Spawning Networks

Andrew T. Campbell, Michael E. Kounavis, and Daniel A. Villela
Columbia University
John B. Vicente, Intel Corporation
Hermann G. De Meer, University of Hamburg
Kazuho Miki, Hitachi Ltd.
Kalai S. Kalaichelvan, Nortel Networks

Abstract

The deployment of new network architectures, services, and protocols is often manual, ad hoc, and time-consuming. In this article we introduce "spawning networks," a new class of programmable networks that automate the life cycle process for the creation, deployment, and management of network architectures. These networks are capable of spawning distinct "child" virtual networks with their own transport, control, and management systems. A child network operates on a subset of its "parent's" network resources and in isolation from other spawned networks. Spawned child networks represent programmable virtual networks and support the controlled access to communities of users with specific connectivity, security, and quality of service requirements. In this article we present a framework for the realization of spawning networks based on the notion of the Genesis Kernel, a virtual network operating system capable of creating distinct virtual network architectures on the fly. We discuss the motivation and principles that underpin spawning networks and focus on the design of the transport, programming, and life cycle environments, which comprise the main architectural components of the Genesis Kernel.

The ability to rapidly create, deploy, and manage new network services in response to user demands presents a significant challenge to the research community and is a key factor driving the development of programmable networks. 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, calling for major advances in open network control, network programmability, transportable software, and distributed systems technology. In the near future competition between service providers is likely to hinge on the speed at which one provider can respond to new market demands over another. Existing network architectures such as the Internet, mobile, telephone, and asynchronous transfer mode (ATM) exhibit two key limitations that prevent us from meeting this challenge:

- Lack of intrinsic architectural flexibility in adapting to new user needs and requirements
- Lack of automation of the process of realization and deployment of new and distinct network architectures

In what follows we make a number of observations about the limitations encountered when designing and deploying network architectures. First, current network architectures are deployed on top of a multitude of networking technologies such as land-based, wireless, mobile, and satellite for a bewildering array of voice, video, and data applications. Since these architectures offer a very limited capability to match the many environments and applications, the deployment of these architectures has predictably met with various degrees of success. Tremendous difficulties arise, for example, because of the inability of TCP to match the high loss rate encountered in wireless networks or for mobile IP to provide fast handoff capabilities with low loss rates to mobile devices. Protocols other than mobile IP and TCP operating in wireless access networks might help, but their implementation is difficult to realize. Second, the interface between the network and the service architecture responsible for basic communication services (e.g., connection setup procedures in ATM and telephone networks) is rigidly defined and cannot be replaced, modified, or supplemented. In other cases, such as 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 a network architecture is a manual, time-consuming, and costly process. To help deal with complexity, network designers capture the blueprint that parameterizes the design space into an architecture. Network architectures identify the network hardware and software components, and show how they can be arranged to build a complete environment to support service requirements. Thus, network architectures clearly embody the major building blocks and their interaction. At its most advanced, the creation process utilizes offline tools for network planning, emulation, and simulation. These tools are, however, invariably narrow in scope and primitive in use, and fail to highlight significant integration problems. To the network architect the creation process is typically ad hoc in nature, based on handcrafting small-scale prototypes that evolve toward widescale deployment. 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." While such a trial and error approach is essential when assessing the advantages and disadvantages of selecting a particular subset of the architectural design space, an architectural design methodology is needed to support such capabilities. As a result, the network life cycle process is iterative, lacking systematic exploration of the network design space.

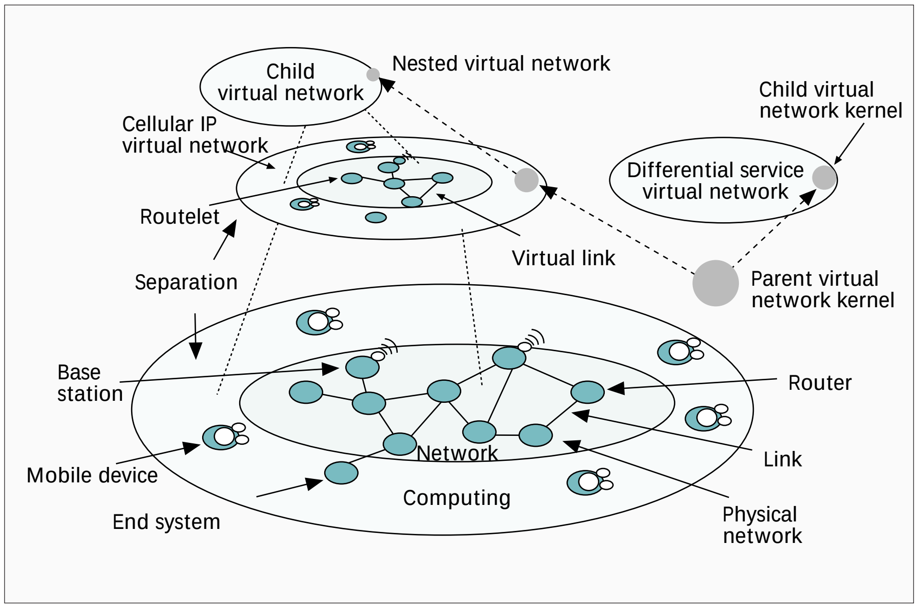


Figure 1. *Spawning networks.*

In response to these limitations, we argue that there is a need to propose, investigate, and evaluate alternative network architectures to the existing ones (e.g., IP, ATM, mobile). This challenge goes beyond the 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, architecting, and management of new network architectures should be automated and built on a foundation of *spawning networks*, a new class of open programmable networks. We describe the process of automating the creation and deployment of new network architectures as *spawning* [1]. The term spawning finds a parallel with an operating system spawning a child process. By spawning a process the operating system creates a copy of the calling process. The calling process is known as the *parent process* and the new process as the *child process*. Notably, the child process inherits its parent's attributes, typically executing on the same hardware (i.e., the same processor). We envision spawning networks as having the capability to spawn not processes but complex network architectures. This is a radical approach to architecting and deploying next-generation networks.

Spawning networks support the deployment of programmable virtual networks. We call a virtual network installed on top of a set of network resources a *parent virtual network*. We propose the realization of parent virtual networks with the capability of creating *child virtual networks* operating on a subset of network resources and topology, as illustrated in Fig. 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 redeployed as a picocellular virtual network that supports fast handoff (e.g., by spawning a Cellular IP [2] virtual network), as illustrated in Fig. 1. In this case the access network is the parent and the Cellular IP network the child. Another example is offered by virtual networks that can be under the control of either a service provider (e.g., an Internet Service Provider, ISP) or the customer. Child networks operate on a subset of the topology of their parents and are restricted by the capabilities of their parents' 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. Typically, spawned network architectures would support alternative signaling protocols, communications services, QoS control, and network management to those of the parent architecture.

In this article we describe a framework for spawning networks based on the design of the *Genesis Kernel*, a virtual network operating system capable of automating a virtual network life cycle process; that is, profiling, spawning, architecting, and managing programmable network architectures on demand. The article is structured as follows. In the next section we provide an overview of programmable networks and discuss the notion of spawning networks in relation to the field. For a comprehensive survey of programmable networks see [3]. Following this, we present an overview of the Genesis Kernel and discuss the principles that underpin spawning networks. We then describe the three main architectural components of the Genesis Kernel framework. We present the detailed design of the transport, programming, and life-cycle environments, respectively. The framework presented in this article offers a systematic methodology for spawning virtual network architectures over the same physical network hardware. The implementation of spawning networks raises a number of engineering and research issues. We discuss open issues associated with implementing the Genesis Kernel framework. Following this, we present the related work and some concluding remarks.

Programmable Networks

A number of research groups are actively designing and developing programmable network prototypes. The Open Signaling (Opensig) community [4] argues that by modeling communication hardware using a set of open programmable network interfaces, open access to switches, routers, and base stations can be provided, thereby enabling third-party software providers to enter the market for telecommunications software. The Opensig community argues that by opening up the network devices in this manner, the development of new and distinct architectures and services can be more easily 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 with QoS.

The Active Network community [5, 30] advocates 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 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* [6] comprise executable programs, consisting of code (e.g., Java code) and data. Active networks allow the customization of network services at packet transport granularity, rather than through a programmable control plane (which is the goal of Opensig). Active networks offer maximum flexibility in support of service creation but at the cost of adding more complexity to the programming model. The Active Network approach is, however, an order of magnitude more dynamic than Opensig's quasi-static network programming interfaces.

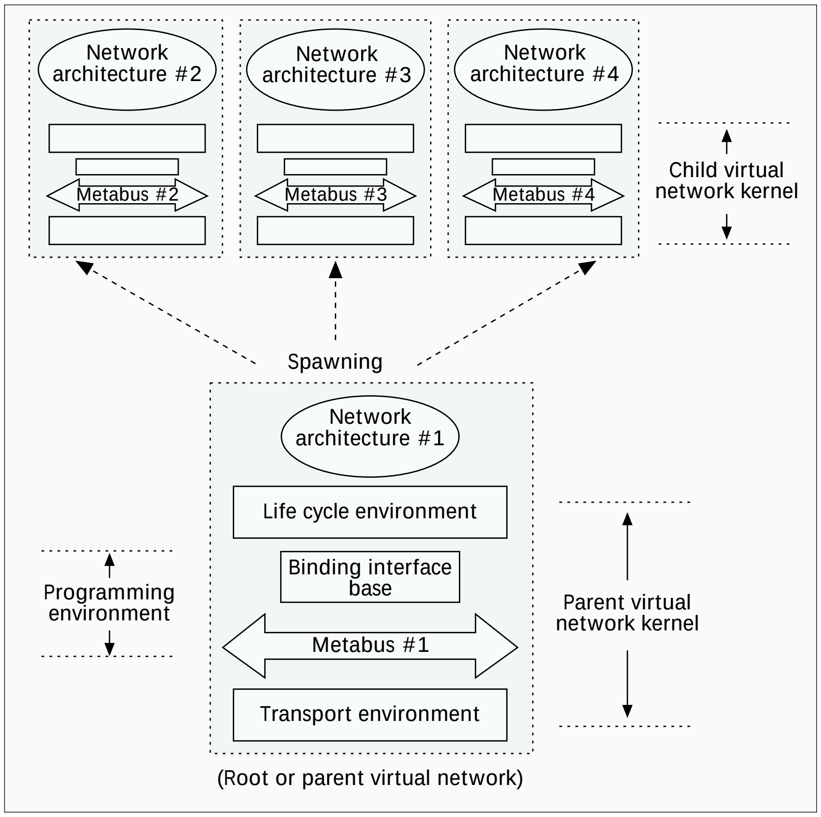
A common set of characteristics [3] govern the construction of programmable networks:

- *Networking technology* implicitly limits the programmability that can be delivered to higher levels. For example, some technologies are more QoS programmable (e.g., ATM), scalable (e.g., Internet), or bandwidth-limited (e.g., mobile networks).
- *Level of programmability* indicates the method, granularity, and timescale over which new services can be introduced into the network infrastructure. This in turn is strongly related to language support, programming methodology, and middleware adopted. For example, distributed object technology can be based on remote procedure call (RPC) [7] or mobile code [6] methodologies, resulting in quasi-static or dynamically composed network programming interfaces, respectively.
- *Programmable communications abstractions* indicate the level of virtualization and programmability of networking infrastructure requiring different middleware and, potentially, network node support (e.g., switch/router, base station). For example, programmable communications abstractions include virtual switches [8], switchlets [9], active nodes [10], virtual base stations [11], and virtual active networks [12].
- *Architectural domain* indicates the targeted architectural or application domain (e.g., transport, signaling, management). This potentially dictates certain design choices and impacts the construction of architectures and services offered, calling for a wide range of middleware support. Examples include composing application services [13], programmable QoS control [8], and network management [14].

We believe that the introduction of new network architectures on demand represents a difficult and complex problem. The complexity stems from the fact that it is difficult to predict all interactions between independently placed architectural components inside the network. We broadly define a network architecture as having the following generic components and attributes:

- *Network services*, which the network architecture realizes as a set of distributed network algorithms and offers to the end systems
- *Network algorithms*, which include transport, signaling/control and management mechanisms
- *Multiple time scales*, which impact and influence the design of the network algorithms
- *Network state management*, which includes the state the network algorithms operate on (e.g., switching, routing, QoS state) to support consistent services

Little work has been reported in the literature on automating the process of realizing distinct network architectures on demand. Spawning networks address this limitation by automating the network life cycle, providing a systematic approach to the design, deployment, and management of distinct internetworking architectures. Spawning networks provide a foundation for composing and deploying virtual network architectures through the availability of open programmable interfaces [15], resource partitioning [9], and the virtualization of the networking infrastructure found in today's programmable networks [3].



■ Figure 2. *The Genesis Kernel framework.*

The Genesis Kernel Framework

The Genesis virtual network kernel [29] represents a next-generation approach to the development of programmable networks, building on our earlier work on open programmable broadband [8] and mobile networks [11]. The Genesis Kernel has the capability to spawn child network architectures that can support alternative distributed network algorithms and services. The Genesis Kernel acts as a resource allocator, arbitrating between conflicting requests made by spawned virtual networks. Virtual networks spawned by the Genesis Kernel operate in isolation, with their traffic being carried securely and independently from other networks. Furthermore, child networks, created through spawning by parent networks, inherit architectural components from their parent networks, including life cycle support. Thus, a child virtual network can be a parent (i.e., provider) to its own child networks, creating the notion of *nested virtual network architectures* within a virtual network. In this respect the child network becomes a spawning network to its own child networks. In what follows, we provide an overview of the Genesis Kernel framework as illustrated in Fig. 2.

The Transport Environment

At the lowest level of the framework, 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 considered to be programmable using the virtual network kernel. Thus, the transport environment represents a programmable data path at a router. Genesis routers are capable of supporting multiple routelets, which are components of distinct virtual networks that share computational and communication resources.

The Programming Environment

Child routelets are instantiated by the parent virtual network during spawning, as illustrated in Fig. 2. These routelets operate on a subset of the parent's resources and operate independently of the parent network. In addition, routelets are controlled through separate *programming environments*. Each virtual network kernel can create a distinct programming environment that supports routelet programming and enables the interaction between distributed objects that characterize the spawned network architecture illustrated in Fig. 2. The programming environment comprises a *metabus* that partitions the distributed object space supporting communications between objects associated with the same spawned virtual network. The metabus is a per-virtual-network software bus for object interaction. A *binding interface base* [8] 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.

The Life Cycle Environment

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 virtual network architectures. The life cycle process comprises three 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 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, routing, 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. In addition, virtual network architecting is supported, which allows the network designer to analyze the pros and cons of a virtual network design space.

As illustrated in Fig. 2, the metabus and binding interface base also support the *life cycle environment*, which realizes the virtual network 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 the 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.

Design Principles

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

Separation — Spawning results in the composition of a child network architecture in terms of transport, control, and management algorithms. Child virtual networks operate in isolation with their traffic carried securely and independent from other networks. The allocation of parent network resources to support a child virtual network is coupled with the separation of responsibilities and transparency of operation between parent and child architectures. We refer to this as the *principle of separation*. Once a child network has been spawned, the child has complete freedom to manage and control its own resources in an autonomous manner based on its instantiated architecture.

Nesting — 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 timescale) and short-lived networks (e.g., a collaborative group network operating within the context of the corporate network but only active for a short period). Spawned networks represent virtual networks and their associated subscribers, where every child virtual network can be a parent (i.e., provider) to its own child networks. The parent-to-child relationship represents a *virtual network inheritance tree*. As illustrated in Fig. 1, two child networks are spawned by the parent network. The first child network is a Cellular IP virtual network that supports wireless extensions to the parent network. The other child network (illustrated in Fig. 1) supports a differentiated services architecture [16] operating over the same network infrastructure. An additional level of nesting is shown where the Cellular IP network spawns a child network. The relationship between spawned networks in this example is captured by an inheritance tree [17].

Inheritance — 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 represented by a set of network objects for transport, control, and management. Inheritance allows the network designer to leverage existing network objects when constructing new child networks. In contrast, the child network composition may be distinct from its parent, thereby not using inheritance.

Transport Environment

The Genesis transport environment consists of a set of open programmable virtual nodes called *routelets* that are connected by virtual links to form the lowest programmable layer of a virtual network kernel.

Routelet Architecture

The 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 Fig. 3. A routelet comprises a forwarding engine, a control unit, and a set of input and output ports.

Ports and Engines — Ports and engines manage incoming and outgoing packets as specified by a virtual network profiling script. A profiling script captures the blueprint of a virtual network architecture, capturing the composition of the 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 plugins with well defined interfaces and globally unique identifiers. Transport modules can be dynamically loaded into routelets by the Genesis Kernel to form new and distinct programmable data paths. We define a generic set of transport modules that can be extended to accommodate new classes of data path that are made programmable:
• *Encapsulators*, which add specific headers (e.g., RTP, IPv4) to packets at the end systems or routelets
• *Forwarders*, which execute particular packet forwarding mechanisms (e.g., IPv6, MPLS, Cellular IP) at routelets
• *Classifiers*, which separate packets in order to receive special treatment by routelets
• *Processors*, which process packets based on architecturally specific plugins (e.g., police, mark, monitor, shape, filter packets)
• *Schedulers*, which regulate the use of virtual link capacity based on a programmable buffer and queue management capability

Child ports and engines can be constructed by directly inheriting their parents' 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 soon as they enter the routelet and process them based on the instantiated transport modules. In the case of a differential services routelet, for example, the input ports would contain differential-service-specific mechanisms (e.g., meters and markers used to maintain traffic conditioning agreements at boundary routers of a differentiated service [16] virtual network). A virtual link is typically shared by user/subscriber traffic generated by end systems associated with the parent network and aggregated traffic associated with its child networks. This user and child network traffic contend for the parent's virtual link capacity. The output port regulates access to communication resources associated with a virtual link among these competing elements. Routelets can be used to support a wide range of network architectures. For example, an IPv4 forwarding engine can be combined with output ports that enforce suitable per-hop behavior for differential services. Differential service routelets would mark and forward IP packets based on a fixed number of per-hop behaviors. Differential service virtual networks would use programmable signaling and policy mechanisms to deliver differentiated QoS to their subscribers. In this case, all the necessary signaling for the establishment of service-level agreements can be made entirely programmable on top of the programming environment.

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

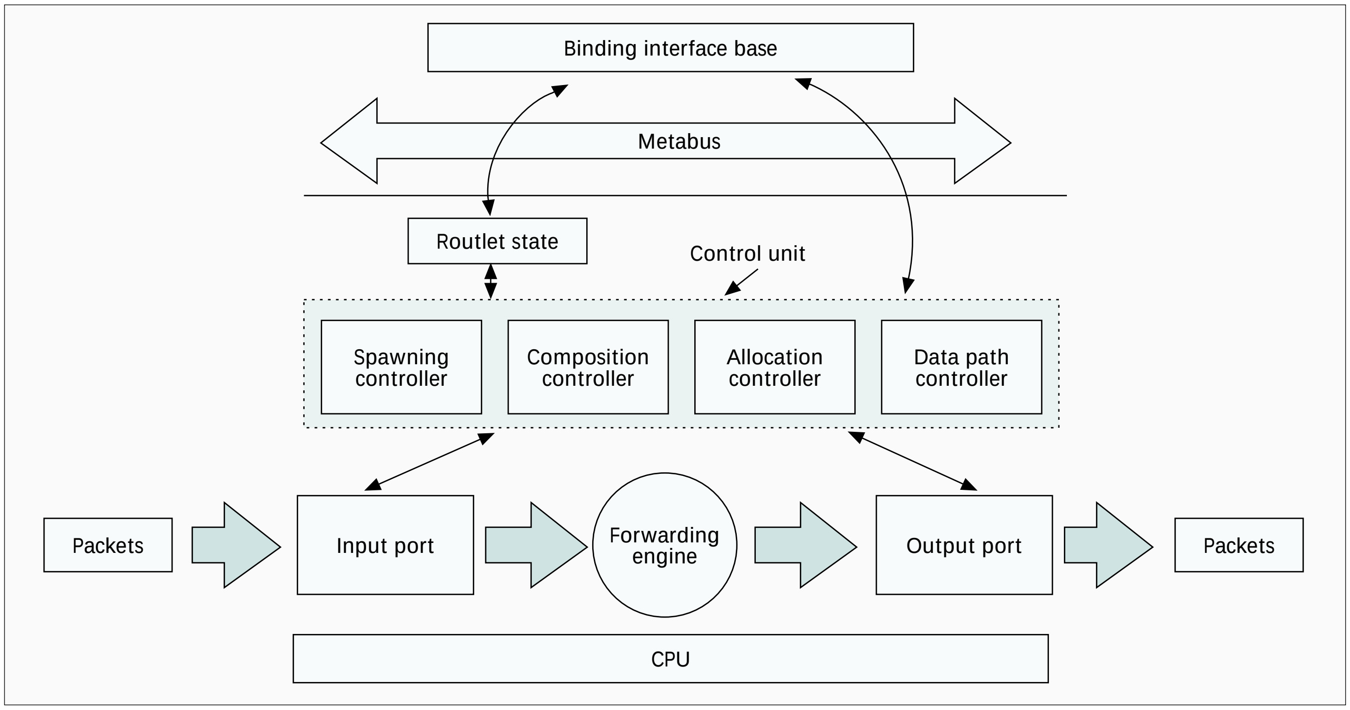
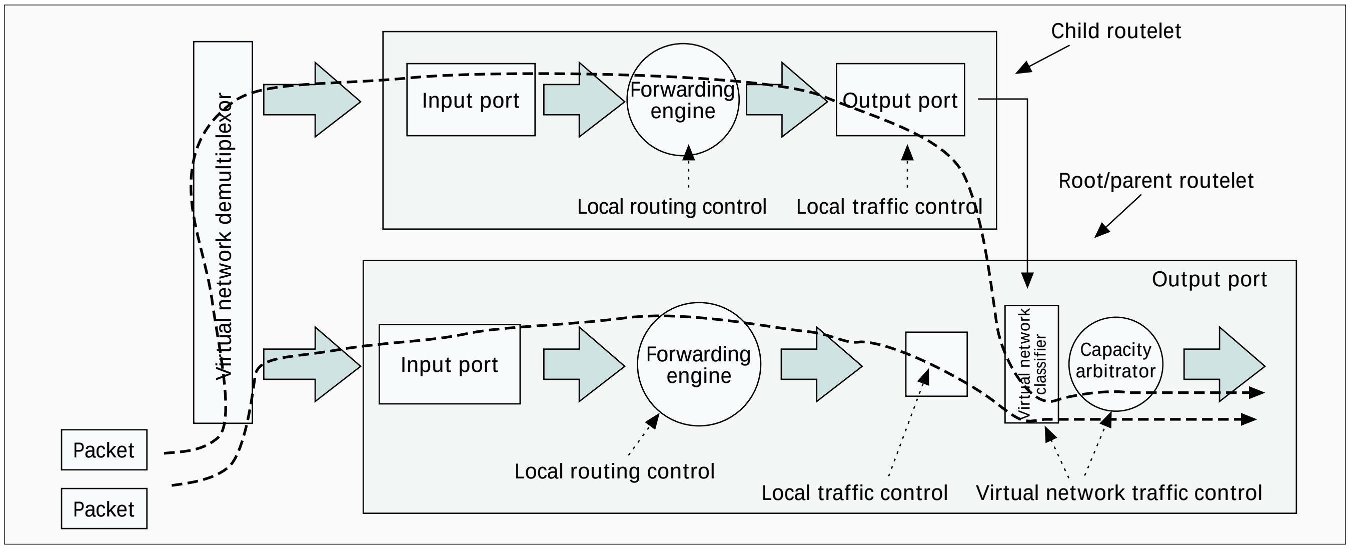


Figure 3. *Routelet architecture.*



■ Figure 4. *Nested routelets.*

- *A spawning controller*, which *bootstraps* child routelets through virtualization
- *A composition controller*, which manages the composition of the routelet using a set of transport module references and a composition graph to construct ports and engines
- *An allocation controller*, which manages the computational resources of a routelet
- *A data path controller*, which manages communication resources and transportation of packets

The spawning, composition, and allocation controllers are common to all routelets associated with specific virtual networks. In contrast, data path controllers are dynamically composed during the spawning phase based on a profile script. Data path 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 also 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 [8, 9]. In [9], for example, switchlets support the coexistence of multiple control architectures operating over the same physical ATM switch. 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 [8] and switchlets [9]. The underlying link-layer technology, however, may impact the level of programmability and QoS provisioning that can be delivered at the internetworking layer, as discussed earlier.

Nested Routelets

Nested routelets operate on the same physical node and maintain their structure according to a virtual network inheritance tree, as discussed previously. Routelet nesting is built on the network virtualization process, resource partitioning, and the separation of control between parent and child networks. Child routelets are dynamically created and composed during the spawning process, where parent 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. The nesting principle helps deal with complexity and maintain transparency of operation and separation between child and parent networks. Nesting maintains the autonomous nature of routelets. Routelets are unaware that packets are processed according to inheritance trees. A routelet simply receives a packet on one of its input ports associated with its virtual links, sends the packet to its forwarding engine for processing, and then forwards the packet to an output port, where it is finally scheduled for virtual link transmission. Based on parent policies, child packets may traverse one or more routelets and their abstracted virtual links through a hierarchy, exiting at the root of the tree onto the physical link.

We use an example scenario to illustrate how nesting is supported in a Genesis router. As illustrated in Fig. 4, two packets arrive at a Genesis router. Every packet arrival must be demultiplexed to a given spawned virtual network. A virtual network demultiplexer 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. The routelet's forwarding engine maps and forwards packets to its output port for further packet treatment and/or virtual link scheduling, as illustrated in Fig. 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 specific 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 tree.

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 and layer headers. Virtual networks are thus 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 from the life cycle environment of the parent kernel. The virtual network demultiplexer maintains a database of virtual network identifiers which it uses to map incoming packets to specific routelets. Genesis routers can also be interconnected with legacy IP routers. Interoperability is accomplished using encapsulation. During spawning, each virtual network configures its own tunnels, which are supported by Genesis routers at borders with legacy IP networks.

At each parent network, local routelet traffic is multiplexed with child virtual network traffic. Multiplexing is managed by a virtual network classifier and provisioned through virtual link capacity class allocations. An *arbitrator,* 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. Every packet traverses the nested hierarchy tree until it is scheduled and exits onto the physical link.

The Programming Environment

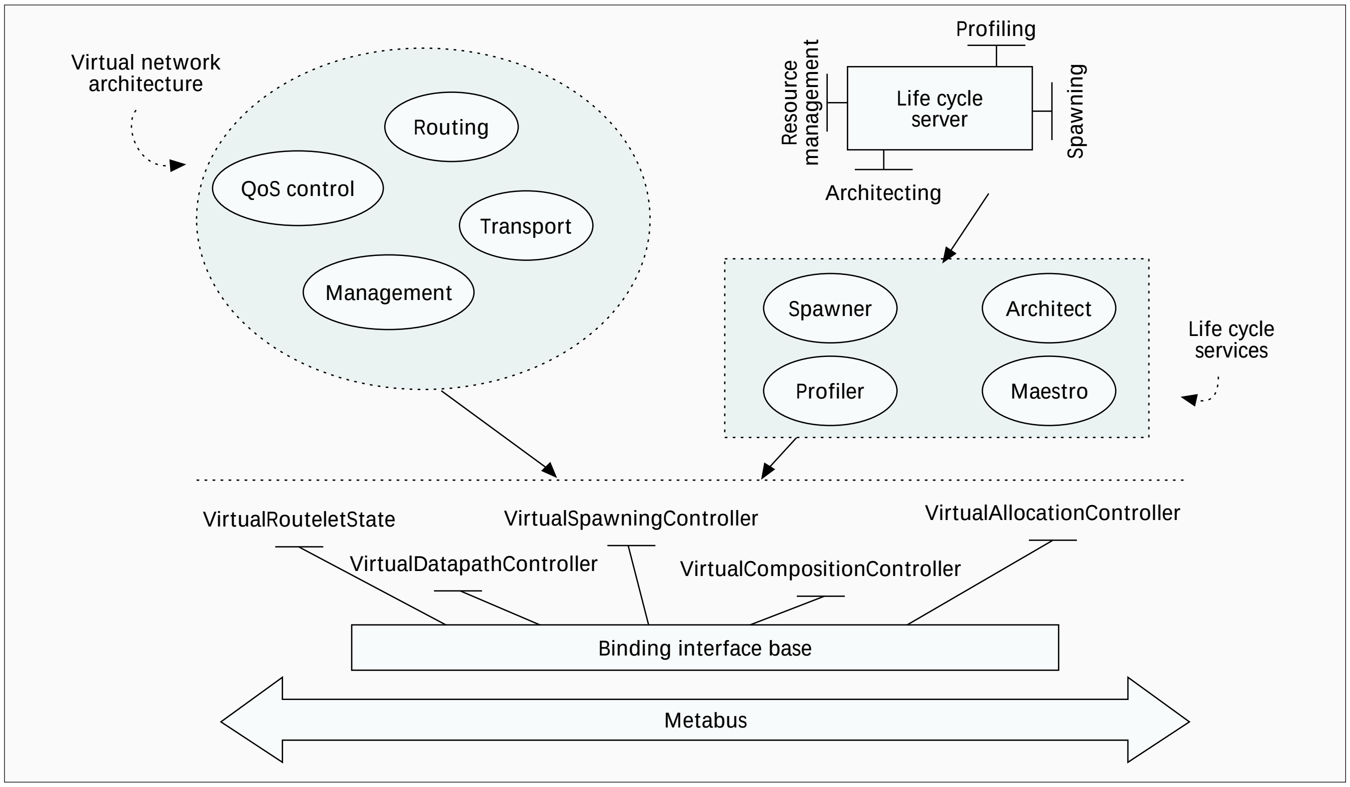
Routelets interconnected by virtual links form spawned virtual networks, providing a basis for making the data path programmable. Each network architecture consists of a programmable data path (i.e., the transport environment) and a set of distributed controllers that realize communication algorithms (e.g., routing, control, management), as discussed earlier. These distributed controllers use the programming environment. We believe that IP network architectures can be made programmable using open interfaces that allow access to the routelet's internal components (e.g., routelet state, packet classifiers and schedulers). While the implementation of routelet transport modules is platform-dependent, the programming environment offers platform-independent access to the router's components, allowing a variety of protocols to be dynamically programmed. The programming environment is illustrated in Fig. 5 and discussed below.

The Metabus

A key architectural component for programmable virtual networks is the metabus, which dynamically partitions the distributed object space supported by the Genesis Kernel. At the lowest level of the programming environment the metabus provides a foundation for the realization of distinct virtual network architectures by enabling remote communications between distributed objects that form virtual network architectures.

Middleware technologies [7] hide the complexity of network and operating system architectures from application developers, allowing distributed object components to communicate over a uniform information/software bus. Middleware technologies typically support a single monolithic software bus per distributed system for the interaction of all distributed objects contained in the system. We argue that the concept of a single monolithic software bus limits the rollout of new distributed networked applications (e.g., virtual networking technologies) that require secure, dedicated, and QoS-configurable communications. The metabus is the enabling technology for dynamic provisioning of dedicated middleware support for networked distributed applications. The concept of the metabus extends the capabilities offered by the traditional software bus by supporting networkwide spawning of multiple (possibly distinct) metabuses that can serve as dedicated information buses for spawned virtual network architectures.

Existing object request brokers (ORBs) [7] support a number of ad hoc solutions for realizing isolation between distributed object computing environments. For example, ORBs can use different naming services to offer dedicated middleware support to distributed objects. Naming services are con-



■ Figure 5. *The programming environment.*

figured manually by system administrators before distributed applications can be launched. Isolation between applications can be achieved through object classification supported by systemwide naming or trading services. We believe that there are scalability and performance issues when these techniques are applied to spawning networks. We argue that in order to meet requirements imposed by large-scale distributed systems (e.g., programmable virtual networks), distributed computing environments need to dynamically provision dedicated middleware support (i.e., dedicated software buses and object services on a per-virtual-network basis). Thus, the network can be shared and partitioned between diverse sets of distributed objects in a secure, scalable, QoS-configurable manner.

Each virtual network supported by the Genesis Kernel has its own metabus. The metabus is dynamically created as part of the spawning process and operates on its associated virtual network resource space. The spawning process instantiates a metabus for each new virtual network, providing the necessary middleware support for the interaction of all the architectural (e.g., routing) and system (e.g., routelet) objects that characterize a new child virtual network architecture. Routelets associated with the same virtual network use their metabus to send and receive signaling messages using remote method invocations. The metabus is a general concept for programmable virtual networking. We have chosen to realize the metabus abstraction as an *orblet*, a virtual ORB derived from the Common ORB Architecture (CORBA) [7] object programming paradigm. Both first-generation kernels [8, 11] we have developed use CORBA technology for service creation, signaling, and management.

The Binding Interface Base

Metabuses support 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* [15] objects provide a set of handlers to the routelet controllers and resources, allowing for the programmability of a range of internetworking architectures using a programming environment. The interfaces that constitute the binding interface base are illustrated in Fig. 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 which 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.

The Life Cycle Environment

The life cycle environment provides support for the profiling, spawning, and management of virtual networks. Profiling, spawning, and management are composed of a set of services and mechanisms that are common to all virtual networks. The virtual network life cycle is realized through the interaction of the transport, programming, and life cycle environments.

Profiling

The Profiling Process — 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 earlier. 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. 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.

Topology Requirements — 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 subnetworks and cover a wide area interconnecting a number of intranets, or be restricted to a few local subnetworks. When profiling large-scale networks the architect has the flexibility to provide an outline of the topology graph, specifying strategic subnetworks 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.

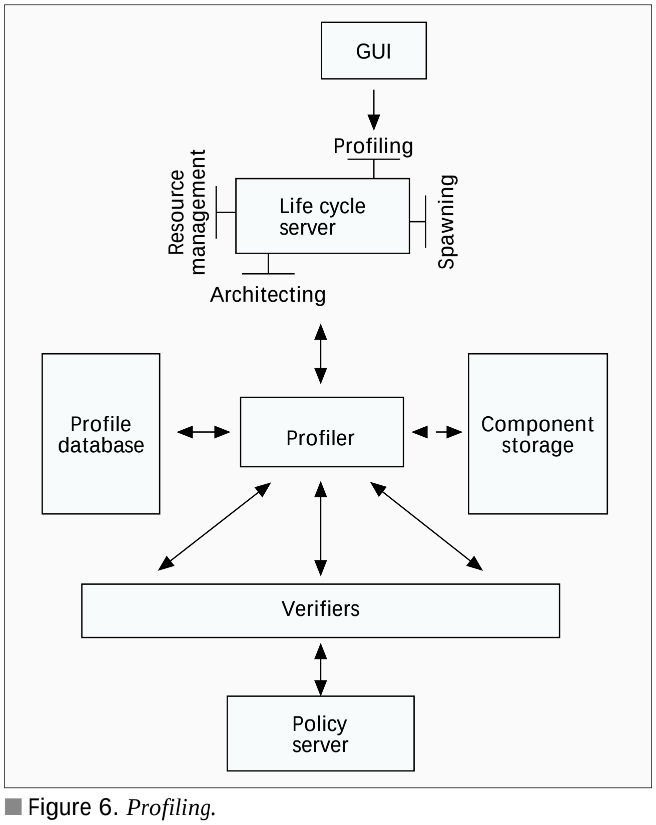
Architectural Components — The Genesis profiling system allows the network designer to dynamically select architectural components and instantiate them as part of a spawned network architecture. For example, a number of routing protocols for intra- and interdomain routing can be made available using the component storage of the parent network (e.g., RIP, OSPF, BGP). Similarly, QoS architectures based on well-founded models (e.g., integrated [18] and differentiated services [16]) can be dynamically selected and used for the provisioning of QoS in virtual networks. Transport protocols, such as TCP, Real-Time Transport Protocol (RTP), and User Datagram Protocol (UDP), and network management components, such as Simple Network Management Protocol (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 networkwide 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.

Resource Requirements — Resource requirements for virtual links are specified in terms of required bandwidth and capacity classes. Virtual links support capacity classes where child network traffic classes are mapped and multiplexed. Capacity classes represent general-purpose *resource pipes* allowing parent networks to manage child traffic in an aggregated manner. The following capacity classes are considered:
- A constant capacity class, which statically allocates bandwidth to a virtual link based on a peak rate specification
- A controlled capacity class, which allocates capacity based on an average rate specification
- A best-effort capacity class, which provides no explicit service assurances

The Profiling System — The Genesis profiling system is illustrated in Fig. 6. 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.

Once the profiling script is generated the profiler service distributes the script to a set of verifiers in order to apply a number of validity checks. Verifiers interact with policy servers to determine the validity of the specification of profiled network architectures. The result of this interaction determines whether the architecture can be safely spawned by the parent network or not. The policy server maintains a set of guidelines that are used to determine whether a profiled child architecture can be spawned on top of a parent network, and rules for architecting routelets and QoS provisioning based on the resource management capability of parent networks. Successful verification allows the life cycle process to move to the spawning phase. Profiling script verification is analogous to the compilation of a program from a source code file.



■ Figure 6. *Profiling.*

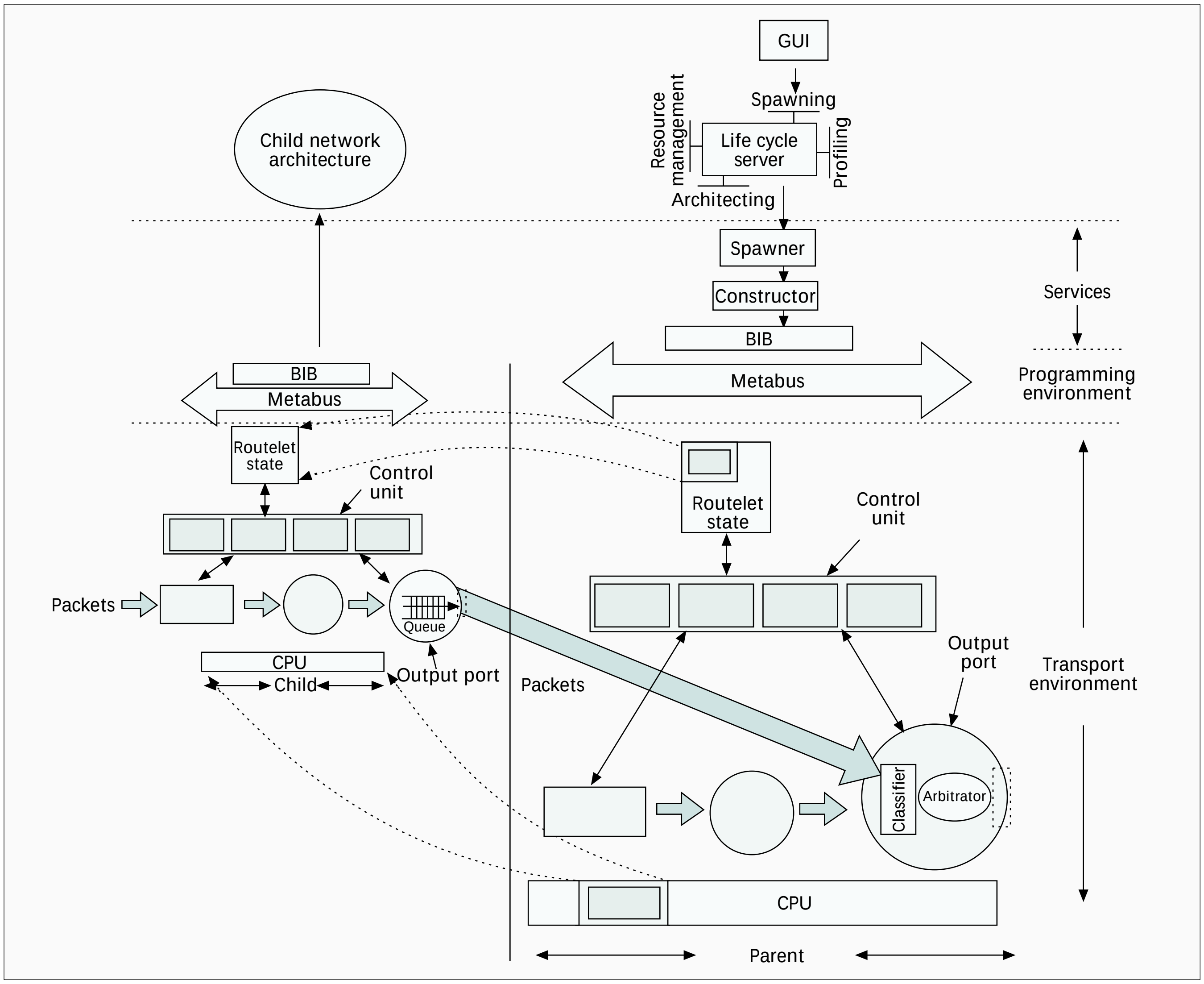
Spawning

The Spawning Process — 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. The spawning phase allocates resources to the newly created virtual network through the process of *virtualization*. The virtualization process realizes resource partitioning and isolation by creating the illusion that each virtual network is the only system making use of the underlying physical networking infrastructure and resources. 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. Once this phase is complete, the virtual network architecture begins executing on the network infrastructure.

Spawning Services — Spawning child virtual network architectures includes creating child transport and programming environments and then instantiating the transport, control, and management objects that characterize network architectures. Life cycle services are common services supported by the Genesis Kernel. Spawning services include:
- A spawner service, which applies centralized control over the spawning process interacting with the profiling and management services
- Component storage, which is a distributed database of virtual network software building blocks
- A set of constructor objects, which run on all the nodes of a parent topology and interact with the spawner to create a child network

Constructors realize the creation of routelets, the instantiation of a new metabus, and the deployment of a child network architecture on a single network node. The spawner is a distributed system, which directs the spawning process, executing a profiling script. The component storage is a database of transport modules and network objects which realize programmable control and management. The spawner "announces" the child network's bandwidth requirements to the maestro service of the parent network. The maestro is a distributed controller which performs admission testing on the link capacity requirements of the child network. If the test is successful, the network is spawned.

Spawning a Virtual Network — The creation process associated with spawning a child transport environment centers around:



■ Figure 7. *Spawning architecture.*

- The creation and composition of routelets and data paths, respectively
- The bootstrapping of routelets into physical routers based on the child network topology
- The binding of virtual links to routelets, culminating in the instantiation of a child transport environment over the parent network

The nesting property allows a child network to inherit life cycle support from its parent. A child routelet is bootstrapped by a parent spawning controller, as illustrated in Fig. 7. The spawning controller interacts with the allocation controller to reserve a portion of the parent routelet's computational resources for the execution of the child routelet. Next, the child routelet's state is initialized. During this phase a spawner acquires all the required transport modules unavailable in the parent network. 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 routelet's composition controller. Finally, the child network's data path controller is composed, and its queues are configured to forward traffic to parent network queues. The last phase of spawning the child transport system is the binding of virtual links to a set of distributed routelets forming a virtual network topology.

Following transport environment creation, the spawning process creates a programming environment and launches the child virtual network architecture. This process is managed by the parent's spawner and constructor services. A metabus is dynamically created to support the child network's distributed object environment. Next, the metabus is initialized to support binding interfaces. The spawner and constructors load network objects from component storage, and instantiate and bind them to the child network's metabus. These distributed objects represent communication protocols and algorithms selected by the network architect to realize routing, signaling, and management services of the virtual network architecture.

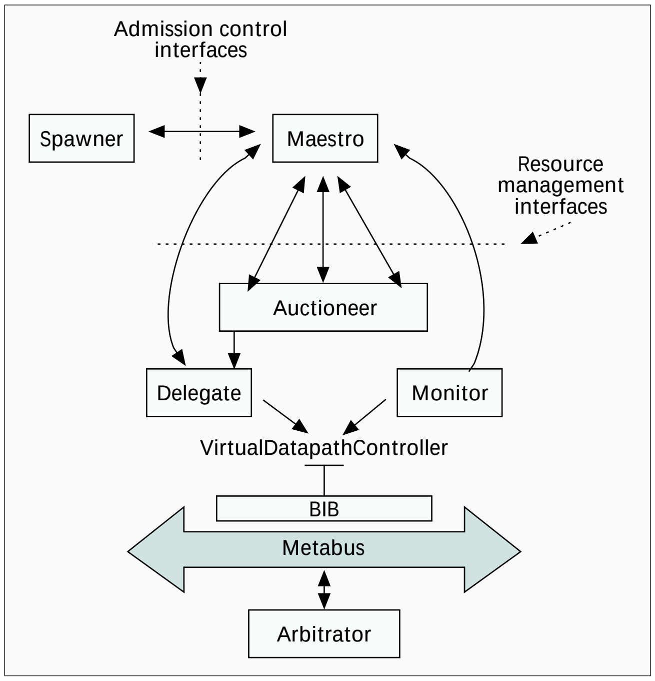
An Illustrated Example: Spawning Cellular IP — In what follows we present a simple illustrative example of spawning a child Cellular IP [2] access network and discuss the parent-child relationship illustrated in Fig. 7. The example focuses on a specific node in the spawning process and does not illustrate the binding of routelets with virtual links to form the complete child network. In our example scenario, the child network "overwrites" the standard IP forwarding engine of its parent and instantiates Cellular IP [2], a new IP protocol for supporting mobile hosts. The Cellular IP architecture is optimized to support fast local handoff control in access networks. 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 these caches. Mobile IP is used to support mobility between gateways. Cellular IP transport modules loaded into the parent network include a Cellular IP forwarder, soft-state management, and handoff control modules.

This example shows how a parent wireline and access network is dynamically extended to support the Cellular IP architecture, services, and protocol. In addition, the child network uses a very simple differentiated service approach to the provisioning of QoS. The child network's output ports contain a low-priority packet discard transport module to respond to congestion. Network traffic which exits via the child output ports enters the output ports of the parent network. While child network flows are treated according to their traffic conditioning agreement by the differentiated service network, their aggregated traffic is scheduled at a finer granularity by the parent network according to the amount of resources that have been assigned to it. Uplink control and data packets received from a mobile device at a routelet's forwarder are used to set up per-mobile soft-state paging and routing information which is used for any subsequent downlink packet data delivery. A base station represents a special case of a routelet with a number of additional transport modules instantiated to deal with the air interface. These wireless transport modules (e.g., priority packet drop) are used to respond to congestion that may be experienced over longer timescales at the wireless link. As illustrated, a parent routelet schedules the packets of a child Cellular IP virtual network using a virtual network capacity scheduler called the *arbitrator*, discussed in the next section.

Management

The Management Process — Once a profiled architecture has been successfully spawned, the virtual network needs to be controlled, managed, and possibly refined. The management phase supports virtual network resource management based on per-virtual-network policy which 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 renegotiation process. The management phase also subsumes the process of refinement of spawned network architectures by observing and controlling the dynamic behavior of virtual networks. 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, Cellular IP can be refined to optimally operate in picocellular, campus, and metropolitan-area environments through the process of architecting and refinement.

Architecting Virtual Networks — During the management phase, spawned virtual networks can be selected and visualized. Once selected, networks can be observed, reprovisioned, managed, and architecturally refined on the fly. Management services include the ability to modify existing architectural features using dynamic plugins (e.g., replacing Cellular IP routing, paging, and handoff services) or reconfiguring existing network services (e.g., modifying Cellular IP system timers for better performance and scalability associated with the operating environment). Architecting represents a methodology for automating the creation, refinement, and provisioning of novel network services, protocol architectures, and technologies, allowing the network designer to add, modify, or delete network services as required. This includes the addition or removal of complete spawned child virtual networks. By tuning distributed network objects that characterize network architectures, the network architect can refine programmable virtual networks. The network architect uses visualization tools to interact with target virtual network kernels in order to exert control over multiple executing virtual networks and their objects. The network architect can modify or replace transport modules (e.g., input ports, output ports, forwarding engines), install new network controllers on top of virtual network programming environments, and override or extend the life cycle environments of virtual network kernels.


Admission control interfaces
Spawner
Maestro
Resource management interfaces
Auctioneer
Delegate
Monitor
VirtualDatapathController
BIB
Metabus
Arbitrator

■ Figure 8. *Virtual network management.*

Virtual Network Resource Management — The Genesis virtual network resource management system [17] leverages the benefits of the kernel hierarchical model of inheritance and nesting, delivering scalable virtual network resource management. It is governed by four basic design goals:

- *Dynamic provisioning* of virtual network resources based on per-virtual-network policy and slow timescale resource renegotiation
- *Capacity classes*, which provide general-purpose *resource pipes*, allowing the underlying parent controller architecture to manage child traffic in an aggregated and scalable manner
- *Inheritance*, which allows child networks to inherit the resource management facilities and provisioning characteristics of their parents
- *Autonomous control*, which allows child virtual networks to manage user-level QoS independent of their parent/provider network

Architectural Components — The Genesis virtual network resource management architectural model comprises a number of distributed architectural elements, as illustrated in Fig. 8. With the exception of the delegate, arbitrator, and monitor elements, which operate on every routelet of a virtual network, all other architectural elements (e.g., auctioneer, maestro) can operate either locally with each routelet or remotely as servers. A routelet object has a maestro and delegate, and one arbitrator and monitor per virtual link.

The maestro is the central controller responsible for managing the global resource policy within the virtual network or virtual network domain. It operates on the virtual network management timescale and is driven by current resource availability and per-virtual network policy. Maestros set pricing and rate strategies across managed resources and influence child virtual networks in a manner to promote the efficient use of resources. A delegate, serving as a decentralized proxy agent for a maestro, manages all local resource interactions and control mechanisms on a routelet. An auctioneer implements an economic auctioning model for resource allocation supporting various strategies between virtual network providers and subscribers. The auctioneer services bids from child virtual networks over slow provisioning timescales, promoting a highly competitive system among subscribers. A monitor performs policing and monitoring on individual parent resources. Policing ensures that child virtual networks do not consume parent resources above and beyond an agreed allocation of the virtual link capacity. An arbitrator is a transport module and represents an abstract virtual network capacity scheduler controlling access to each parent resource. The arbitrator receives a virtual network policy (capacity classes and weights) from the maestro over slow timescale provisioning intervals upon completion of a resource allocation process. The virtual link arbitrator manages the access and control of the parent link capacity, based on the virtual network policy. A hierarchical link sharing system is composed of capacity arbitrators executing on distinct virtual network routelets. Arbitrators support isolation and transparency of operation between parent and child network architectures.

Timescales — 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. We believe that it is unlikely that spawning networks would operate in a stable manner on faster timescales. Network architecting takes place over much slower configuration management timescales. Network architects analyze the network design space and modify or replace network objects that characterize distributed network architectures. Profiling and spawning of virtual networks takes place at the slowest timescales when new collaborative environments are created to satisfy communication needs of distinct communities. We believe that the combination of slow timescale, dynamic provisioning, and inheritance characteristics make the design of the Genesis Kernel efficient, flexible, and scalable. For full details on the Genesis virtual network resource management system see [17].

Implementation Considerations

Spawning network architectures appears to be an exceedingly difficult task. To achieve our goal of building spawning networks we plan (via a three-phase approach):

1. to deploy programmable foundation services, network controllers (objects), and application programming interfaces (APIs) required to program and support a set of well understood baseline architectures over a spawning networks testbed. The baseline architectures include a Cellular IP network architecture [2], a differentiated services IP network architecture [16], and a mobile ad hoc network architecture [19].
2. to implement the life cycle environment that will allow us to profile, spawn, and manage network architectures.
3. to architect new virtual network architectures from the baseline set through spawning, visualization, and management.

We plan to build a testbed based on our implementation strategy and study its behavior. We believe that we need to take a hands-on approach coupled with analysis of the Genesis Kernel framework.

If phase three proves to be as easy as the life cycle process suggests, discovering new, more useful architectures may become vastly simpler than it is today. The three-phase implementation approach appears to be straightforward; however, it raises a number of challenging issues. Phase two assumes that it will be simpler to profile an architecture and determine the location for the deployment of network objects than it is presently. But, can this be done efficiently? How will management become easier? One of the goals of our work is to build more powerful tools to help with this architecting process, allowing for a more systematic study of the design space under operational conditions. The development of visualization tools is a key contribution to phase three of the implementation strategy. However, the effective use of this depends on more than a visualization tool. It depends on a well founded understanding of what should be achieved vs. what may be achieved, and how to modify a prototype network architecture accordingly. Exploration of the network design space is a challenging issue that we will investigate during the implementation and evaluation of spawning networks.

Another challenging issue is related to the computational efficiency and performance of spawning networks. Currently, transmission rates are increasing more rapidly than the computational power needed for routing and congestion control. Programmable networks require more computational resources than existing networks in order to support the introduction and architecting of spawned virtual networks. Spawning network technology does not reduce the requirements of programmable networks in terms of processing power and storage capacity; however, spawning networks do not increase these requirements either.

Transport modules should be capable of forwarding packets as fast as handcrafted optimized code executing on routers today. Results from extensible router architectures [20] indicate that software-based packet scheduling and forwarding is viable. Spawning networks may impact the performance of packet forwarding due to hierarchical link sharing design that underpins routelet nesting and virtual network isolation. However, fast-track and cut-through techniques will be deployed during the implementation phase to offset any potential performance costs that may be associated with nested virtual networks.

Another question is related to the complexity associated with profiling network architectures. To address this complexity, the Genesis Kernel framework introduces the concept of inheritance of architectural components and provisioning characteristics to programmable networks. Inheritance allows the network designer to leverage existing network objects when constructing new child networks. The network designer may also select architectural components from libraries to realize certain architectural subsystems and network services. In this manner complex virtual network kernel architectural subsystems and services can be hidden from the network designer, whereas other subsystems and services may be accessible for customization. Currently, the Genesis Kernel framework does not support the notion of multiple inheritance. Child networks described in this article have a single parent. This design decision allows investigation of the spawning, nesting, and inheritance capabilities of the Genesis Kernel, without burdening the architecture with the additional complexity of supporting multiple inheritance. Future work will consider extending the Genesis architecture to support multiple inheritance of architectural components and provisioning characteristics.

Related Work

The Tempest project has investigated the deployment of multiple coexisting control architectures in broadband ATM environments [9]. 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 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 programmable virtual network architectures.

Virtual private network services in broadband ATM networks have been the subject of a substantial amount of research. In [21] the concept of a virtual path group is introduced as a virtual network building block to simplify virtual path dynamic routing. In [22] the concept of nested virtual networks is introduced and an architecture that supports resource management of broadband virtual networks presented. The Genesis Kernel framework also uses the concept of nesting in pursuit of the programmability and automated deployment of network architectures spanning transport, control, and management planes at level three and above. Typically, spawned network architectures support alternative signaling protocols, communications services, QoS control, and network management from those of parent architectures. A related project called Virtual Network Service (VNS) [23] is investigating the provisioning of QoS 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. Genesis is investigating the use of an auctioning mechanism to perform virtual network resource management. We argue that auctions can capture a diverse set of incentives between various virtual network service subscribers over multiple timescales.

The X-Bone [24] project aims to automate the process of establishing IP overlay networks. Currently, overlays (e.g., MBone, 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 MBone and uses a two-layer multicast IP system to facilitate the dynamic deployment of different overlays over the Internet. X-Bone overlays are not programmable, however. The Supranet [25] project considers a networkless society where networks and service creation are facilitated and tailored to group collaborative needs. A supranet is a virtual network that requires definition of the characteristics of the collaborative environment which 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 has investigated the deployment of multiple coexisting execution environments through appropriate operating system support [10] and an active network encapsulation protocol. In [12] the use of active networking technology is studied for the deployment of IP-based virtual networks. In most current research in active networks the dynamic deployment of software at runtime is being accomplished within the confines of a given network architecture and node operating system. By 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 recent Internet Engineering Task Force (IETF) proposals have discussed IP virtual private networks [26]. Others have addressed issues of performance [27].

Conclusion

We believe that the design, creation, and deployment of new network architectures should be automated and built on spawning networks. We argue that this will result in better network customization, resource control, and time to deployment for new network architectures, services, protocols, and technologies. In this article we present the design of the Genesis Kernel; a virtual network operating system capable of spawning network architectures on demand. The Genesis Kernel framework presents a new approach to the deployment of network architectures through the automation of the virtual network life cycle. In the next phase of our work we plan to implement [28] the framework on a spawning testbed, taking into account the implementation considerations discussed in the previous section.

Acknowledgments

This work is supported in part by the National Science Foundation (NSF) under CAREER Award ANI-9876299 and with the support of COMET Group industrial sponsors. In particular, we would like to thank Intel Corporation, Hitachi Limited, and Nortel Networks for supporting the Genesis Project. John B. Vicente (Intel) would like to thank the Intel Research Council for their support during his visit with the Center for Telecommunications Research, Columbia University. Miki Kazuho (Hitachi) would like to express his thanks to Hitachi Ltd. for their support of his work on programmable networks at Columbia University. Hermann G. De Meer is grateful to Deutsche Forschungsgemeinschaft (DFG) for providing his fellowship and research grant Me 1703/2-1. Daniel A. Villela would like to thank the National Council for Scientific and Technological Development (CNPq-Brazil) for sponsoring his scholarship at Columbia University (ref. 200168/98-3). The authors would like to thank the anonymous reviewers for helping to improve this article. Finally, we would like to thank Aurel A. Lazar (xbind Inc. and Columbia University) for his contribution to the concepts of spawning networks.

References

[1] A. A. Lazar and A. T. Campbell, “Spawning Network Architectures,” White Paper, Ctr. for Telecommun. Res., Columbia Univ., comet.columbia.edu/genesis, Jan. 1998.

[2] A. G. Valko, A. T. Campbell, and J. Gomez, "Cellular IP," draft-valko-cellularip-00.txt, internet-draft, Nov. 1998, work in progress.
[3] A. T. Campbell *et al.*, "A Survey of Programmable Networks," *ACM SIGCOMM Comp. Commun. Rev.*, Apr. 1999.
[4] OPENSIG Working Group comet.columbia.edu/opensig
[5] DARPA Active Network Program, www.darpa.mil/ito/research/anets/projects.html
[6] D. Wetherall, J. Guttag, and D. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," *Proc. 1st Int'l. Conf. Open Architectures and Network Programming*, San Francisco, CA, Apr. 1998.
[7] OMG, "The Common Object Request Broker: Architecture and Specification," Rev. 2.3, Nov. 1998.
[8] A. A. Lazar, K. S. Lim, and F. Marconcini, "Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture," *IEEE JSAC*, Special Issue on Distributed Multimedia Systems, Sept. 1996.
[9] J. E. Van der Merwe *et al.*, "Tempest: A Practical Framework for Network Programmability," *IEEE Network*, May 1998.
[10] K. Calvert, "Architectural Framework for Active Networks," AN WG draft, Aug. 1998.
[11] A. T. Campbell, M. E. Kounavis, and R. R.-F. Liao, "Programmable Mobile Networks," *Comp. Networks and ISDN Sys.*, Apr. 1999.
[12] S. da Silva, D. Florissi, and Y. Yemini "NetScript: A Language-Based Approach to Active Networks," Tech. rep., Comp. Sci. Dept., Columbia Univ., Jan. 27, 1998.
[13] E. Amir, S. McCanne, and R. Katz, "An Active Service Framework and Its Application to Real-Time Multimedia Transcoding," *Proc. ACM SIGCOMM '98*, Vancouver, Canada, Aug. 1998.
[14] B. Schwartz et al., "Smart Packets for Active Networks," *Proc. 2nd Int'l. Conf. Open Architectures and Network Programming*, New York, 1999.
[15] A. A. Lazar, "Programming Telecommunication Networks," *IEEE Network*, vol. 11, no. 5, Sept./Oct. 1997.
[16] Y. Bernet *et al.*, "A Framework for Differentiated Services," draft-ietf-diffserv-framework-01.txt, internet-draft, Feb. 1999, work in progress.
[17] A. T. Campbell, J. Vicente, and D. A. M Villela, "Virtuosity: Performing Virtual Network Resource Management," *Proc. 7th Int'l. Wksp. QoS*, London, U.K., May 1999.
[18] R. Braden, D. Clark, and S. Shenker, "Integrated Services in the Internet Architecture: an Overview," RFC 1633, June 1994.
[19] S-B. Lee and A. T. Campbell, "INSIGNIA: In-band Signaling Support for QoS in Mobile Ad Hoc Networks," *Proc. 5th Int'l. Wksp. Mobile Multimedia Commun.*, Berlin, Germany, Oct. 1998.
[20] D. Decasper *et al.*, "Router Plugins: A Software Architecture for Next Generation Routers," *Proc. ACM SIGCOMM '98*, Vancouver Canada, 1998.
[21] M. C. Chan, A. A. Lazar, and R. Stadler, "Costumer Management and Control of Broadband VPN Services," *Proc. 5th IFIP/IEEE Int'l. Symp. Integrated Network Mgmt.*, San Diego, CA, May 1997.
[22] A. Yun, A. Leon-Garcia, and M. Jaseemuddin, "Virtual Networks: A Divide-and-Conquer Approach to Network Resource Management," *Proc. OPENSIG Wksp.*, New York, Oct. 1997.
[23] "Implementing Quality of Service Virtual Network Service (VNS) on CAIRN," www.cs.cmu.edu/~hzhang/VNS
[24] J. Touch and S. Hotz, "The X-Bone," 3rd Global Internet Mini-Conf. in conjunction with GLOBECOM '98, Sydney, Australia, Nov. 1998.
[25] L. Delgrossi and D. Ferrari, "A Virtual Network Service for Integrated-Services Internetworks," *Proc. 7th Int'l. Wksp. Network and Op. Syst. Support for Digital Audio and Video*, St. Louis, MO, May 1997.
[26] B. Gleeson, A. Lin, J. Heinanen, "A Framework for IP Based Virtual Private Networks," draft-gleeson-vpn-framework-00.txt, internet-draft, Feb. 1999, work in progress.
[27] N. Duffield *et al.*, "A Performance Oriented Service Interface for Virtual Private Networks," draft-duffield-vpn-qos-framework-01.txt, internet-draft, Nov. 1998, work in progress.
[28] The Genesis Project: Spawning Networks, comet.columbia.edu/genesis, 1998.
[29] A. T. Campbell *et al.*, "The Genesis Kernel: A Virtual Network Operating System for Spawning Network Architectures," *Proc. 2nd Int'l. Conf. Open Architectures and Network Programming*, New York, Mar. 1999.
[30] D. Tennenhouse and D. Wetherall, "Towards an Active Network Architecture," *Proc. Multimedia Comp. and Networking*, San Jose, CA, 1996.

Biographies

ANDREW T. CAMPBELL (campbell@comet.columbia.edu) 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, distributed systems, and QoS research. He is a past co-chair of the 5th IFIP/IEEE International Workshop on Quality of Service (IWQOS '97) and is currently co-chair of the 6th IEEE International Workshop on Mobile Multimedia Communications (MOMUC '99). He received his Ph.D. in computer science in 1996 and the NSF CAREER Award for his research in programmable mobile networking in 1999.

MICHAEL E. KOUNAVIS [StM] (mk@comet.columbia.edu) is a Ph.D. candidate and graduate research assistant in the COMET Group at the Center for Telecommunications Research, Columbia University, New York. He received a Diploma in electrical and computer engineering from the National Technical University of Athens, Greece, in 1996, and an M.Sc. degree from Columbia in 1998. His main area of research is the development of spawning networks. Over the past two years he has been actively involved in mobile network programmability and active transport over wireless networks.

DANIEL VILLELA [StM] (dvillela@comet.columbia.edu) received a degree in electrical engineering in 1997 and an M.Sc. degree in electrical engineering in 1998 from the Federal University of Rio de Janeiro (UFRJ/COPPE), Brazil. In 1998 he was awarded a scholarship from the Brazilian Government, through the National Council for Scientific and Technological Development (CNPq - Brazil, ref. number 200168/98-3), for pursuing his graduate study toward a Ph.D. degree at Columbia University. Since 1998 he has been a Ph.D. student in the COMET Group at the Center for Telecommunications Research, Columbia University, New York. His current research focuses on programmable virtual networking and resource management for virtual networks. He is a student member of the Brazilian Computer Society (SBC).

JOHN VICENTE (john.vicente@intel.com) was a visiting researcher in the COMET Group at the Center for Telecommunications Research, Columbia University, New York during which time he contributed to the Genesis Project. He is also actively engaged in the IEEE P1520 initiative for programmable interfaces for networks. He 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 received his M.S. in electrical engineering from the University of Southern California, Los Angeles, and his B.S. in computer engineering from Northeastern University, Boston, Massachusetts.

HERMANN G. DE MEER [M] (hdm@comet.columbia.edu) received his Ph.D. from the University of Erlangen-Nuremberg. He has been a postdoctoral fellow at Duke University and the University of Texas at Austin. He was appointed an assistant professor at the University of Hamburg in 1993. In 1998 he joined the Department of Electrical Engineering, Columbia University, New York, as visiting professor, having been awarded a research fellowship from the Deutsche Forschungsgemeinschaft (DFG). His research interests cover distributed computer and communication systems, QoS, and multimedia communications and performance modeling.

KAZUHO MIKI [M] (kazuho@crl.hitachi.co.jp) received his B.E. and M.E. degrees in electronics and communications from Waseda University, Tokyo, Japan, in 1990 and 1992, respectively. He joined the Central Research Laboratory of Hitachi Ltd. in 1992, where he is engaged in research and development of ATM switching and IP routing systems. Since 1998 he has been a visiting researcher in the COMET Group at the Center for Telecommunications Research, Columbia University. He is a member of IEICE of Japan.

KALAI S. KALAICHELVAN (kalai@nortelnetworks.com) is currently the director and general manager in Nortel Networks responsible for next-generation routing services software. In this role, his team will address architectural changes for IP networks and provide software solutions to meet fast time-to-market demands. He has been with Nortel Networks over ten years in various areas of software research and development and has been awarded the Nortel President's Award of Excellence on two occasions. He completed his Ph.D. in 1987 at the University of Toronto.