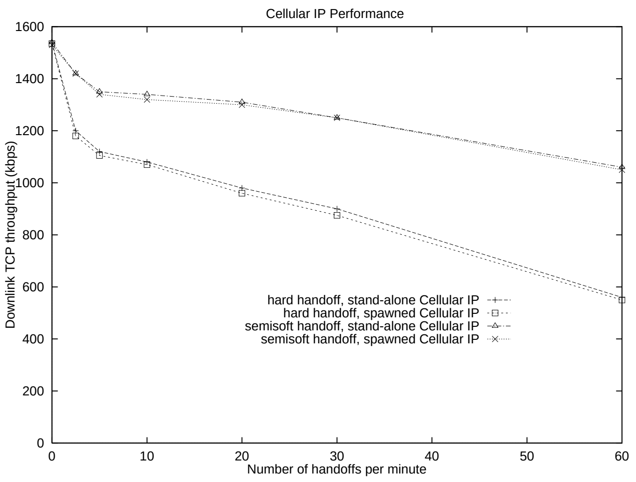


Figure 13: Performance of Spawned Network Architectures

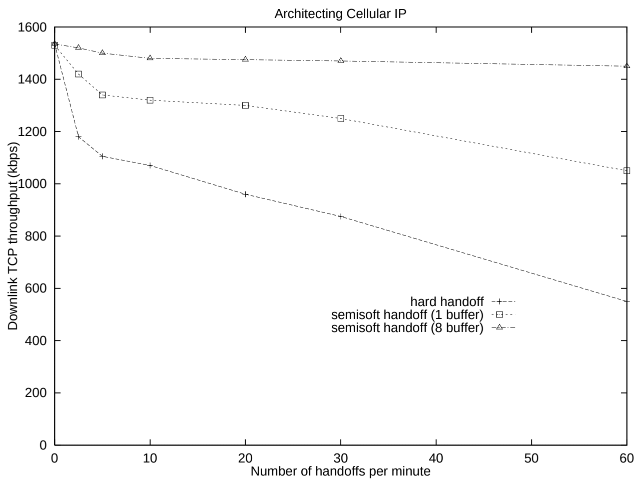


Figure 14: Architecting Cellular IP

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 architectures [23]. 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 14 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 14. 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 out performs 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.

V. Related Work

The Tempest project [7] 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 [6] and switchlets [7].

Virtual private network services have been the subject of a substantial amount of research in broadband ATM networks. In [24], the concept of a virtual path group is introduced as a virtual network building block to simplify virtual path dynamic routing. In [25], 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) [27] is investigating 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 [28] 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 [29] project considers 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 [30]-[34] has investigated the deployment of multiple coexisting execution environments through appropriate operating system support and an active network encapsulation protocols. In [9], 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 investigate ways to construct network architectures that are fundamentally different from their underlying infrastructures.

A traditional challenge in the deployment of virtual private networks has been the separation of traffic and service differentiation between communities of users that share a common infrastructure. Methods for creating virtual and secure private network services include controlled route leaking, Generic Routing Encapsulation, network layer encryption or link layer methodologies for virtualization. These techniques have been used in a variety of commercial products. Finally, a number of IETF proposals have discussed IP virtual private networks [36]. Others have addressed issues of performance [37].

VI. Conclusion

In this paper 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.

A number of challenges remain before we can realize the full potential of spawning networks, however. 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. Exploration of the network design space is one of the most challenging aspects of building spawning networks. We want to provide extensions to the kernel that allows designers to observe, analyze and architecturally refine spawned network architectures. As part of that challenge we are developing a set of well-founded models and tools for observing and refining spawned networks. In this paper, we have discussed some partial refinement techniques for virtual networks. However, we need to better understand architectural refinement; that is, what to modify, how to modify it, and how to measure the impact of what we have modified. Furthermore, we want to be able to do this at run time while the network architecture is executing on the network hardware.

We are porting the Genesis Kernel to the Intel IXA routers based on the Intel programmable network processor IXP1200 as part of a new project on *Signaling Engines* [39]. Currently, we have ported the transport environment to the IXP1200. In the next phase of our work we plan to further develop and evaluate the Kernel focusing on its life cycle environment.

ACKNOWLEDGEMENT

This work is supported in part by the National Science Foundation (NSF) under CAREER Award ANI-9876299 and with the support from the Intel Corporation, Hitachi Limited and Nortel Networks. John. B. Vicente (Intel Corp) would like to thank the Intel Research Council for their support during his visit with the Center for Telecommunications Research, Columbia University. The authors would like to thank Kazuho Miki (Hitachi Ltd.), Hermann G. De Meer (UCL), Daniel A. Villela (Columbia University) and Kalai S. Kalaichelvan (Nortel Networks) for their contributions to the Genesis Project. We would also like to thank the anonymous reviewers for helping to improve this paper and Aurel A. Lazar (xbind Inc./Columbia University) for his support and contribution to spawning networks.

REFERENCES

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

[2] Campbell A. T., Gomez, J., Kim, S., Turanyi, Z., Wan, C-Y. and Valko A, "Design, Implementation and Evaluation of Cellular IP", *IEEE Personal Communications,* Special Issue on IP-based Mobile Telecommunications Networks, 2000.

[3] Campbell, A.T., Kounavis M.E. and R. R.-F. Liao, “Programmable Mobile Networks”, *Computer Networks and ISDN Systems*, April 1999.

[4] Campbell, A.T., Vicente, J., and D.A.M Villela,. “Virtuosity: Performing Virtual Network Resource Management”, *Proc. 7th International Workshop on Quality of Service (IWQOS’99)*, London, May 1999.

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

[6] Lazar, A.A., Lim, K.S. and Marconcini, F., “Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture,” *IEEE Journal on Selected Areas in Communications, Special Issue on Distributed Multimedia Systems*, No. 7, September 1996.

[7] Van der Merwe, J.E., Rooney, S., Leslie, I.M. and Crosby, S.A., “The Tempest - A Practical Framework for Network Programmability”, *IEEE Network*, May 1998.

[8] Vinoski, S., “CORBA, Integrating Diverse Applications Within Distributed Heterogeneous Environments”, *IEEE Communications Magazine*, February 1997.

[9] Da Silva, S., Florissi D., and Yemini Y., “NetScript: A Language-Based Approach to Active Networks”, *Technical Report, Computer Science Dept., Columbia University* January 27, 1998.

[10] Braden, R., Clark, D., and Shenker S., “Integrated Services in the Internet Architecture: an Overview” *Request For Comments 1633*, June 1994.

[11] Campbell, A.T., Kounavis M., Villela D., Vicente, J., De Meer, H., Miki, K., and Kalaichelvan K., “Spawning Networks”, *IEEE Network*, August 1999.

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

[13] The Mobiware Project Home Page and Source Code Distribution, comet.columbia.edu/mobiware.

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

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

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

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

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

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

[20] XML4C, www.alphaworks.ibm.com

[21] Campbell, A.T., Vicente, J., and D.A.M Villela, “Virtuosity: Programmable Resource Management for Spawning Networks” *Computer Networks,* Special Issue on Active Networks, (to be published), 2001.

[22] GateD Consortium, www.gated.org

[23] Kounavis M. E., Campbell A. T., Ito G., and Bianchi G., “Design, Implementation and Evaluation of Programmable Handoff in Mobile Networks”, *ACM/Baltzer Journal on Mobile Networks and Applications*, to be published, 2001.

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

[25] Yun, A., Leon-Garcia, A., and Jaseemuddin, M., “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.

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

[27] Virtual Network Service (VNS), project site: www.cs.cmu.edu/~hzhang/VNS.

[28] Touch, J. and Hotz, S., “The X-Bone”, *Proc. Third Global Internet Mini-Conference in conjunction with Globecom ’98*, Sydney, Australia, November 1998.

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

[30] Tennenhouse, D., and Wetherall, D., “Towards an Active Network Architecture”, *Proc. Multimedia Computing and Networking*, San Jose, CA, 1996.

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

[32] Wetherall, D., Guttag, J. and Tennenhouse, D., “ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols”, *Proc. First International Conference on Open Architectures and Network Programming (OPENARCH)*, San Francisco, CA, April 1998.

[33] Peterson L., “NodeOS Interface Specification “, *AN Node OS Working Group*, February 1999.

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

[35] The Genesis Project Home Page, comet.columbia.edu/genesis.

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

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

[38] Intel Exchange Architecture (IXA), www.intel.com/ixa

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

Michael E. Kounavis is a Ph.D. candidate in the Department of Electrical Engineering, Columbia University, New York. He received a Diploma in Electrical and Computer Engineering from the National Technical University of Athens, Greece in 1996 and a MSc. degree from Columbia in 1998. His main area of research is the development of spawning networks. Over the past years he has been actively involved in mobile network programmability and the realization of adaptive mobile middleware. Michael E. Kounavis has received a Fulbright scholarship in 1996.

Andrew T. Campbell is an Assistant Professor in the Department of Electrical Engineering and member of the COMET Group at the Center for Telecommunications Research, Columbia University, New York. His areas of interest encompass programmable networks, mobile networking and QOS research. He is currently the co-chair for the IEEE Conference on Open Architectures and Network Programming (OPENARCH 01). Andrew T. Campbell received his Ph.D. in Computer Science in 1996 and the NSF CAREER Award for his research in programmable mobile networking in 1999

Stephen Chou is a PhD student in computer science at Columbia University. He has been a senior software engineer at Microprocessor Research Labs at Intel, where he helped to develop functional models to simulate the IA32 and IA64 platforms. His current research interest is on programmable networks and multimedia communications. He received a B.Sc. degree in computer engineering from Carnegie Mellon University and an M.Sc degree in computer science from Georgia Institute of Technology.

Fabien Modoux received a MSc. degree in computer science from Columbia University in 1999, and an engineering degree in communication systems from the Swiss Federal Institute of Technology, Lausanne, Switzerland, and Institute Eurecom, Sophia Antipolis, France, in 1998. He joined Voicemate Inc., New York, NY, in 2000.

John Vicente is a member of Intel's Information Technology organization where he is involved with strategy and technology in the areas of Internet-QOS, policy-based networking, and multimedia and programmable networks. He is also actively engaged in the IEEE P1520 initiative for programmable interfaces for networks. He received his M.S. in Electrical Engineering from the University of Southern California, Los Angeles, CA, and his B.S. in Computer Engineering from Northeastern University, Boston, MA.

Hao Zhuang is currently working at Celox Networks, Inc. He is an MS graduate of the Department of Electrical Engineering of Columbia University. His research interests include broadband networks, traffic statistics and engineering, DiffServ/InteServ, fast routing algorithms and programmable networks.