# The Genesis Kernel: A Virtual Network Operating System for Spawning Network Architectures

A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. Vicente, and D. A. Villela

*Abstract*

***The deployment of network architectures is often manual, ad-hoc and time consuming. In this paper we introduce a new paradigm for automating the life cycle process for the creation, deployment and management of network architectures and envision programmable networks 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 virtual networks. Child networks support the controlled access to communities of users with specific connectivity, security and quality of service requirements. In this paper we introduce the Genesis Kernel, a virtual network operating system capable of profiling, spawning and managing virtual network architectures on-the-fly.***

***Keywords— open programmable networks, virtual networks, distributed telecommunications systems***

## I. INTRODUCTION

The ability to rapidly create, deploy and manage new network services in response to user demands is a key factor driving the programmable networking community. Results from this field of research are likely to have a broad impact on customers, service providers and equipment vendors across a range of telecommunication sectors calling for major advances in open network control architecture, network programmability and distributed systems technology. Competition between service providers may hinge on the speed at which one provider can respond to new market demands over another. Currently, the creation and deployment of network services and architecture is a manual, time consuming and costly process. To the network architect the creation process is typically ad-hoc in nature based on handcrafted small-scale prototypes and rudimentary design and deployment tools that in most cases fail to ease the deployment process or highlight significant shortcomings in the design approach. As a result the network life cycle process is iterative lacking a systematic exploration of the network design space.

We believe that the design, creation and deployment of new network architectures should be automated and built on a foundation of programmable networks. The emergence of open programmable networks [18] [8] [4] [14] is enabling new approaches to the problem of service creation and support for multiple control architectures [16]. This is resulting in better network customization, resource control and time to deployment for new network services. We describe the process of automating the creation, deployment and management of new network architectures as "spawning" [15]. 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. CPU). We envision programmable networks as having the capability to spawn not processes but complex virtual network architectures. This is a departure from the operating systems analogy, where the parent and child typically share the same hardware. In this paper, we describe the *Genesis*[1] *Kernel*, a virtual network kernel capable of automating the virtual network life cycle process, that is, profiling, spawning and managing programmable virtual network architectures on-the-fly.

The paper is structured as follows. In Section II we present an overview of the Genesis Kernel and discuss principles that underpin our framework. We then describe the three main architectural components of the Genesis Kernel. We present a detailed discussion of the transport, programming and life cycle environments in Sections III, Section IV and Section V, respectively. Following this, we present the related work in Section VI and some concluding remarks in Section VII.

*Andrew T. Campbell and Michael E. Kounavis are affiliated with Center for Telecommunications Research, Columbia University, USA. Hermann G. De Meer is affiliated with University of Hamburg, a research fellow of DFG, Germany, and is a Visiting Professor at Columbia University. Kazuho Miki is affiliated with Hitachi Limited, Japan, and is a Visiting Researcher at Columbia University. John Vicente is affiliated with Intel Corporation, USA, and is a Visiting Researcher at Columbia University. Daniel A. Villela is affiliated with Center for Telecommunications Research, Columbia University, USA, and is a CNPq-Brazil Scholar, {campbell, hdm, mk, miki, jvicente, dvillela}@comet.columbia.edu*

[1] Genesis means 'the origin or generation of a thing'. In this context 'the thing' is a 'virtual network architecture'.

0-7803-5261-0/99/$10.00 © 1999 IEEE

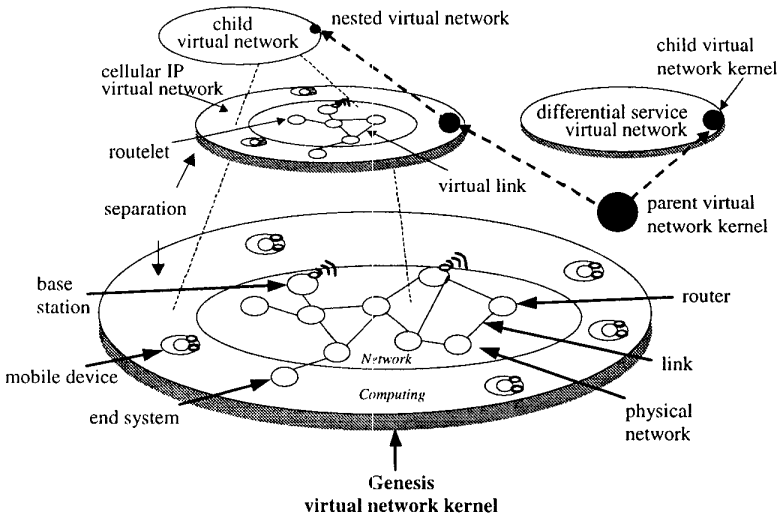


**Fig. 1: Spawning Virtual Network Architecture**

## II. Genesis Kernel

The Genesis virtual network kernel represents a next-generation approach to the development of programmable networks building on our earlier work on open programmable broadband [22][13][14] and mobile networks [3][17]. The Genesis Kernel has the capability to spawn child network architectures that can support alternative signaling protocols, communication services, quality of service (QOS) control and network management in comparison to their parent network architectures. We call a virtual network installed on top of a set of network resources a parent network. The parent virtual network kernel has the capability of creating "child networks" as illustrated in Figure 1. A child network operates in isolation on a subset of its underlying "parent network" resources and topology, supporting the controlled access to a set of users with specific connectivity, security, QOS and isolation requirements.

### A. *Architecture*

The Genesis Kernel based on [13] explicitly supports the virtual network life cycle process and a set of principles that underpin its design. Communication hardware and network resources are abstracted and represented as a set of distributed network objects that can be programmed by controllers to meet the demands of virtual network architectures. The Genesis Kernel acts as a resource allocator, arbitrating between conflicting requests made by spawned virtual networks.

Currently, routers bundle together the transportation of digital information with the algorithms that realize network control and management. These algorithms are usually implemented as a part of proprietary operating systems that execute on routers. Typically, the access to router hardware, state and algorithms is very limited. As a result routers represent a "closed box" to the introduction of new architectures, services and protocols. An alternative "open box" approach based on open programmable networking [18] argues for a clean separation between the network control and management architecture and the underlying transport hardware.

In what follows, we provide an overview of Genesis Kernel architecture as illustrated in Figure 2.

#### *1) Transport Environment*

At the lowest level, a *transport environment*[2] delivers packets from source to destination end-systems through a set of open programmable virtual router nodes called *routelets*. 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 while control algorithms (e.g., routing and resource reservation) are considered to be programmable using the virtual network kernel. Genesis routers are capable of supporting multiple routelets. Each routelet corresponds to components of distinct virtual networks that use router computation and communication resources.

[2] The term transport environment is used to refer to a set of transport modules that forward packets through the network and not to the algorithms that typically run in end-systems (e.g., flow control).

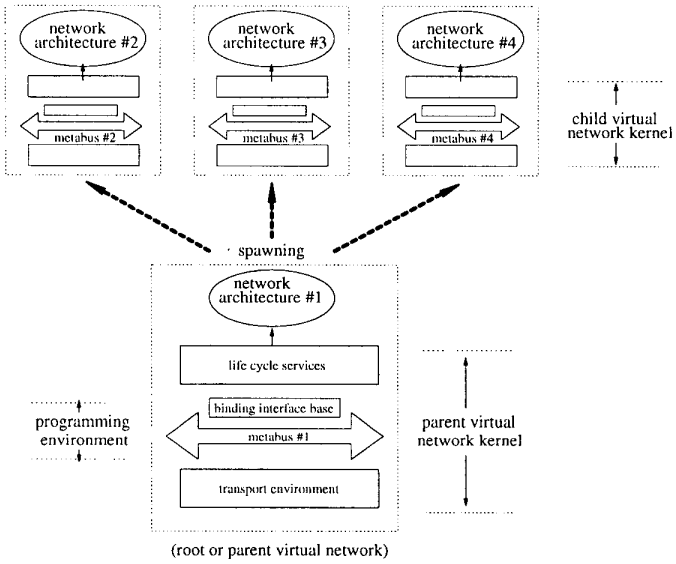


**Fig. 2: Genesis Architecture**

*2) Programming Environment*

Child routelets are instantiated by the parent virtual network during spawning. These routelets operate on a subset of the parent's resources but operate independently. In addition, routelets are controlled through separate programming environments. Each virtual network kernel can create a distinct *programming environment* that supports routelet programming and the interaction between set of distributed objects that characterize the spawned network architecture as illustrated in Figure 2. The programming environment comprises a *metabus*[3] that partitions the distributed object space supporting communications between objects associated with the same virtual network Each virtual network has its own metabus as illustrated in Figure 2. A *binding interface base* [1] supports a set of open programmable interfaces on top of the metabus as illustrated in Figure 2. These interfaces provide open access to a set of distributed routelets and virtual links that constitute a virtual network. The metabus and binding interface base also support a set of life *cycle services* as illustrated in Figure 2.

*3) Life Cycle Environment*

A key capability of Genesis is its ability to support a virtual network life cycle process that supports the dynamic creation and deployment of virtual network architectures through the process of:

- *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; and
- *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.

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 virtualization of the parent's 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.

[3] Metabus is a per-virtual network software bus for object interaction

### B. Design Principles

The Genesis Kernel is governed by a set of design principles.

#### 1) 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 being carried securely and independently from other networks. The allocation of parent network resources to support a child virtual network is coupled with the separation of responsibilities and the 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 resources in an autonomous manner based on its instantiated architecture.

#### 2) 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 long time-scale) 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). Child networks represent virtual network service subscribers where every child virtual network can be a parent (i.e., provider) to its own child networks. The parent-to-child relationship is represented by a 'virtual network inheritance tree' as illustrated in Figure 1. Two child networks are spawned by the parent wireline network. The first child network is a Cellular IP [20] virtual network that supports wireless extensions to the parent wireline network. The other child network (illustrated in Figure 1) supports a differentiated services architecture [25] operating over the same wireline infrastructure. An additional level of nesting is shown where the Cellular IP [20] network spawns a virtual network.

#### 3) 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 distributed object technology, 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 completely distinct from its parent thereby not using inheritance.

## III. 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 level of a virtual network.

### A. Routelet Architecture

The routelet operates like an autonomous virtual router that forwards packets from its input ports to its output ports, scheduling virtual link capacity and computation resources. Routelets support a set of transport modules that are specific to the spawned virtual network architecture as illustrated in Figure 3. A routelet comprises a forwarding engine and a control unit and a set of input and output ports.

#### 1) 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 virtual network architectures 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 plug-ins, having well defined interfaces and globally unique identifiers. Transport modules can be dynamically loaded through the control plane making the data path fully programmable. We define a generic set of transport modules that can be easily extended with new classes:

- *encapsulators*, which add specific headers (e.g., RTP, IPv4) to packets at the end-systems or routelet;
- *forwarders*, which execute particular packet forwarding mechanisms (e.g., IPv6, MPLS, Cellular IP) at the routelet;
- *classifiers*, which separate packets in order to receive special treatment by the routelet;
- *processors*, which process packets based on architectural

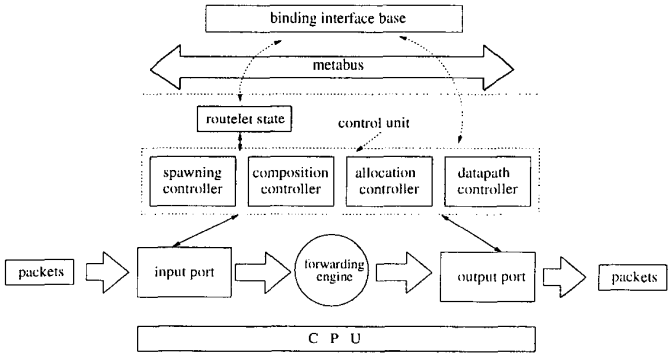


**Fig. 3: Routelet Architecture**

specific plug-ins (e.g., police, mark, monitor, shape, filter packets); and

- *schedulers*, which regulate the use of virtual link capacity based on a programmable buffer and queue management capability.

Once ports and engines have been dynamically created they are composed based on the profiling script. Child ports and engines may inherit directly from their parents transport modules or dynamically compose new modules on-the-fly. Forwarding engines bind to a number of input and output ports dynamically constructing a data path to meet the specific network architecture's needs. Ports are composed in a similar manner. 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, the input ports could contain differential service specific mechanisms (e.g., meters and markers used to maintain traffic conditioning agreements on the boundary routers of a differentiated service [25] virtual network). A virtual link is typically shared by user traffic generated by end-systems associated with the parent network and by aggregated traffic associated with child networks. This user and child network traffic contends for the parents' 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 [25].

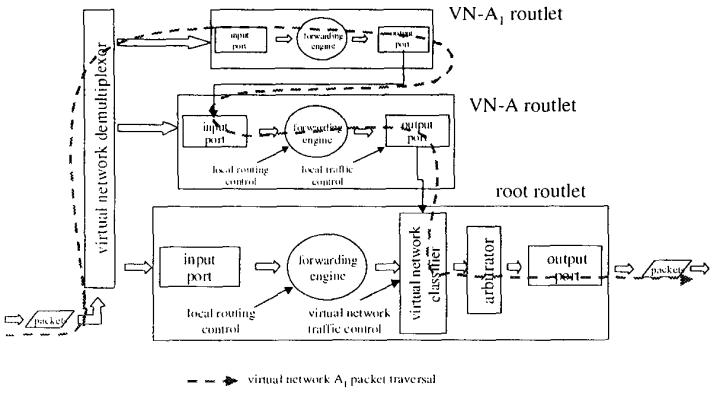


**Fig. 4: Nested Routelets**

In this case, all the necessary signaling for routing and service level agreement establishment can be made entirely programmable on top of the programming environment.

*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 the routelet using a set of transport module references and a composition graph to construct ports and engines;
- *an allocation controller*, which manages the computation resources of a routelet, and
- *a data path controller*, which manages the communication resources and the transportation of packets.

The spawning, composition and allocation controllers are common for all routelets associated with specific virtual networks. In contrast, data path controllers consist of a set of transport modules (discussed above) that are dynamically composed during the spawning phase based on a profile script. Data path controllers collectively manage these 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 switch resource partitions called switchlets [16] 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).

*B. Nested Routlets*

The nested routelet abstraction is a product of the virtualization process, resource partitioning and separation of control between parent and child in the context an inheritance tree. Child routelets are dynamically composed through the spawning process, which partitions a parent routelet's computation and communication resources to support the execution of a child routelet. Nested routelets operate on the same physical node and maintain their structure according to a virtual network inheritance tree discussed in Section II-B. 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 maintain transparency of operation and separation between child and parent networks. It maintains the autonomous nature of routelets. Routlets are unaware that packets are processed according to the 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 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 and eventually exit at the root of the tree onto the physical link.

We use an example scenario to illustrate how nesting is resolved in a Genesis router. As illustrated in Figure 4, the arrival of packets into the router must be appropriately demultiplexed to the root network or appropriate virtual network. A virtual network demultiplexor must identify the appropriate virtual network 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 the targeted virtual network routelet. The routelet's forwarding engine maps and forwards the packet to its output port for further packet treatment and/or virtual link scheduling as illustrated in Figure 4. When the packet exits the first level (VN-A1) routelet, mapping is performed between the child and the parent environments using the switchlet. In this instance mapping is through a series of composition controllers managing transport module references that represent 'connectivity' through a series of ports and engines. This mapping is performed at each layer (i.e., routelet) down to the root of the tree. At each parent network, local routelet traffic is multiplexed with child virtual network traffic. Multiplexing is managed by a virtual network classifier and specifically provisioned through virtual link capacity classes allocations (see Section V-C). An 'arbitrator' located at the parent's output port controls access to the parent's link capacity. Packet tree traversal and packet processing is driven by virtual network policy, which may be different at each routelet or virtual network. The packet traverses the nested hierarchy tree until it eventually is scheduled and exits onto the physical link.

## IV. Programming Environment

Routelets interconnect with virtual links forming a programmable data path for spawned virtual networks. Each network can program its own routelets and deploy communication architecture on top of them. We believe that IP network architectures can be made programmable using open interfaces that allow access to router internal components (e.g., routing tables, 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 for a variety of protocols to be programmed. The network programming environment's architecture is illustrated in Figure 6 and discussed below.

### A. *The Metabus*

A key enabling technology for programmable virtual networks is the metabus, which dynamically partitions the distributed object space supported by the Genesis Kernel. At

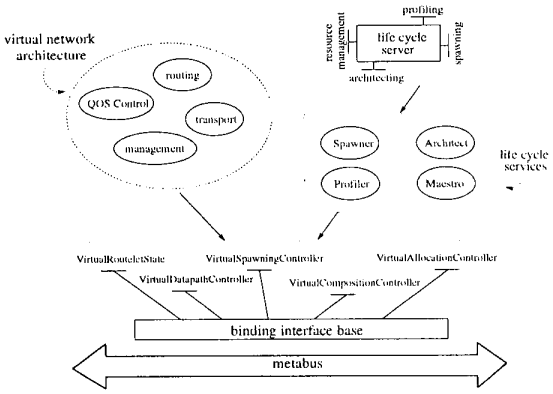


**Fig. 5: Programming Environment**

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.

The metabus is derived from the concept of a 'software' or 'object bus' found in distributed systems for the uniform interaction and communications of distributed objects. Typically, there is a single, monolithic software bus per distributed system for the interactions of all distributed objects contained in the system (e.g., each CORBA [34] system has a single software bus that is realized by a single Object Request Broker). The Genesis Kernel extends this idea to support multiple programmable virtual networks which operate in isolation from each other in the same distributed systems space. Each virtual network has its own metabus that has all the characteristics of a software bus but with one distinction as illustrated in Figure 2. Metabuses partition the distributed object space and by doing so create

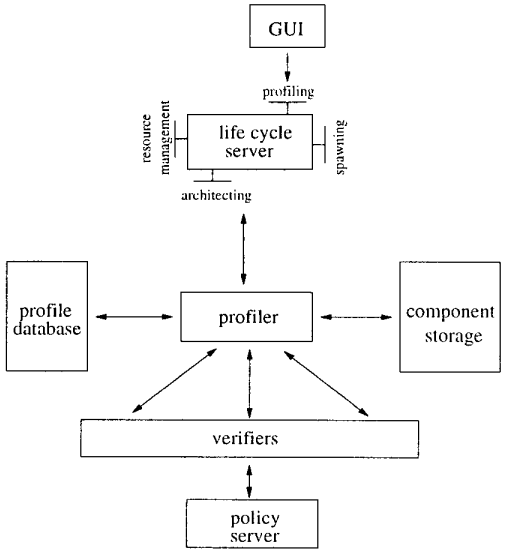


**Fig. 6: Profiling Architecture**

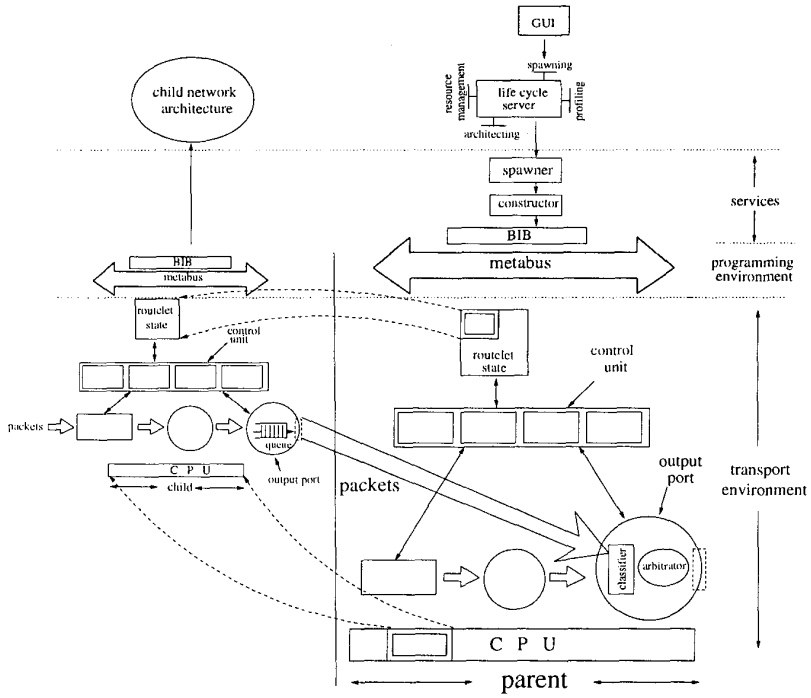


**Fig. 7 Spawning Architecture**

isolation between virtual networks and the distributed objects that represent them. This isolation allows unrestricted programmability of virtual network architectures on the transport, control and management planes[4].

The metabus is dynamically created as a part of the virtual network spawning process and operates on the resource space of the virtual network it is associated with. The spawning process therefore 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., routelets) objects that characterize a new child virtual network architecture. Note that routelets associated with the same virtual network use their metabus to exchange signaling messages.

The metabus is a general concept for programmable virtual networking. In our implementation we have chosen to realize the metabus abstraction as an *orblet*, a virtual Object Request Broker derived from the CORBA [34] object-programming paradigm. Both first generation kernels [13][3] that we have developed used CORBA technology for service creation, signalling and management.

[4] Planes are sets of mechanisms that support transport, control, and management in network architectures. The notion of separation between transport, control, and management is evident in the Internet. There is a single data path, but one can visualize transport (e.g., UDP), control (e.g., RSVP) and management (e.g., SNMP) mechanisms.

### *B. 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 the binding interface base objects provide a set of handlers to the routelet controllers and resources allowing the programmability of a range of IP architectures on top of a programming environment. The interfaces that constitute the binding interface base are illustrated in Figure 6. 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 routelet's transport structure (e.g., forwarding engine) is modified the binding interface base is dynamically updated to include the new module interface 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.

## V. Virtual Network Life Cycle Environment

The life cycle environment provides support for the profiling, spawning and management of virtual networks. Profiling, spawning and management provide 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.

### A. *Profiling*

#### *1) 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 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 computation requirements), user membership (e.g., privileges, confidentiality, 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.

#### *2) 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 sub-networks and cover a wide area interconnecting a number of Intranets or it may be restricted to a few local area networks. When profiling large-scale networks the architect has the flexibility to provide an outline of the topology graph, specifying strategic sub-networks or backbone routers that should be included in the topology. In this case a profiling tool [24] completes the specification. Once the topology is specified it is augmented with a user connectivity model and membership assignments.

#### *3) Architectural Components*

The Genesis profiling system offers a new capability to the network designer; that is, to explicitly select the protocols that should be included in a spawned architecture. A number of routing protocols for intra-domain and inter-domain routing are available at the component storage of the parent network (e.g., RIP, OSPF, BGP). QOS architectures based on well-founded models (e.g., integrated services [26] and differentiated services [25]) can be selected and used for the provisioning of QOS in virtual networks. Transport protocols (e.g., TCP, RTP, UDP) and management systems (e.g., SNMP) available as software building blocks for virtual network architecture can be adopted.

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 then form a network-wide topology. Ports and forwarding engines are composed to meet the specific architectural requirements of the virtual network architecture in terms of the characteristics of the instantiated transport modules that bind to create a low-level data path.

#### *4) 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' in Genesis 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; and
- *a best-effort capacity class*, which provides no explicit service assurances.

#### *5) Profile System*

The Genesis profiling system is illustrated in Figure 6. Network architects utilize a graphical utility profiling tool [24] to generate the virtual network profile. The profiling

system interacts with a parent virtual network kernel through a life cycle server as illustrated in Figure 6. 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 from 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 if 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 in source code file.

### *B. Spawning*

#### *1) 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 a network architecture 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 network 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.

#### *2) Spawning Services*

Spawning child virtual network architectures includes creating child transport and programming environments and then instantiating the transport, control and management network objects that characterize network architectures. Life cycle services are common services supported by the Genesis Kernel. Spawning services include: i) a spawner service, which applies centralized control over the spawning process interacting with the profiling and management services; ii) component storage, which is a distributed database of virtual network software building blocks; and iii) 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 that constitute a child network's routelets and network objects, which realize programmable control and management. The spawner "announces", the child network's bandwidth requirements to the maestro service (see Section V-C-2) 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.

#### *3) Spawning a Virtual Network*

Most of the creation process associated with spawning the child transport environment revolves around the process of a creating routelet and composing its data path then bootstrapping routelets into each physical node in the child network topology followed by binding the virtual links to routeletes. The nesting property is exhibited here as a child network inherits the life cycle support from its parent as discussed below. This results in the instantiation of the child transport environment over the parent network. Child routelet are bootstrapped by parent routelet's spawning controllers as illustrated in Figure 7. The spawning controller interacts with the allocation controller to reserve a portion of the parent routelet's computation 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, 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 binding the 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 controllers. A metabus is dynamically created supporting the child networks distributed object environment. Next, the metabus is initialized to support binding interfaces. The spawner and constructors load network objects from the 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.

*4) Illustrated Example: Spawning Cellular IP*

In what follows we present a simple illustrative example of spawning a child Cellular IP [20] access network and discuss the parent-child relationship as illustrated in Figure 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. Child networks are virtualized on the basis of the address space they use and the parent classifier is programmed to identify different streams of virtual network traffic from the range of the source and destination addresses carried in the headers of the packets that traverse the routelet. In our example scenario, the child network overwrites the standard IP forwarding engine of its parent and instantiates Cellular IP, a new IP protocol for supporting mobile applications. 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 network 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 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 port ports contain a low priority packet discards transport module to respond to congestion. Network traffic, which exits the child output ports enter the output ports of its parent. 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 devices at a routelet's forwarder are used to set up per-mobile soft-state paging and routing information which is used for any subsequent down link packet data delivery. A base station represents a special case of a routelet having 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 fading issues over longer time-scales at the wireless link. As illustrated, a parent routelet schedules the packets of child virtual network using a packet-by-packet generalized processor-sharing algorithm. This algorithm is supported by a virtual network 'capacity' scheduler called an arbitrator which is discussed in Section V-C-2.

## *C. Management*

*1) 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 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 management phase also subsumes the process of refinement of spawned network architecture by observing and controlling the dynamic behavior of virtual networks. Through the process of 'architecting[5]' a network designer uses visualization and management tools to analyze the pros and cons of the virtual network design space and through refinement can modify network objects that characterize the spawned network architecture.

*2) Architecting Virtual Networks*

During the management phase executing virtual networks can be selected and visualized either as a standalone network or in relation to the inheritance tree. Once selected, networks can be observed, re-provisioned, managed and architecturally refined on-the-fly. Management services include the ability to modifying existing architectures through dynamic plug-ins (e.g., replacing the Cellular IP routing, paging or handoff services) or by re-configuring existing network services (e.g., modifying Cellular IP system timers for better performance).

Architecting represents a methodology for automating the creation and delivery of network services and architectures allowing the network designer to add, modify or delete network services as required. This can includes the addition

[5] Cellular IP can be refined to optimally operate in pico-, campus- and metropolitan-area environments through architecting.

or removal of complete child virtual networks. Using distributed network objects that constitute the communication architecture the network architect communicates by way of visualization tools with the target virtual network kernel to exert control over executing virtual networks and their virtual infrastructure objects. The network architect can redesign the programmable communications architecture by modifying transport modules (e.g., ports, forwarding engines and control units through the virtual network's binding interface base objects). In addition, the network architect can override or extend the underlying computational services installing new programmable services supporting the programming environment.

*3) Virtual Network Resource Management*

Genesis virtual network resource management leverages the benefits of the kernel hierarchical model of inheritance and nesting delivering scalable virtual network resource management. The Genesis virtual network resource management system is governed by four basic design goals that include:

*i)* *dynamic provisioning*, which provision virtual network resources based on per-virtual network policy and slow time-scale resource re-negotiation;
*ii)* *capacity classes*, which provide general purpose 'resource pipes' allowing the underlying parent controller architecture to manage child traffic in an aggregated and scalable manner;
*iii)* *inheritance*, which allows child networks to inherit the resource management facilities and provisioning characteristics of their parents; and
*iv)* *autonomous control*, which allows child virtual networks to manage user level QOS independently from their parent/provider network.

*4) Architectural Components*

The Genesis virtual network resource management architectural model [6] comprises a number of distributed architectural elements as illustrated in Figure 8. With the exception of the delegate, arbitrator and monitor elements, all other architectural elements (viz. auctioneer, maestro) are fully distributed and can operate 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 its 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,

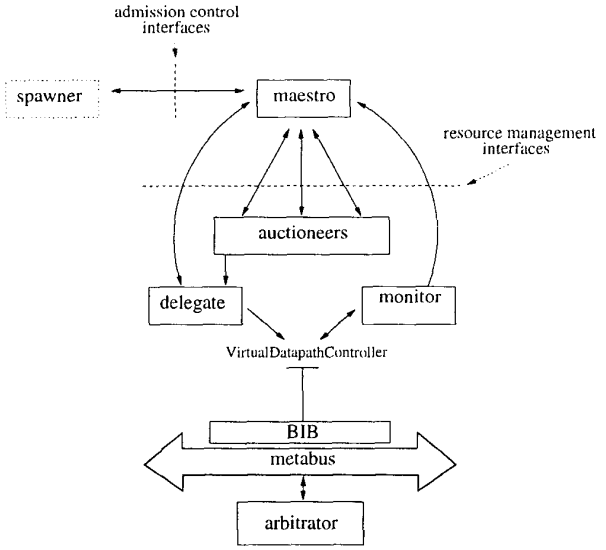


**Fig. 8: Virtual Network Management**

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 assures that child virtual networks are not consuming parent virtual networks 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 the completion of a resource allocation process. The virtual link arbitrator manages the access and control of the parent link packet scheduler based on the virtual network policy.

## VI. Related Work

The Tempest project [16] 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 as a resource divider. 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 programmable virtual network architectures.

Virtual private network services in broadband ATM networks have been the subject of a substantial amount of research. In [28], the concept of a virtual path group is introduced as a virtual network building block to simplify virtual path dynamic routing. 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 time-scales.

The X-Bone [19] 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 over the Internet. X-Bone overlays are not programmable, however.

The Supranet [10] 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 has investigated the deployment of multiple coexisting execution environments through appropriate operating system support [7] and an active network encapsulation protocol [2]. 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 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 the 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 [12]. These techniques have been used in a variety of commercial products [29] [30]. Finally, a number of IETF proposals have investigated IP virtual private networks [31] [32] [33]. Others have addressed issues of performance [27].

## VII. Conclusion

In this paper we have introduced the Genesis Kernel, an operating system capable of spawning virtual network architectures on-the-fly. We believe that the design, creation and deployment of new network architectures should be automated and supported by programmable networks. We argue that this will result in better network customization, resource control and time to deployment for new network architecture, services and protocol. The Genesis Kernel presents a new approach to the deployment of network architectures through the automating of the virtual network life cycle.

## Acknowledgement

The authors would like to thank the COMET Group industrial sponsors for supporting this work. In particular, we would like to thank the Intel Corporation and Hitachi Limited for supporting the Genesis Project. 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. Miki Kazuho (Hitachi, Ltd) 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).

## References

[1] Adam, C. .M., et al., "The Binding Interface Base Specification Revision 2.0", OPENSIG Workshop on Open Signalling for ATM, Internet and Mobile Networks, Cambridge, UK, April 1997.

[2] Scott, A. D., Braden, B., Gunter, C. A., Jackson, A. W., Keromytis, A. D., Milden, G. J., and Wetherall, D., "Active Network Encapsulation Protocol (ANEP)", Active Networks Group Draft, July 1997.

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

[4] Biswas, J., et al., "Application Programming Interfaces for Networks", IEEE P1520 Working Group Draft White Paper, www.ieee-pin.org.

[5] Session on "Enabling Virtual Networking", Organizer and Chair: Andrew T. Campbell, OPENSIG '98 Workshop on Open Signaling for ATM, Internet and Mobile Networks, Toronto, October 5-6 1998.

[6] Campbell, A. T., Vicente, J., Villela, D., "Virtuosity: Performing Virtual Network Resource Management", Technical Report, Comet Group, Columbia University, December 1998.

[7] Calvert, K. et al, "Architectural Framework for Active Networks", AN Working Group Draft, July 1998.

[8] DARPA Active Network Program, www.darpa.mil/ito/research/anets/projects.html, 1996.

[9] Sushil da Silva, Danilo Florissi and Yechiam Yemini "NetScript: A Language-Based Approach to Active Networks", Technical Report, Computer Science Dept., Columbia University January 27, 1998

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

[11] The Genesis Project: Programmable Virtual Networking comet.columbia.edu/genesis , 1998.

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

[13] 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.

[14] Lazar,A.A., "Programming Telecommunication Networks", IEEE Network, vol.11, no.5, September/October 1997.

[15] Lazar, A.A. and Campbell, A. T., "Spawning Network Architecture", White Paper, Center for Telecommunications Research, Columbia University, comet.columbia.edu/genesis Janurary 1998.

[16] Van der Merwe,J.E., Rooney,S., Leslie,I.M. and Crosby,S.A., "The Tempest - A Practical Framework for Network Programmability", IEEE Network, November 1997.

[17] The mobiware toolkit source code distribution comet.columbia.edu/mobiware

[18] OPENSIG Working Group comet.columbia.edu/opensig/

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

[20] Valko, A. G., Campbell, A. T., Gomez, J., "Cellular IP", INTERNET-DRAFT, draft-valko-cellularip-00.txt

[21] Vicente, J., Campbell, A.T., De Meer, H., A.T., Kounavis, M. and Miki, K, "The Genesis Project: Toward Programmable Virtual Networking", Proc. OPENSIG'98 Workshop on Open Signalling for ATM, Internet and Mobile Networks, Toronto, Canada, October 1998.

[22] The xbind broadband kernel code, http://comet.columbia.edu/xbind, 1996.

[23] Hui Zhang, "Implementing Quality of Service Virtual Network Service (VNS) on CAIRN", project site: www.cs.cmu.edu/~hzhang/VNS

[24] Zhaung., H., and R. Treadway "Genesis Profiler Tool", prototype software distribution comet.columbia.edu/genesis.

[25] Blake, S., Bernet, Y., Binder, J., Carlson, M., Keshav, S., Davies, E., Ohlman, B., Verma, D., Wang, Z., Weiss, W., "A Framework for Differentiated Services", draft-ietf-diffserv-framework-01.txt.

[26] Wroclawski, J., "The use of RSVP with IETF Integrated Services", RFC 2210, September 1997.

[27] Duffield N., Goyal, P., Greenberg, A., Mishra, P., Ramakrishnan, K.K., Van der Merwe, J., Doraswamy, N., Jagannathet, S., "A Performance Oriented Service Interface for Virtual Private Networks", draft-duffield-vpn-qos-framework-00.txt. Work in progress.

[28] Chan, M.C., Lazar, A.A. and Stadler, R., "Costumer Management and Control of Broadband VPN Services", Fifth IFIP/IEEE International Symposium on Integrated Network Management, San Diego, CA, May 1997, pp. 301-314.

[29] Compatible Systems, www.compatible.com

[30] Cisco Systems, "Quality of Service for Virtual Private Networks", White Paper, www.cisco.com/warp/public/779/largeent/vpne/qsvpn_wp.htm

[31] 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.

[32] Heinanen, J., "VPN support with MPLS", draft-heinanen-mpls-vpn-01.txt, INTERNET-DRAFT, Work in progress.

[33] Rosen, E., "BGP/MPLS VPNs", draft-rosen-vpn-mpls-01.txt, INTERNET-DRAFT, Work in progress.

[34] OMG, The Common Object Request Broker: Architecture and Specification, Rev. 2.3, November 1998.