

# Virtuosity: Programmable Resource Management for Spawning Networks

*Daniel Villela and Andrew T. Campbell*  
Center for Telecommunications Research,  
Columbia University, USA

*John Vicente*  
Intel Corporation and Center for  
Telecommunications Research,  
Columbia University, USA

## Abstract

*The creation, deployment and management of network architecture is manual, time consuming and costly. To the network architect the creation process is ad-hoc in nature, based on hand crafting small-scale network prototypes that evolve toward wide scale deployment. We envision a different paradigm where ‘spawning networks’ are capable of profiling, spawning, architecting and managing distinct virtual network architecture. This paper provides an overview of the Genesis Kernel framework and its life cycle of spawning virtual networks, and focuses on the design, implementation and evaluation of virtuosity, a plug-in module for the Genesis framework that supports programmable resource management of spawned virtual networks. The design goal of virtuosity is to minimize the complexity of handling multiple spawned virtual networks that potentially operate over multiple resource management timescales. We present a spawning network emulation environment and a virtuosity simulation system. Both of these components are implemented as virtual network extensions to ns. Through simulation we evaluate programmability, signaling, timescales, measurement-based estimation and traffic metering of virtual networks using virtuosity.*

*Index words: programmable networks, virtual networks, resource management*

## 1 Introduction

The rapidly evolving nature of the application base, service demands and underlying network technology presents a significant challenge to the deployment of new network architectures. This challenge calls for new approaches to the way we design, develop, deploy and analyze next-generation network architecture in response to future needs and requirements. Currently, the creation and deployment of network architecture is manual, time consuming and a costly process. To the network architect the creation process is typically ad-hoc in nature, based on hand crafting small-scale prototypes that evolve toward wide scale deployment. We envision [1] a future communication middleware platform capable of profiling, spawning, architecting and managing distinct virtual network architecture. Virtual networks can be viewed as customized communication environments allowing controlled access to groups of users with specified requirements for connectivity, privacy, isolation and service demand provisioning.

Recently, the growth of the Internet has given rise to the demand for more sophisticated provisioning of virtual private networks. Existing solutions, however, are inflexible at supporting the introduction of distinct network architecture above and beyond the ‘root’ network architecture (e.g., best effort IP architecture, MPLS, etc.). We believe that the design, creation and deployment process for realizing new architectures must be automated and built on a foundation of programmable networks. The emergence of open programmable networks [2] is enabling new approaches to the problem of service creation and

support for multiple control architectures [3]. This results in better network customization, resource control, service delivery and flexible traffic treatment. We describe the process of automating the creation, deployment and management of new network architectures as “spawning”. 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 the child process. The child process inherits its parent’s attributes typically executing on the same hardware (i.e., the CPU). We envision communication middleware associated with *spawning networks* as having the capability to spawn not processes but complex network architectures. As the term spawning implies, we believe that the power of distributed systems technology needs to be brought to bear to dynamically create, deploy, manage and architect new network architectures. In [1] we described the Genesis Kernel framework that provides middleware support for spawning distinct virtual network architectures.

A key component of the Genesis Kernel is the resource management of spawned virtual networks. In this paper, we describe *virtuosity*, a kernel pluggin that minimizes the complexity of handling multiple spawned virtual networks over the same physical network hardware. The virtuosity architectural model [4] comprises a number of distributed elements. These elements are instantiated as part of the child virtual network kernel during the spawning phase and are deployed as a set of distributed objects. Virtuosity leverages the benefits of the kernel’s hierarchical model of inheritance and nesting with the goal of delivering scalable virtual network resource management. The virtuosity system is governed by four basic design goals that include slow time-scale dynamic provisioning, capacity classes, inheritance and autonomous virtual network control.

The structure of the paper is as follows. In Section 2, we present an overview of spawning networks and the Genesis Kernel. In Section 3, we discuss the virtuosity framework, which forms an integral part of the kernel. Following this, we present an initial implementation of virtuosity based on virtual network extensions to ns [5] in Section 4. We present an evaluation of the virtuosity system in Section 5 and discuss open research issues in Section 6. Finally, we present some concluding remarks in Section 7.

## 2 Spawning Networks

We call the virtual network kernel installed on top of the network resources the “parent virtual network kernel”. We propose the realization of a parent virtual network kernel with the capability of creating a “child virtual network kernel” operating on a subset of network resources, as illustrated in Figure 1. This is a departure from the operating system analogy where the parent and child typically share the same hardware.

The partitioning of the network resources and the architecture controlling those resources (i.e., the child virtual network kernel) are spawned by the parent virtual network kernel. 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 Cellular IP [6] virtual network or Differentiated Services [7] virtual network, as illustrated in Figure 1. In the Cellular IP spawned virtual network, the wired network represents the “parent” and the wireless network the “child”. Typically, spawned network architectures support alternative transport, signaling, traffic treatment and network management capabilities in comparison to their parent architecture. Alternatively, child networks may inherit the same set of distributed algorithms as their parents.

The Genesis virtual network kernel can dynamically build virtual network architectures over the physical network through the deployment of virtual network infrastructure. The parent virtual network kernel “bootstraps” the child virtual network, then creates a set of routelets and virtual links that constitutes the

virtual network topology. Virtual links are partitioned bandwidth segments interconnecting routelets. Routelets represent virtual routers in the virtual network topology capable of forwarding packets based on instantiated virtual control objects. This includes a set of distributed objects that encapsulate transport (e.g., TCP, RTP) and network (e.g., IPv4, IPv6) services and protocols, signaling (e.g., RSVP, in-band approaches), control (e.g., differentiated services, best effort) and management (e.g., SNMP, CMIP) objects. These objects and the spawning capability are encapsulated in the routelet object abstraction.

The child, like its parent, inherits the capability to spawn other virtual networks creating the notion of spawning “nested” virtual networks within a virtual network. This is consistent with the idea of dynamically creating a virtual network infrastructure that supports relatively long lived (e.g., a corporate virtual network that operates over long timescales) and short lived (e.g., collaborative group networking operating within the context of the corporate virtual network but is only active for a short period) virtual networks, as illustrated in Figure 1. Virtual networks represent child networks (i.e., subscribers) where every child virtual network can be a parent (i.e., provider) to its own children and the depth of the virtual network inheritance tree (as illustrated in Figure 2) is determined by the overall availability of resources and the child network requirements.

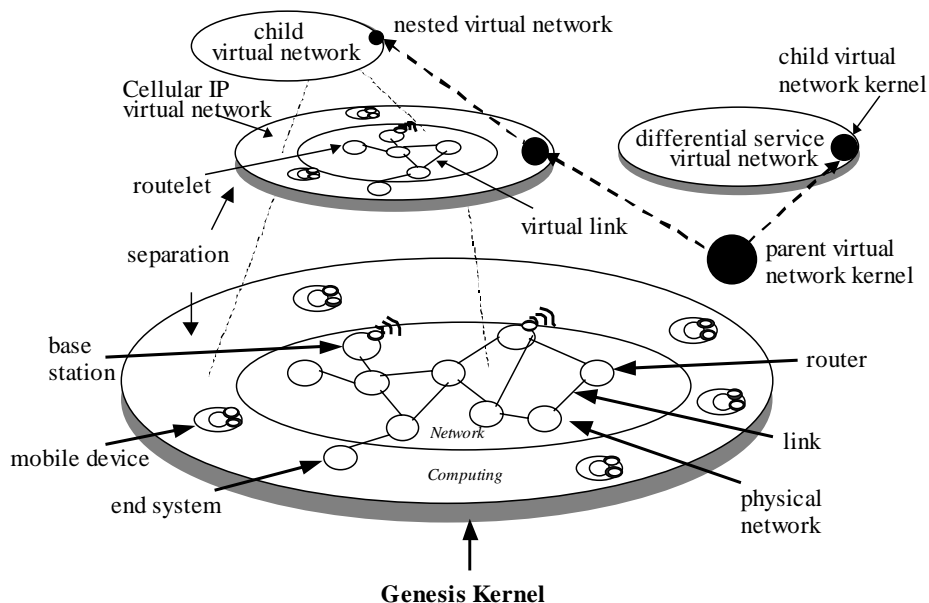


Figure 1: Spawning Networks

The Genesis Kernel is driven by a life cycle, which supports the dynamic creation and deployment or spawning of virtual network architectures through:

- *profiling*, which captures the ‘blueprint’ of a 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 the virtuosity framework, which exerts 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’s design space.

For details on the Genesis Kernel framework and experimentation with spawning networks see [1] and [18], respectively. The contribution of this paper is the design and evaluation of the virtuosity kernel plug-in.

### 3 Virtuosity Framework

The Genesis Kernel creates a natural hierarchy through partitioning and isolation of virtual networks; promoting inheritance and the autonomous control of network resources. Virtual networks are formed hierarchically through nested parent-child formations along a virtual network inheritance tree structure, as illustrated in Figure 2. The complexity of spawning virtual networks and managing nested virtual networks can be reduced by building a well-organized hierarchy over the physical network and adopting inheritance and state aggregation techniques. These design attributes allow virtuosity to scale well as the number of virtual networks grows. In this respect, we feel that the virtual network inheritance tree model is preferable to constructing virtual networks based on ‘flat’ models.

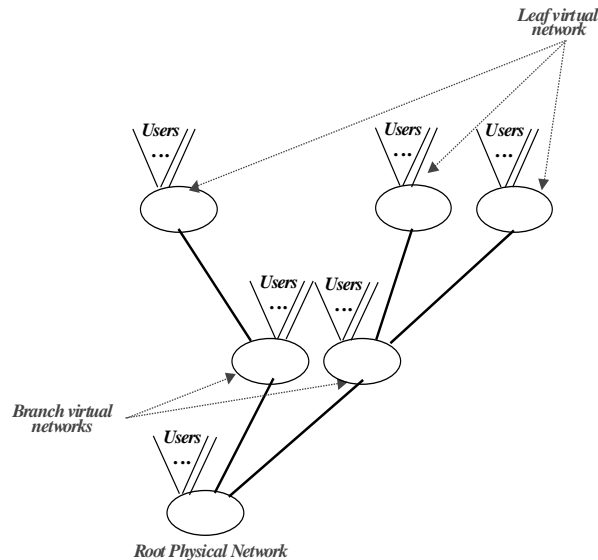


Figure 2: Virtual Network Inheritance Tree

#### 3.1 Design Characteristics

The virtuosity architectural model is embedded in the Genesis Kernel and supports the concept of virtual network resource management through the virtual network life cycle.

The virtuosity framework is based on the following design principles:

- *Autonomous Control*

Spawning results in the composition of a child virtual network architecture, partitioning of parent network resources in support of a child's resource needs, the separation of responsibilities and transparency of operation between parent and child architectures. Once a child network has been spawned, the child has complete freedom to manage its own resources and users' quality of service (QOS) in an autonomous manner based on its instantiated architecture. Spawning supports this separation and the notion of autonomous control between the parent and child architectures.

- *Dynamic Provisioning*

Virtuosity supports the dynamic provisioning of virtual network resources. Currently, the provisioning of virtual network resources is limited to either static or policy-based provisioning [8]. Virtuosity supports a different form of provisioning where the capacity needs of individual virtual networks may change more dynamically in term of timescales and events. Virtuosity employs a per-virtual network policy-based approach that can be programmed by the subscriber to support a wide range of dynamic resource provisioning strategies.

- *Capacity Classes*

Virtual links support capacity classes within which child traffic classes are mapped and multiplexed. Capacity classes provide general purpose 'resource pipes' allowing the underlying parent controller architecture to deal with child traffic in an aggregated manner. This approach is similar to the concepts of resource management in differentiated service networks [7] but with application to virtual networks. The mapping of the child QOS to parent capacity classes is made transparent to the parent and is the responsibility of the child virtual network architecture. Virtuosity supports the following capacity classes:

- 1) *constant capacity*, which statically allocates bandwidth to virtual links based on a peak rate specification providing resource isolation and an assured rate virtual link service;
- 2) *controlled capacity*, which allocates bandwidth based on an average rate specification to virtual links providing resource sharing with other controlled capacity virtual links and a statistical rate virtual link service; and
- 3) *best-effort capacity*, which allocates bandwidth based on some fixed amount for the transmission of all best effort traffic across a virtual link with no explicit service assurance.

- *Inheritance*

Through distributed object technology, resource management inheritance allows a child virtual network to transform itself to serve as a providers; giving it resource management capabilities and provisioning characteristics of its parent or, alternatively, to spawn completely distinct capabilities. Through inheritance, aggregation and the provisioning of a common set of capacity classes, virtuosity can efficiently support the resource management needs of multiple child virtual networks in the same manner that it handles new client (i.e., local end-system) resource needs. The nesting process allows us to push the complexity of the management of virtual network resources up the inheritance tree into the child virtual network, with the benefit of only having to manage reduced state information.

Virtuosity manages and controls virtual network resources on a slow performance management [9] timescale that operates in the order of minutes to tens of minutes. We argue that this is a suitable timescale for virtuosity to operate over while allowing virtual networks to perform dynamic provisioning, as needed. We believe that it is unlikely that virtual networks would operate in a stable manner over faster timescales [4]. The combination of slow timescale control, dynamic provisioning, inheritance, aggregation and the adoption of a small set of common capacity classes within a virtual network inheritance tree makes virtuosity flexible and scalable.

## 3.2 Architectural Components

The virtuosity architecture comprises a number of components that include *maestros*, *delegates*, *brokers*, *arbitrators* and *monitors*, as illustrated in Figure 3. These programmable elements are instantiated as part of the child virtual network kernel during the spawning phase and are deployed as a set of distributed plug-in objects. The maestros, delegates, brokers and monitors elements operate in the “management plane” while the arbitrator operates in the “data plane”, as illustrated in the figure. Through the process of virtualization, virtual networks are separated from the physical or parent virtual network within a partitioned and separate name and address space.

Virtuosity operates within the child and parent virtual network kernels managing the partitioned resource space and interfacing with the parent virtuosity system to increase or decrease the current partitioned resource space through dynamic provisioning. Within a partitioned virtual network an arbitrator and monitor are instantiated on all routelet ports, managing the integration of provisioned capacity and local resource policy over each routelet virtual link. A single delegate and broker are instantiated on a per routelet basis and manage local resource management activities. The delegate is a coordination proxy working on behalf of the maestro to distribute local per routelet or per port activities, while the broker negotiates the provisioning (i.e., outgoing capacity classes on virtual links) needs of child network routelets. The maestro is the only virtuosity element that oversees the entire resource domain. While managing domain resources, the maestro seeks optimal global policy (i.e., global and local resource provisioning, pricing) and distributes (via per routelet delegates) per routelet resource provisioning parameters and per virtual link arbitration policy. Although conceptually a centralized controller, the maestro can be implemented as a centralized controller or as a set of decentralized cooperating agents instantiated on a per routelet basis.

### 3.2.1 Distributed Virtual Network Control

The maestro coordinates virtual network control through a set of distributed virtuosity components performing virtual network monitoring, renegotiation-based resource allocation and capacity-based scheduling, all of which operate over management-level timescales. The maestro uses dynamic provisioning of virtual network resources to meet the changing needs of its child networks (captured by child per-virtual network policy) and to react to changes in its global state; that is, it dynamically responds to the changing needs of its own virtual network (e.g., changing the resource needs of its clients), spawned child networks and underlying parent network. The maestro interacts with its local broker via a delegate to handle the dynamic provisioning of its own virtual network in relation to its child networks' needs. Upon the aggregation of client and child network resource demands, the maestro interacts with its parent network broker renegotiating resources to support updated demands.

The maestro may also influence the way in which resources are allocated to its child networks by optionally setting market pricing strategies [11] or alternative resource allocation strategies, for example, under provisioning its own virtual link resources but overbooking resources to child networks to maximize revenue for controlled capacity traffic. Because the maestro maintains global state of its virtual network resources, being cognizant of the overload or under-use of its own virtual network links, it can influence child networks to respond to this global state via optimization strategies. The maestro can, for example, influence a child network's routing mechanisms to reroute traffic along a more cost effective path (e.g., an underutilized path) in the child virtual network, hence, improving the utilization of its own resources and potentially maximizing the revenue it can earn as a provider.

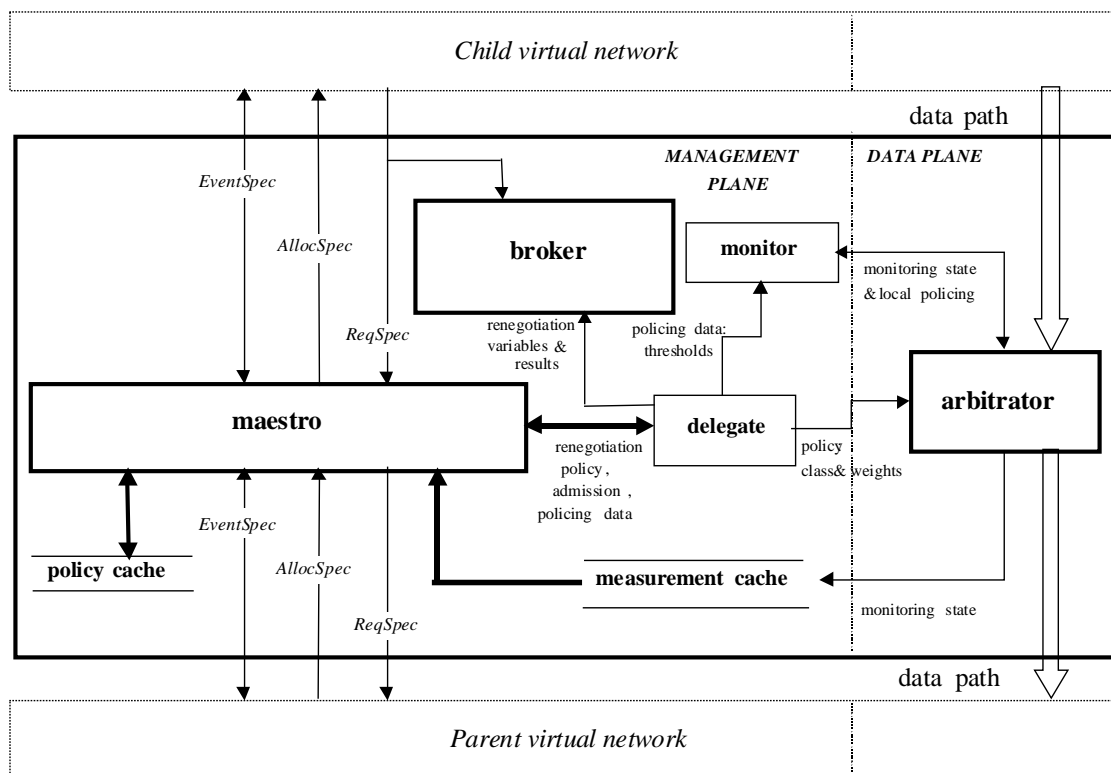


Figure 3: Virtuosity Framework

During the spawning phase of a child network the maestro conducts a virtual network admission control test based on the resources requested by a child network. If the admission control test is positive then the parent provider network admits the child network and allows it to become a participant in the resource allocation process controlled by the broker and governed by the child virtual network's policy. Admission control is coordinated by the parent maestro using its virtual network inheritance tree. The parent maestro receives a request for admission from a child virtual network and determines if sufficient resources are available within the context of its own network to meet those new demands. If this is the case, it indicates that the parent has sufficient residual capacity in its own right to accommodate the child's needs. Virtuosity implements a measurement-based virtual network admission control test. By monitoring the available capacity along its virtual links, the maestro can determine if allocated virtual network resources are underutilized. Based on this measurement state information (which is maintained in the measurement cache) and capacity threshold violations (which indicate the number of times a monitored capacity has been violated) the maestro can allocate underutilized resources based on the capacity class, bandwidth and policy requested by a new virtual network. A child network is admitted and allowed to participate in the resource allocation process if capacity is available. In the case where the parent network has insufficient resources to accommodate a new child network then the parent needs to renegotiate its resource needs with its own parent (and hence its provider) at the next level down its virtual network inheritance tree. In this case, the provisioning request is submitted by the maestro to its parent broker to cover the additional resource demands. This request enters the parent provisioning process, following a successful admission control sequence that traverses the inheritance tree until a provider can accommodate the new demands.

The maestro can leverage longer timescale measurements to proactively plan resource capacity using the parent's resource allocation system. By staying ahead of the demand growth of its local client base and child network, the maestro can leverage capacity planning techniques used today to adjust the capacity class provisioning demands. This results in better management of demand fluctuations, potentially reducing the number of times admission control needs to 'traverse' the virtual network inheritance tree to accommodate new child virtual networks capacity needs.

The maestro maintains global state information in a distributed measurement cache. Parent and child networks aggregate resource states and capacity class usage compiling profiles that summarized the state information over several timescales, (e.g., per resource allocation interval, minutes, and hours). The maestro leverages measurement cache to influence brokers in optimizing the allocation of their resources to child virtual networks. A management event-based system allows parents to setup dynamic event invocations under conditions related to the admission of a new virtual network, resource usage inefficiencies or renegotiation of parent resources (up or down) due to, for example, capacity class demand fluctuations. New events precede the resource allocation process, whereby child virtual networks vie for resources.

The maestro also cooperates with local delegates, which distributes resource allocations to each child virtual network, and then distributes local policy to arbitrators for mediating and policing usage of local parent resources used by child networks. Policing thresholds are set based on either capacity class allocations over the last provisioning period or long-standing provisioning contracts [4]. Monitors assume the responsibility of policing these thresholds noting any violations in the measurement cache. Stored violation data is used by the maestro to inform or treat non-conforming child networks over the next provisioning interval.

### **3.2.2 Renegotiation-based Resource Allocation**

We propose a virtual network resource allocation process based on supply and demand of virtual network services where competing child virtual networks, working on behalf of a community of users and through appropriate specification, request resources and pay for such services to a provider of virtual network services. There are inherent behaviors and objectives that dictate the economics, and more importantly, the effective allocation, partitioning and utilization of such services. We argue that the provider (i.e., parent virtual network) and subscriber (i.e., child virtual network) behavior, and their corresponding objectives serve as fundamentals that can be leveraged for resource maximization through the influence of economic variables. Network providers seek to achieve resource efficiency through the effective utilization of link resources using price-based, load balancing and the addition of new virtual network subscribers. Clearly, it is in the best interest of the parent to seek a greater number of subscribers for a given set of resources without impacting service requirements, in order to increase revenues. On the other hand, subscribers serve primarily the global interests of the virtual network users, but secondly seek to maximize the use of virtual network resources at a minimal cost. In addition, virtual network users negotiate with virtual network subscriber to request and achieve their required QOS. The competing nature that both the provider (parent) and subscriber (child) exhibit, we argue, should create the necessary dynamics that leads to a more aggressive environment for achieving resource efficiency.

The interface between the customers and the virtual network provider is achieved through the provider maestro when the customer requests a virtual network. The maestro interfaces with the broker to coordinate the resource allocation process, which occurs on a periodic or static deadline basis. This period is driven by slow timescale considerations in the order of tens of minutes. This allows the provisioning process to reach equilibrium and maintain constant services over longer timescales. This slow time scale is motivated by these factors, as is the virtual network creation timescale. While we believe that short-lived virtual networks will exist the vast majority of 'virtual network holding times' will be over longer

durations. This indicates that virtuosity will only be invoked on timescales associated with virtual network creation and removal, which will be far longer than today’s call holding times.

### 3.2.3 Dynamic Capacity Scheduling

A key component of virtuosity is its capacity scheduling capability. A virtual network scheduling abstraction resident at each parent resource arbitrates child virtual network access to the parent virtual links. The capacity arbitrator is based on a set of virtual network capacity classes and capacity class weight policies that are distributed to the arbitrator by delegates on behalf of the maestro. Capacity classes represent virtual network differentiated policy for provisioning capacity. The class weights are calculated by the maestro based on the following parameters: rate\_allocation and percentage (or optionally price factors). These policies may be updated and are distributed recursively at each provisioning cycle upon completion of the resource allocation process. The capacity classes and weights translate the resource allocation negotiated by individual child virtual resources during the resource negotiation process to a set of virtual capacity scheduling policies. These policies are then used to differentiate child virtual network capacity allocations and the ordering of packet delivery to the parent link resource.

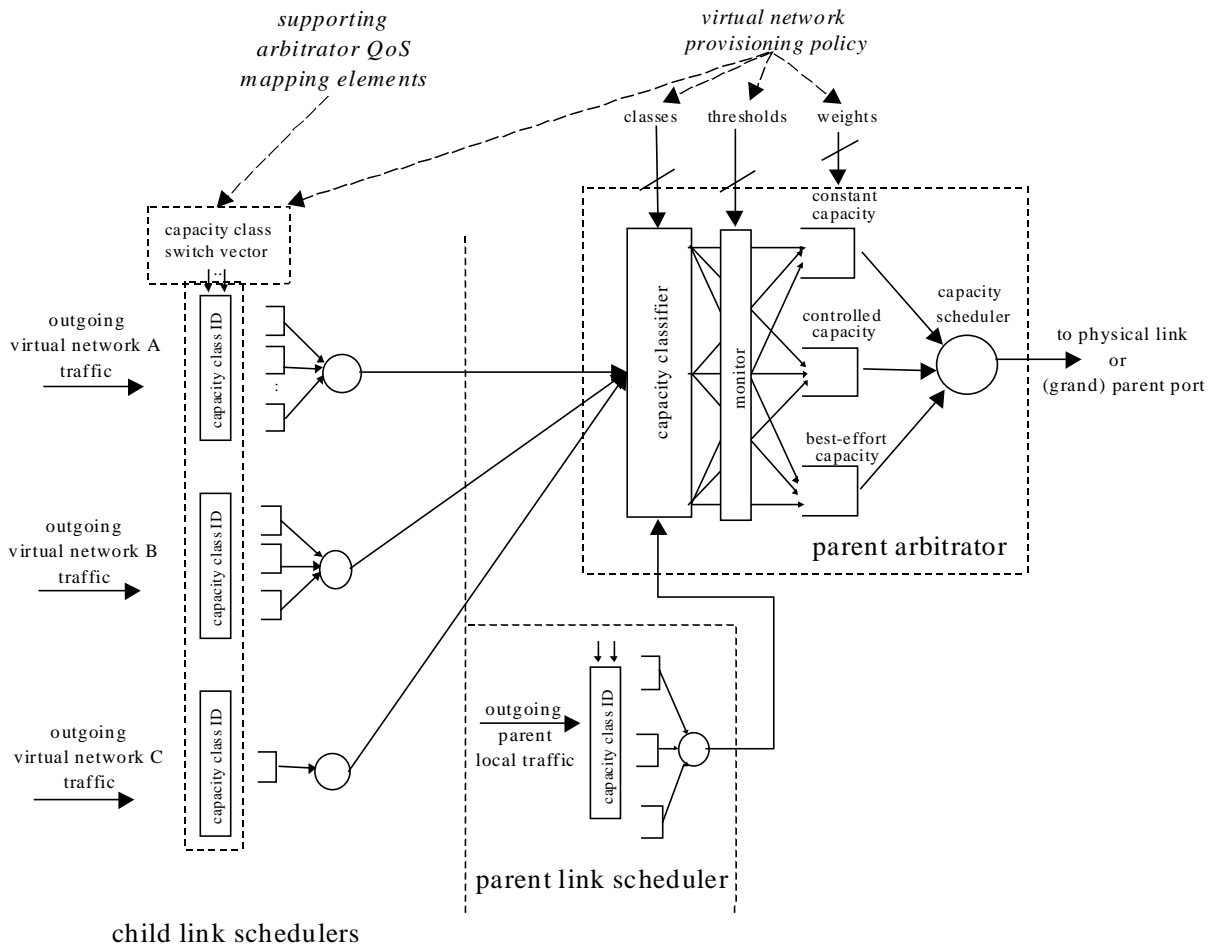


Figure 4: Arbitrator Architecture

Each child virtual network is assigned a unique virtual network ID to distinguish its traffic from other child network traffic. Also, by way of the spawning process, a *capacity class identifier* function is introduced into the arbitrator architecture prior to the child's link scheduler function to deliver the specific QOS treatment (e.g., best-effort, controlled load, expedited forwarding, etc.) associated with the child QOS architecture. This function interworks with a *capacity class switch vector* to assign each packet a *stamp* that associates it with a particular capacity class prior to its arrival at the child's link scheduler, as illustrated in Figure 4. A switch vector is formed from the required capacity classes requested during the provisioning process and allocated to the child network during the resource allocation process. If, for example, a customer or child network supports only best-effort IP traffic classes within a spawned child network and provisions for constant capacity then the switch vector would stamp all traffic with a constant capacity classification. On the other hand, if the customer supports integrated services classes (viz. guaranteed, controlled load and best-effort services) within a spawned child network and provisions for all three capacity classes then the class identifier would stamp the traffic with corresponding capacity classifications by default.

The introduction of the virtuosity arbitrator at the output port merges both child and parent QOS scheduled traffic and provides coarse capacity scheduling of the composite traffic based on the allocated provisioning policies. A capacity classifier is used to identify virtual networks and their capacity classes. The classifier queues incoming (stamped) packets (from the output of child network link schedulers) to the appropriate capacity queue structures (viz. constant, controlled, and best-effort). Individual queues are created for each child virtual network within an allocated capacity queue structure. Each virtual network queue is then assigned an appropriate weight based on the policy previously negotiated and distributed by the maestro. Within the provisioned interval, the arbitrator manages the scheduling of virtual network control based on the capacity class priority and weights, allowing child networks (and local user traffic) to queue available packets to the parent's output link scheduler. The capacity arbitrator leverages space (i.e., available resource bandwidth), time (i.e., provisioning interval) and capacity-class abstractions to manage the scheduling of its own user traffic and packets from child virtual networks onto the parent link scheduling facility. The capacity arbitrator services the capacity queues in priority order and weighted round-robin for the same capacity class queues.

It is important to note that the current illustration represents the default virtual network resource management implementation and is not the only parent option for managing child network traffic. Alternatively, the parent may override the arbitrator function and integrate child network traffic through its local routelet port link scheduler. In this case, the child architecture is based on the parent's resource management policy set during the virtual network profiling phase and composed during the spawning phase of the life cycle process.

## 4 Implementation

We have implemented virtuosity and a virtual network emulator through extensions to the ns simulator [5]. The virtual network emulator partially supports the Genesis Kernel's ability to spawn virtual networks. The emulator is limited to constructing virtual networks from a small set of predefined resource management schemes. By emulating the creation and deployment of spawned virtual networks, we can examine the dynamic aspects of virtuosity. In what follows, we present our ns implementation. In Section 5, we present a number of experiments that investigate and validate virtuosity's design characteristics.

### 4.1 Virtual Network Emulator

The ns simulator includes a number of important packages supporting Internet protocol analysis, wireless links and other facilities. However, ns does not support the creation of virtual networks. We adopted several models (i.e., agents, applications etc.) defined by the ns architecture and augmented existing ns

components such as schedulers, links, TCP traffic sources to simulate virtuosity functions. We also implemented a number of new features to support virtual network emulation.

#### 4.1.1 Virtual Network APIs

In our model, the fundamental objects that emulate virtual networks over physical resources are the *virtual link* and *virtual link aggregate* objects. The virtual link object is an abstraction that relates to a partition of a virtual resource being used by a given virtual network. As an example, two or more virtual networks may use a common physical resource; they may, however, have different virtual links instantiated for their specific provisioning needs. A virtual link exists between two nodes in the parent network infrastructure. Therefore, a designer may specify capacity and price and attach nodes as endpoints. By definition, this occurs over a single virtual resource, but the node endpoints are not necessarily connected through a single physical link. Rather, they may be connected by one or more physical (or parent virtual) links.

As part of the virtual network emulator, we implemented the dynamic instantiation of virtual links, virtual link aggregates and traffic generation of flows over virtual links. Abstracting and emulating the creation of virtual links and virtual link aggregates represents an important component part of the virtuosity simulation system discussed in Section 5. While virtual network ‘call’ models (i.e., models that accurately capture the creation, ‘holding times’ and deletion of virtual networks) do not currently exist, we simulated virtual network call generation by leveraging well-known on/off source models for modeling traffic flows. The dynamic creation of virtual links is useful for simulating requests from customers to spawn virtual networks. For each provisioning request there may be one or more simulated instantiations of virtual links, (e.g., as new virtual links are being created over a physical link). The instantiation of a virtual link represents a virtual network request from a customer over a particular virtual network resource. This allows us to model and manage multiple virtual network resources over an existing set of virtual networks. The underlying ns system handles the routing of packets within each virtual network.

An example of the virtual link object API is shown in the following TCL<sup>1</sup> script:

```
set virtual_link [new VirtualLinkAgent/VLMixed]

# attachments for a virtual link
$virtual_link attach-src $node1
$virtual_link attach-sink $node2
# virtual link aggregate to which the virtual link
# belongs
$virtual_link attach-aggr $vlagg

$virtual_link set capacity_ 10Mb
$virtual_link set flowduration_ 25

# duration of the virtual link
$virtual_link set avgduration_ 2000
$virtual_link set exponential_ 1
$virtual_link set deterministic_ 0
```

A virtual-link aggregate abstraction is a set of virtual links sharing common provisioning characteristics. The virtual link aggregate abstracts the virtual resource at a given level of the resource inheritance tree (see Figure 2). It is also where the arrival process is defined for a given resource within a virtual network. For example, the environment may detail the exponential process for instantiation of virtual links and the relevant arrival parameters such as average arrival time. The designer will follow similar steps and

---

<sup>1</sup> Most of the scripts shown in this paper are TCL language scripts, however, they are simplified for clarity.

reasoning for a virtual link as he defines the ‘attachment’ to the provider network. An example of the API for the virtual link aggregate object is shown in the Appendix (see *vlaggregate.tcl*).

### 4.1.2 Virtual Network Provisioning

From a customer’s perspective, we use two distinct models for provisioning virtual networks. We do not, however, propose new virtual network provisioning models in this paper. Rather, we consider two models from the literature. In the first model, a *customer-pipe* [12] is set up for every virtual link requested from a customer (i.e., between any two source-destination endpoints within the customer’s virtual network). An alternative way of looking at the customer-pipe model is an emulation of a lease line or frame-relay connection. The advantage of this approach is that the customer has the flexibility to determine service level agreement (SLA) guarantees for each virtual link requested. From the provider’s perspective this also makes it simpler to manage virtual networks. We also consider the *hose* model [13] that simplifies resource management for customers. In this model, a virtual network customer specifies a set of endpoints to be connected with common endpoint-to-endpoint performance guarantees. In this case, the resource management parameters are given as aggregates, (i.e., based on aggregate incoming traffic and aggregate outgoing traffic observed at a hose interface). The hose interface is defined between a source and a set of endpoints.

The difference between the customer-pipe and hose models is that for a given set of endpoints, the customer-pipe model requires performance guarantee specifications for each pipe, thus, requiring the customer to have specific knowledge of the provider’s infrastructure. In a large and highly meshed virtual network, this results in lengthy and time consuming specifications. From the customer’s perspective, the customer-pipe model requires more administrative effort in comparison to the definition of a SLA based on aggregate information required by the hose model. The virtuosity simulation system implements both of these models as virtual link aggregates.

### 4.1.3 Constructing Virtual Networks

The virtual link and the virtual link aggregate objects are the core building blocks used to construct virtual networks. In what follows, we present a number of simple examples on how these building blocks can be used to create a variety of virtual networks.

**Constructing a Single Virtual Network.** It is sufficient to use a single virtual link object to compose a virtual network, as shown in the virtual network script, *singleVN.tcl* (see Appendix). As shown, we define a simple virtual network composed of three virtual links. The virtual links exist over a topology of physical resources. The virtual links do not depend on the physical topology, however.

**Constructing Multiple Virtual Networks.** When constructing a numbers of virtual networks using the same physical resource(s) (or parent virtual resources), the virtual link aggregate object is essential. While virtual link entities belong to customer or child virtual networks, the virtual link aggregate is an entity that is used by a parent or provider. The parent maestro coordinates resource management using virtual link aggregates. As shown in the Appendix, we demonstrate the creation of an aggregate object and three virtual links that are associated with different customers, (i.e., different virtual networks). As shown in the virtual network script, *muxVN.tcl* (see Appendix), individual virtual links point to a common virtual link aggregate. Once the virtuosity framework is instantiated, the maestro object is attached to the aggregate object to perform resource management. In this example, we have provisioned virtual links with capacities of 5 Mb/s, 2 Mb/s and 1 Mb/s from the partitioned resource. This is how we aggregate virtual links. In a constrained environment, the request for virtual links requires virtual network admittance (i.e., via virtual network admission control) prior to instantiation. The example also demonstrates how the building blocks can be used to compose virtual networks.

**Nesting Virtual Resources.** The *nestVN.tcl* script (see Appendix) illustrates an example virtual network that enables the creation of nested virtual resources. This is a simple declaration and does not require a node-to-node interconnection specification. Other parameters may be defined, however. In this case, virtual network admittance is automatically passed for the instantiation of a virtual nested resource. The command `create-nested-agg` creates a dependency link between two virtual link aggregates, one in the parent domain and the other in the nested aggregate.

**Creating Virtual Network Topologies.** Our last example (see Appendix, *topologyVN.tcl*) illustrates how virtual resources are independent of the underlying infrastructure. We use three different virtual links that are formed from physical resources. In this example, only a single virtual link object is used. In the case of multiple virtual networks, individual requests would need to be made for each requested virtual resource. In our simulation environment (discussed in the next section), dynamic requests are emulated through an application designed specifically for this purpose. This allows multiple virtual networks to make use of a common infrastructure of resources.

The virtual network script (*topologyVN.tcl*) demonstrates the flexibility of the virtual network emulator to instantiate virtual resources. We describe the creation of three different virtual resources over two distinct topologies. The first topology is composed of four nodes connected in a tree formation using three links. The second topology is composed of three links connecting three nodes in a triangular formation. The virtual resources emulate the same triangular form and do not depend on the physical topology. For the first topology each virtual resource is composed of two physical links. Thus, when virtuosity is instantiated only pointers to the virtual resources are needed, allowing virtuosity to be transparent to the lower levels. This demonstrates the autonomous control property of virtuosity.

## 4.2 Realizing the Virtuosity Components

The virtuosity components and their associated functions implemented as ns extensions are shown in Table I. The maestro supports virtual network admittance, orchestrates the virtual network resource management process through interfaces with the other virtuosity component objects and handles provisioning requests received as part of the resource renegotiation process. The maestro also utilizes measurement information obtained (via the delegate) from the arbitrator and its measurement cache to support the provisioning process, (e.g., by responding to policed threshold violations). The arbitrator implements three different measurement algorithms, as shown in Table I. The broker is implemented in a simple fashion, and while pricing information is not considered in our current implementation, we

<i>Components</i>	<i>Functions</i>
<i>Maestro</i>	<ul style="list-style-type: none"> <li>• <i>Virtual network admittance</i></li> <li>• <i>Customer-pipe and hose models</i></li> <li>• <i>Processing of monitoring information</i></li> <li>• <i>Resizing techniques for hoses</i></li> <li>• <i>Instantiated interfaces for renegotiation through the broker</i></li> </ul>
<i>Arbitrator &amp; Monitor</i>	<ul style="list-style-type: none"> <li>• <i>Measurement process</i></li> <li>• <i>Delivery of monitoring information to maestro</i></li> <li>• <i>Hose monitoring</i></li> </ul>
<i>Broker</i>	<ul style="list-style-type: none"> <li>• <i>Renegotiation process</i></li> <li>• <i>Renegotiation period is specified</i></li> </ul>
<i>Delegate</i>	<ul style="list-style-type: none"> <li>• <i>Proxy agent for delivery of information between virtuosity components and the maestro</i></li> </ul>

*Table I: Virtuosity Components and Functions*

consider pricing a determinant for the partitioning and allocation of resources. In what follows, we outline the implementation of each of the major components listed in Table I.

#### 4.2.1 The Maestro

The primary tasks of the maestro are to dynamically control the provisioning and partitioning of resources to meet the changing requirements of virtual network customers (subscribers). The maestro analyzes and makes resource allocation decisions based on stored state and policy information. In addition, the maestro translates and propagates specific policies to the other virtuosity components. The maestro simulation focuses on the provisioning, resource allocation and resource partitioning aspects of virtuosity framework.

In implementation, we utilized a virtual resource agent to deploy the customer-pipe model [12]. This is done in the form of a virtual link aggregate where a source-destination pair and performance parameters define a customer-pipe. In order to generate a virtual network, a customer requests virtual links over a given parent network. Those virtual links are then instantiated from the parent’s virtual link aggregate objects. All maestro management operations are performed over each virtual link. This includes monitoring, resizing, admission control and renegotiation. The maestro also supports resource hoses [13] allowing a customer to request a hose over a provider’s infrastructure. A customer may also set up virtual links over its own virtual network. Those links may overlay multiple virtual resources contained in the hose specification. In contrast, a hose may be formed over an existing virtual network infrastructure representing a nested hose with associated endpoints and corresponding SLA.

For capacity management, we considered the resizing techniques described in [15] and adopt the resized provider-pipe technique using the local maximum predictor. Based on this technique, the allocation of capacity is followed by capacity resizing using online measurements. Because the virtuosity framework is based on storing metering and measurement information, the resizing approach is appropriate. This technique requires that we first allocate capacity following a worst case demand and then resize over intervals in which prediction is made, following the maximum bandwidth measured over the interval. The basic idea is that a time window is partitioned into smaller periods where the maximum measurements from each of the partitioned intervals provide the basis for the prediction. A similar approach, called average window method, is also used in the metering portion of the arbitrator (see Section 4.2.3).

The maestro incorporates a virtual network admittance process based on measurement-based algorithms. The virtual network admittance algorithm is defined as the long-term policy control for admission of virtual networks. It should be noted that the admission process allows a virtual link aggregate to be instantiated over a given resource representing a new virtual network request over the resource. We adopted two algorithms from flow-based measurement-based admission control [16] and apply them to the virtual network environment; these include, the measured sum (MS) and the Hoeffding bound (HB) algorithms. As a proof-of-concept, we chose to implement these algorithms as a means to investigate the properties of the virtuosity framework and not necessarily to strive for optimal resource efficiency. As such, other algorithms can be implemented using the maestro. The definition of this algorithm can be specified through a simple TCL script using the ns virtuosity extension for virtual network resource management. The measured sum algorithm simply uses the measurement of the current traffic load to estimate the available capacity for new flows. The algorithm returns a positive response if the following test is successful [16]:

$$w + r^\alpha < v\mu,$$

where  $w$  is the measured load of existing traffic,  $v$  is the user-defined utilization target,  $r^\alpha$  is the new requested capacity. The parameter  $\mu$  is the total capacity defined for the virtual link aggregate. The parameter  $v$  is called the utilization target. The admission control algorithm maintains the utilization

below a certain level according to the utilization target. In our simulations, we selected  $v = 0.9$ . The simple rationale behind this algorithm applied to the admission control problem is to compute the capacities requested for all virtual links and to compare the sum with the capacity of the partitioned link. However, this procedure is quite inefficient if we consider statistical multiplexing. A more elaborate idea is to use the measured sum through sampling intervals computing the maximum during an average window, and then computing the available bandwidth. Based on a theoretical foundation and with consideration for statistical multiplexing, the *equivalent bandwidth* is proposed as a measurement metric for a set of flows. The equivalent bandwidth is defined as  $C(\epsilon)$  such that the stationary bandwidth requirement of the set of flows exceeds this value with probability  $\epsilon$  [16]. The formula for the measurement-based equivalent bandwidth ( $C_H$ ) based on Hoeffding bounds is:

$$C_H(v, \{p_i\}_{1 \leq i \leq n}, \epsilon) = v + \sqrt{\frac{\ln(1/\epsilon) \sum_{i=1}^n (p_i)^2}{2}},$$

where  $v$  is the measured average arrival rate of the existing traffic,  $n$  is the number of flows and  $p_i$ 's are the peak rates for each flow. We extend the concept for a set of virtual links and calculated the equivalent bandwidth for the traffic over a set of virtual links, (i.e., the virtual link aggregate). Thus the  $p_i$ 's become the virtual link throughput samples and  $v$  is the measured load over a virtual resource. For new requests to be admitted the procedure checks to determine if the requested bandwidth plus the equivalent bandwidth of the virtual resource exceeds the total capacity for the virtual link aggregate.

#### 4.2.2 The Broker

In [4] we proposed an auctioneer model as the core algorithm for the resource allocation process. In this paper we argue for a more general approach where customers issue resource requests and providers attempt to meet these requests using resource partitioning. The broker provides an interface between the customers and the provider through the maestro for resource renegotiation. The simulation environment allows the designer to define the frequency at which renegotiations take place. This is equivalent to the frequency at which customers are allowed to request renegotiations. This frequency is defined by the network architect using an API (i.e., TCL script) and is usually in the order of minutes or greater. The renegotiation period should be large enough to avoid instability, but small enough to allow for efficient resource partitioning.

The interface between the customers and the virtual network provider is achieved through the maestro when the customer requests a virtual network. When this occurs, the maestro signals the broker that the customer has a suitable element to interface with the broker for renegotiations. This element is called a "buyer" and represents the interface between customer requests and the virtuosity framework, specifically the broker. In order to simulate the behavior of a customer that requests renegotiation, each buyer has a timer that drives its renegotiation process. This occurs if the measured bandwidth within a specified average window exceeds the contracted bandwidth. The broker determines the best allocation according to the aggregation of user requests. The maestro leverages the admission control algorithm to facilitate a simple renegotiation process.

### 4.2.3 The Arbitrator

We have implemented a simple arbitrator that is embedded in the virtual link aggregate abstraction performing FCFS scheduling of packets over a set of virtual resource links and provisioned capacities. Other tasks for which the arbitrator is responsible include resource monitoring, policing and event notification.

We adopt a high-level measurement component within the arbitrator. The metering process is operated through a virtual network queue monitor (QM) object. While the queue monitor counts every packet and the number of bytes that pass through the arbitrator, there also exists a queue monitor for each instantiated virtual link. Virtual links are metered and virtual link aggregates are updated accordingly. For nested virtual networks, traffic measurements are available to parents. This process may continue down the chain of parents as necessary. Thus it is important to have optimal and consistent sampling periods between networks to avoid inconsistent state or stability issues.

Metering performed by a virtual network queue monitor object is conducted on a different level of abstract to that of simple IP traffic metering. The queue monitor must inspect and maintain state for each source-destination pair (virtual link). In our current implementation, the monitor function is embedded in the arbitrator. The maestro requests the measurement information through the virtual link aggregate. Management information is stored in a cache database that is constantly updated by the arbitrator. Measurement tasks include getting a maximum throughput over a sampling interval and calculating average utilization. Calculated parameters include the Hoeffding bound parameter for capacity admission. This state information may be propagated to the parent network in the case of nested virtual network.

While the maestro receives the monitoring data from each resource, virtual link and virtual link aggregate agents also provide this information to the maestro. We adopt different methods for measurement and bandwidth prediction. The first method computes the bandwidth consumed over each sampling period. This method is called “point sampling”. The second form of measurement performs the well-known “exponential averaging” method. A third method uses a time window measurement where the time window comprises a reasonable number of sampling periods (e.g., at least 10 sampling periods). The user can define the desired sampling granularity setting the appropriate weights for providers. By controlling these parameters, the designer can have direct influence on the appropriate measurement and preferred behavior.

## 4.3 Instantiating Virtuosity

When a maestro is first deployed by a provider it obtains information about the virtuosity algorithms plugged in to the framework to configure its operation. This information consists of the selected virtuosity algorithms and the necessary parameters to support virtual network resource management. It is assumed that the provider configures such algorithms for appropriate operation. The Genesis Kernel validate the kernel and architectural configuration [1]. As part of the spawning process renegotiation and virtual network admittance algorithms can be configured or installed.

A customer is responsible for managing its own network and operational policy. The maestro is accessible to the customer at each level in the virtual network inheritance tree to manage its traffic as well as the spawned, subscriber (child) networks. Such examples demonstrate the principle of autonomous customer control identified earlier. The TCL script, *virtuosity.tcl* in the Appendix, shows the steps to build the elements of the virtuosity framework and the relationships between them. The autonomous control principle provides the customer with the flexibility to define virtuosity during instantiation or normal operations. This is achieved through a simple specification. The following script illustrates three examples of defining or configuring virtual network admittance schemes:

- During instantiation:  
maestro HB
- At a pre-defined time:  
At time1 "maestro HB"
- Given a specific condition:  
If (efficiency <= acceptable) {  
"maestro HB"}

## 5 Simulation

One of the key ideas behind virtuosity is its programmability. The virtuosity plug-in affords network designers the flexibility to deploy alternative resource management schemes for spawned virtual networks. The maestro, arbitrator and broker can be augmented through a set of programmable interfaces. A virtual network provider may choose to deploy alternative algorithms for resource management for a number of reasons including diversity of provisioning demand profiles, or matching virtual network deployment models and timescales, etc. A key aspect of the spawning architecture discussed in [1] [18] is the nesting of spawned virtual networks. Virtuosity inherits this feature. However, the virtuosity architecture enforces separation and autonomy between nested resource management schemes. This approach has the benefit of supporting scalability and flexible provisioning. In this section, we explore these issues through experimentation using the virtuosity simulation environment. These experiments provide insights into virtuosity's support for virtual network (e.g., nested admission control, load aggregation, operational timescales and signaling) and autonomous resource provisioning using different resource management schemes.

### 5.1 Simulation Environment

To generate virtual network dynamics within our simulation environment, we created a virtual network application that periodically generates requests for new virtual links. An arrival process based on exponential or deterministic distribution, models the interval between virtual link requests. Figure 5 illustrates this simulated behavior. We defined a virtual network application as an entity that dynamically

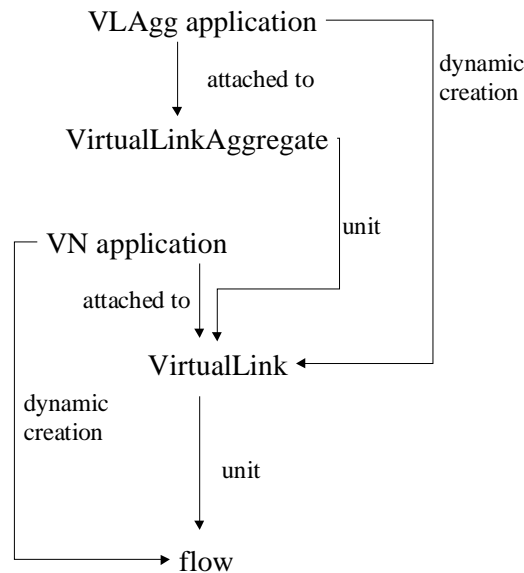
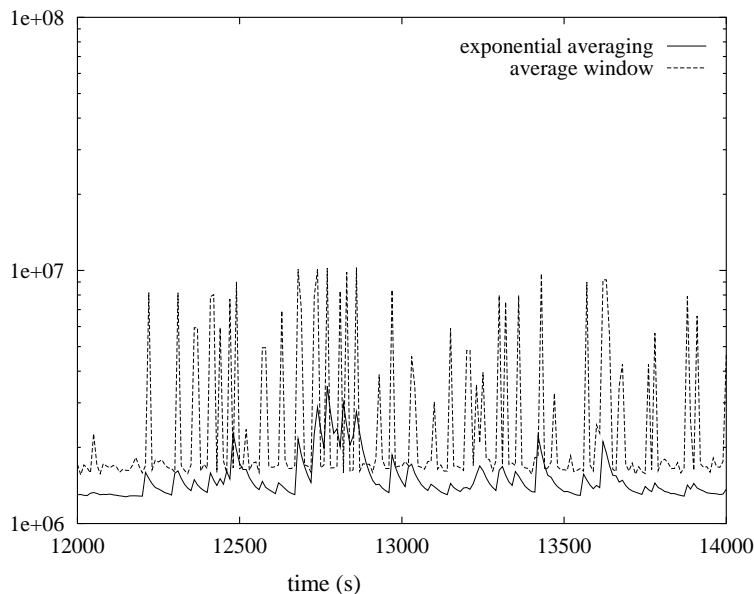


Figure 5: Components of the Dynamic Generation of Virtual Links and Traffic

creates virtual links given a virtual link aggregate. The interval between arrivals for virtual links represents another process that follows exponential, deterministic and Pareto distributions. The unit for traffic generation in a virtual link is a flow. For a given virtual network, flows correspond to regular traffic generated by network users. There is a second application that simulates flows within a single virtual link. The virtual network application creates traffic for the virtual link to which the application is attached. The flow has a duration time, occurs between two nodes and follows a process with specified parameters. The traffic generation source is modeled as an on/off source.

## 5.2 Measurement-based Admission Control

The Hoeffding bound mechanism is used to support virtual network admittance for a flat (i.e., non-nested) partitioned virtual network configuration. A virtual resource is defined with 2 Mb/s of bandwidth for this experiment. Figure 6 shows the consumed bandwidth against two different admission control metrics: *average time-window* and *exponential averaging*. The exponential averaging measurement is the metric considered by the Hoeffding bound algorithm. The customer (i.e., buyer) resource renegotiation interval is set to 900 seconds. Renegotiation only occurs when the bandwidth consumed on a given virtual link is overused. In this experiment, the simulation is operational for 18000 seconds. For clarity, we only show the window between 12000-14000 seconds in the plot. We observe the consumed bandwidth remains below the specified target capacity of 2 Mb/s most of the time, which represents the configured exponential averaging metric.



*Figure 6: Hoeffding Bound Mechanism being used for a Virtual Resource (bit/sec) whose Bandwidth is 2 Mb/s. All Time-Window Measurements, Regular Samples and Exponential Averaging Measurements are shown in the plot.*

## 5.2.1 Virtual Network Dynamics

In the next experiment, we support different levels of nesting in the virtual network inheritance tree. We considered five different nesting levels over the physical network. The virtual network script shown below illustrates how the five levels are defined and composed – simple cascading of the virtual link aggregates at each level of the inheritance tree.

```
set resource_(1) [new VirtualLinkAgent/VLAgg]
$resource_(1) attach-src $node_(0)
$resource_(1) attach-sink $node_(1)
$resource_(1) set capacity_ 10Mb
$resource_(1) attach-maestro $maestro_(1)

set nested_(1) [new VirtualLinkAgent/VLAgg]
$resource_(1) create-nested-agg $nested_(1)
$nested_(1) set capacity_ 5Mb
$nested_(1) attach-maestro $maestro_(2)

set nested_(2) [new VirtualLinkAgent/VLAgg]
$nested_(1) create-nested-agg $nested_(2)
$nested_(2) set capacity_ 3Mb
$nested_(2) attach-maestro $maestro_(3)

set nested_(3) [new VirtualLinkAgent/VLAgg]
$nested_(2) create-nested-agg $nested_(3)
$nested_(3) set capacity_ 2Mb
$nested_(3) attach-maestro $maestro_(4)

set nested_(4) [new VirtualLinkAgent/VLAgg]
$nested_(3) create-nested-agg $nested_(4)
$nested_(4) set capacity_ 1Mb
$nested_(4) attach-maestro $maestro_(5)
```

**Nested Admission Control.** Figure 7 shows the usage profile for the nested virtual network at level four of the virtual network inheritance tree using the nested configuration discussed above. Exponential averaging and average windowing algorithms are active in this case. At this level, we can observe the virtual network admittance operation.

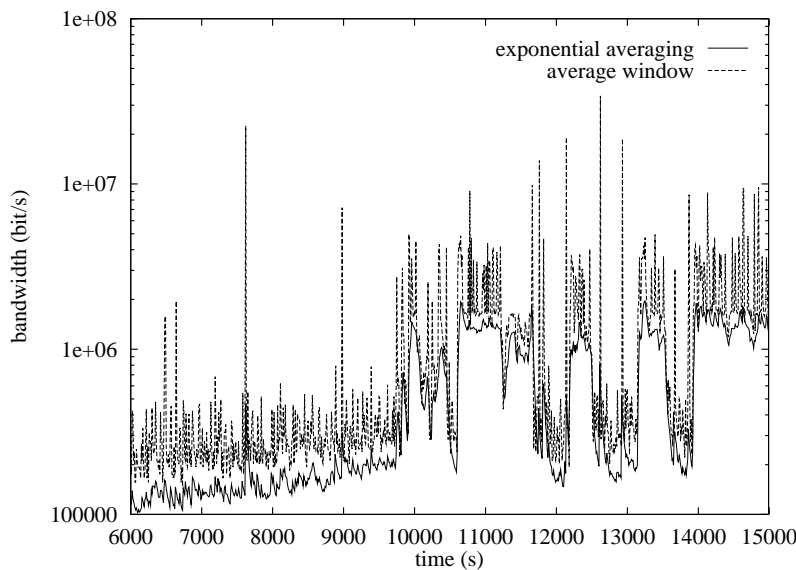


Figure 7: Level Four: Virtual Network Inheritance Tree

As the bandwidth reaches 2Mb/s, a break occurs as a new request is denied, forcing load reduction. Following this, arrivals for new virtual network requests are accepted and the usage once again increases.

The process cycles in this manner to reflect the expected (not necessarily optimal) load response behavior and the associated virtual network admission control sequences.

**Virtual Network Load Aggregation.** The load behavior of the nested configuration is illustrated in Figure 8. The traffic load is shown against time for each nested level in the inheritance tree. From the figure, we can observe that the nesting effect is stable and manageable. Within virtuosity, maestros are instantiated at each level of the inheritance tree for each virtual network. Thus, we can partition functions, such as signaling requirements across the network tree inheritance. This represents an advantage over flat, partitioned virtual network models. Table II shows the impact of signaling observed at different nesting levels in the virtual network nested hierarchy.

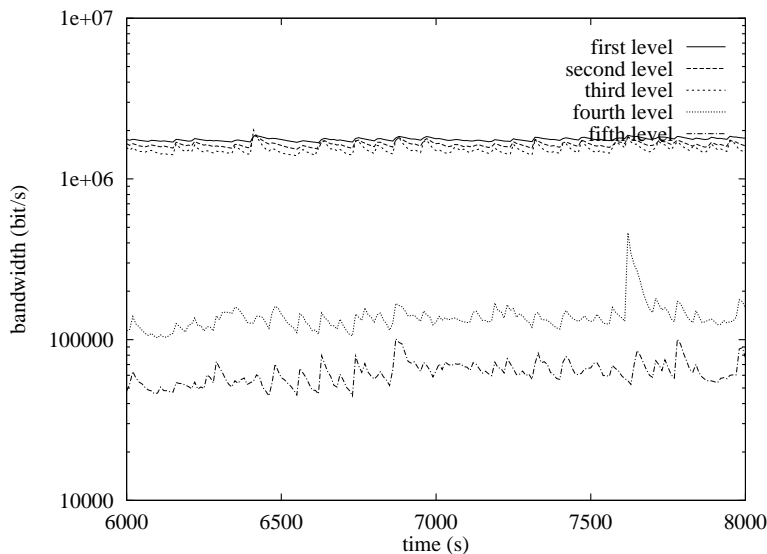


Figure 8: Nesting – Levels 1-5 of a Virtual Network Inheritance Tree

The nested load aggregation is shown in Figure 8 where the bandwidth at level four is aggregated with level five. It is also apparent that level one aggregates all other levels including its own traffic. This cascaded effect is shown Figure 8, where level one occupies the highest load (i.e., 10 Mb/s) encompassing the other levels. In each case, the exponential averaging metric is defined at 10 Mb/s, 5 Mb/s, 3 Mb/s, 2 Mb/s, and 1 Mb/s, respectively. Flow generation for each virtual network varied at each nesting level.

**Timescales and Signaling.** We observe that traffic changes are visible over larger timescales showing that the maestro can perform virtual network resource management tasks at these timescales. Therefore, virtual network admittance can operate in the order of minutes without affecting the performance stability of the system. Worst case conditions (i.e., high bandwidth demanding virtual networks spawned over smaller (msec/seconds) intervals) would certainly stress this environment, and thus, the maestro would also need to be capable of operating over finer timescales for optimal management.

Table II shows a sample of signaling messages recorded during the nested simulation test discussed above. The number of signaling messages at each level is summarized over a simulation run of 15000 seconds. The results show the impact of decentralized control over several levels of nesting. Because a child network manages its own traffic and the aggregate load from its child network, the parent maestro does not have to deal with signaling messages originating of nested levels beyond its immediate child networks. In this experiment, a single maestro would have approximately 1 message per 150-second interval. We observe that signaling is less a concern under nested condition, regardless of tree depth. On

the other hand, for bandwidth-constrained environments, we believe policing may occur more frequently with increased nesting. This is primarily due to the multi-level provisioning interactions and dynamics.

<i>Virtual Network Level</i>	<i>Signaling Messages</i>
1	17
2	14
3	27
4	16
5	21
1-5	95

Table II: Observed Signaling for Nested Resources

## 5.2.2 Autonomous Resource Provisioning

We performed several experiments using hose provisioning. As discussed earlier, a single hose comprises two resources. Figure 9 shows the resizing of the two resources used by a hose provisioning scheme. The declaration of the hose is shown in the virtual network script below. In this example, capacity is first

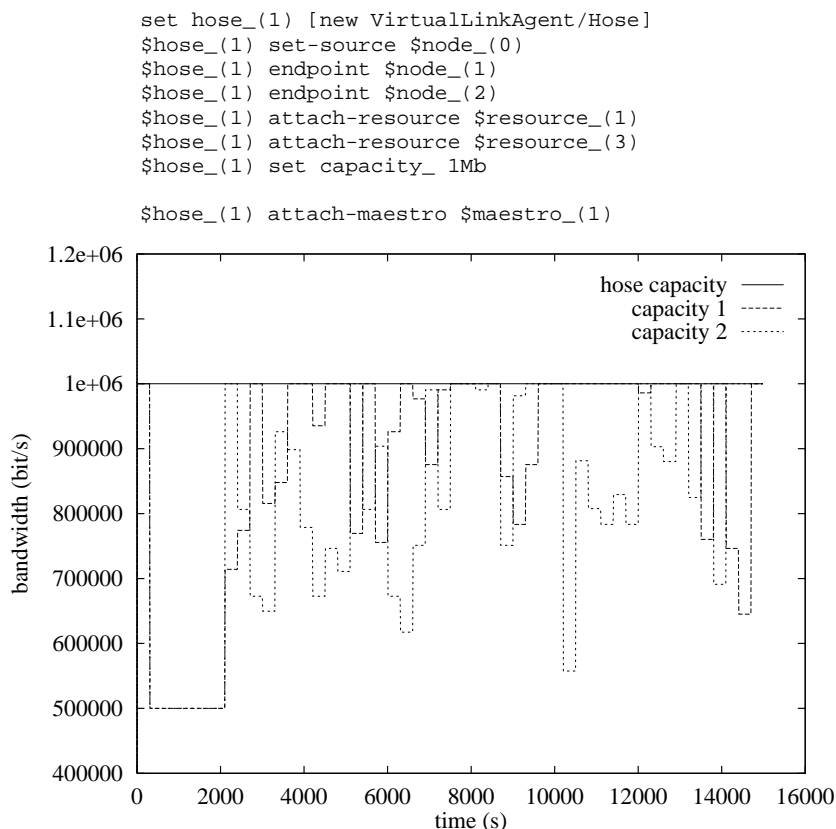


Figure 9: Resizing within a Single Hose

allocated at the maximum level following worst-case demands. During the operation, resources are resized using online measurements and estimates are computed using the average window mechanism. To demonstrate the provisioning flexibility of the virtuosity system, we built a nested hose virtual network from a customer-pipe model. We first defined the virtual network infrastructure built over a physical network composed by three nodes. The first level is based on the customer-pipe model and different virtual link aggregate objects. On top of this level we built the nested hose virtual network, demonstrating

provisioning autonomy. This allows customers to flexibility provision their virtual network using alternative schemes without dependency on the provider’s configuration. In addition, each of these two levels can have different resource management mechanisms to support specific optimization and provisioning needs.

Figures 10(a) and 10(b) shows the load observed for one resource at the hose level, respectively. Figure 10(a) shows the load for the virtual resource at the first level. This represents a portion of the hose traffic and its own user traffic. Figure 10(b) shows the total load in the hose virtual network including the loading of the resource used for Figure 10(a). We observe that the load in the hose virtual network is greater due to traffic aggregation.

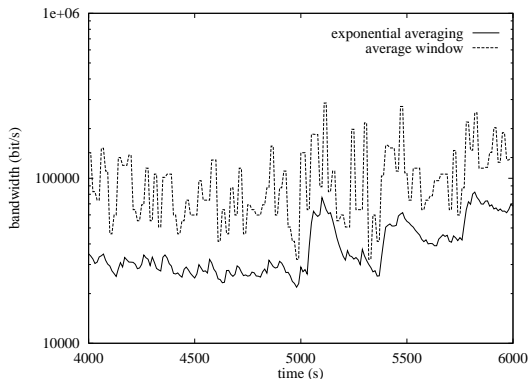
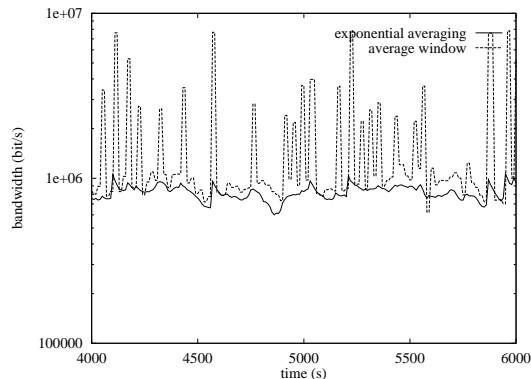


Figure 10: (a) Resource Load



10 (b) Hose Load

The script (*nestprovision.tcl*) used to create the nested scenario discussed above is given in the Appendix.

## 6 Open Issues

In the next several years we will likely see considerable activity in the area of virtual networks, particularly in the area of dynamic service and resource provisioning, network management, and network personalization and automation. The implementation discussed in this paper provides a proof of concept for virtuosity. In particular, we were interested in studying programmable resource management of spawned virtual networks. A number of issues remain open, however. First, there is a need develop an flexible and efficient signaling protocol capable of managing state information for each virtuosity element databases. A similar requirement for such a signaling protocol is also recognized by the Darwin Project [17].

Virtual network monitoring is a costly activity. Issues to be considered include new models to perform traffic measurement with consideration of per packet monitoring and timescales. Another issue for study would be mechanisms to efficiently measure the traffic within virtual networks. One possible approach could be to have ‘blank intervals’ and use prediction over the blank intervals. An interesting issue to address is the duration of these intervals. If the interval is too large then this may result in less accurate measurements. In contrast, if the interval is too small then this could lead to higher measurement cost. We plan to use the virtuosity simulation environment to investigate measurement algorithms in more detail.

For virtual network admittance, we leverage existing admission control algorithms built on flow-level QOS semantics and demand profiles. In this paper, we did not set out to find the best admission control algorithm. Rather, we demonstrated that different algorithms may coexist and that programmable interfaces can provide customized resource management for spawned virtual networks. However, finding

optimal virtual network admittance algorithms and better virtual network models is an important area of research.

Finally, virtual network capacity scheduling is an interesting approach. With such a model, we may be able to simplify network provisioning by leveraging ideas of slow-time scale management and capacity aggregation. These concepts help minimize complexity and allow us to consider the scheduling problem in a different manner. The introduction of an efficient scheduling model to implement the constant, controlled and best-effort capacity classes is an area that we are investigating.

## **7 Conclusion**

In this paper, we have described the virtuosity architecture and focused on the design implementation, simulation and evaluation of virtuosity using the ns simulator. We have implemented multiple virtuosity components and presented an emulation model for spawning virtual network based on virtual link, virtual link aggregates, pipes and hose abstractions. We have investigated programmable resource management, virtual network admission, nesting dynamics and autonomous virtual network provisioning. Under the Genesis model of spawning networks, we have emulated the virtual network creation process and observed the operational interactions of nested virtual networks. We have discussed a number of virtual network resource management issues that need to be further considered. We have implemented the Genesis Kernel v1.0 in an experimental spawning network testbed [18] at Columbia University. This version of the kernel does include virtuosity, however. This is part of our on-going work.

## **8 Acknowledgements**

This work is supported in part by the National Science Foundation (NSF) under CAREER Award ANI-9876299 and with the support from COMET Group industrial sponsors. In particular, we would like to thank the Intel Corporation, Hitachi Limited and Nortel Networks 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. 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 paper. Finally, we would like to thank Michael E. Kounavis for his comments on this paper.

## 9 References

- [1] Campbell, A.T., Kounavis, M.E., Villela, D., Vicente, J., De Meer, H., Miki, K., and K. S. Kalaichelvan, "Spawning Networks", *IEEE Network*, Vol. 13 No. 4 pg. 16-30, July/August 1999.
- [2] Campbell, A.T., Kounavis, M.E., Vicente, J., Villela, Miki, K. and H. De Meer, "A Survey of Programmable Network", *ACM SIGCOMM Computer Communication Review*, Vol. 29 No. 2 pg. 7-24, April 1999.
- [3] 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.
- [4] Vicente, J., Campbell, A. T., Villela, D., "Virtuosity: Performing Virtual Network Resource Management", *Proceedings of the 7th IEEE/IFIP International Workshop on Quality of Service (IWQOS'99)*, pg. 65-76, London, June 1999.
- [5] UCB/LBNL/VINT Network Simulator - ns (version 2) <http://mash.cs.berkeley.edu/ns/ns.html>
- [6] Campbell, A. T., Kim, S., Gomez, J., Wan, C-Y., Turanyip, Z., and A. Valko, "Cellular IP", *Internet Draft*, draft-ietf-mobileip-cellularip-00.txt, IETF Mobile IP Working Group Document, Work in Progress, December 1999.
- [7] Blake, S., et al. "A Framework for Differentiated Services", *Internet Draft*, draft-ietf-diffserv-framework-01.txt, Work in Progress.
- [8] Rajan, R., Martin, J. C., Kamat, S., See, M., Chaudhury, R., Verma, D., Powers, G., Yavatkar, R., "Schema for Differentiated Services and Integrated Services in Networks", *Internet Draft*, draft-rajana-policy-qoschema-00.txt, Work in Progress, October 1998.
- [9] Keshav, S., and Sharma, R., "Achieving Quality of Service through Network Performance Management", *Proceedings of the 8th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'98)*, Cambridge, July 1998.
- [10] Campbell, A. T., De Meer, H. G., Kounavis, M. E., Miki, K., Vicente, J., Villela, D. A., "The Genesis Kernel: A Virtual Network Operating System for Spawning Network Architectures", *Proceedings of the IEEE 2nd International Conference on Open Architectures and Network Programmability (OPENARCH'99)*, pp. 115-127, October 1998.
- [11] Semret, N., and Lazar, A. A., "Design, Analysis and Simulation of the Progressive Second Price Auction for Network Bandwidth Sharing", *CTR Technical Report*, CU/CTR/TR 487-98-21, April 1998.
- [12] Lazar, A.A., "A Real-Time Control, Management and Information Transport Architecture for Broadband Networks", *Proceedings of the 1992 International Zurich Seminar on Digital Communications*, pp. 281-296, March 16-19, 1992.
- [13] Duffield, N.G., P. Goyal, A.G. Greenberg, P.P. Mishra, K.K. Ramakrishnan, Jacobus E. van der Merwe, "A Flexible Model for Resource Management in Virtual Private Networks", *Proceedings of the ACM SIGCOMM'99, Computer Communication Review*, Vol 29, No 4, October 1999, pp. 95-108.
- [14] The Genesis Project: Spawning Networks <http://comet.columbia.edu/genesis>, 1998.
- [15] Duffield, N.G., P. Goyal, A.G. Greenberg, P.P. Mishra, K.K. Ramakrishnan, Jacobus E. van der Merwe, "A Performance Oriented Service Interface for Virtual Private Networks", *Internet Draft*, draft-duffield-vpn-QOS-framework-00.txt, Work in Progress.
- [16] Jamin, S. Shenker, S. J., and P. Danzig, "Comparison of Measurement-based Admission Control Algorithms for Controlled-Load Service", *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, Kobe, Japan, April 1997.

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

[18] Kounavis, M.E., Campbell, A.T., Chou Stephen, Modoux Fabien, Vicente, J and Hao Zhuang, "The Genesis Kernel: A Programming System for Spawning Network Architectures", *IEEE Journal on Selected Areas in Communications*, Special Issue on Active and Programmable Networks, to be published, 2001.

## 10 Appendix

<b>vlaggregate.tcl</b>	<pre> set virtual_resource [new VirtualLinkAgent/VLAgg]  \$virtual_resource attach-src \$node_(0) \$virtual_resource attach-sink \$node_(1)  # variables of bandwidth allocation and pricing \$virtual_resource set capacity_ 10Mb  # variables for measurement procedures \$virtual_resource set window_ 0.02 \$virtual_resource set sampleinterval_ 1.0  # composition of flows in virtual links \$virtual_resource set ppareto_ 0.0 \$virtual_resource set pexp_ 1.0 \$virtual_resource set pcbr_ 0.0  # average generation time for flows \$virtual_resource set avggentime_ 0.5 # whether flows have exponential duration \$virtual_resource set flowdurationexp_ 1 # average duration time of flows (seconds) \$virtual_resource set flowavgduriantiontime_ 25.0  #whether virtual links have deterministic duration \$virtual_resource set vldurationdet_ 1 #average duration time for virtual links (seconds) \$virtual_resource set vlavgduriantiontime_ 18000.0 </pre>
<b>singleVN.tcl</b>	<pre> set virtual_link1 [new VirtualLinkAgent/VLMixed] # attachment for a virtual link \$virtual_link1 attach-src \$node1 \$virtual_link1 attach-sink \$node2 \$virtual_link1 set capacity_ 5Mb  set virtual_link2 [new VirtualLinkAgent/VLMixed] # attachment for a virtual link \$virtual_link2 attach-src \$node2 \$virtual_link2 attach-sink \$node3 \$virtual_link2 set capacity_ 2Mb  set virtual_link3 [new VirtualLinkAgent/VLMixed] # attachment for a virtual link \$virtual_link3 attach-src \$node2 \$virtual_link3 attach-sink \$node3 \$virtual_link3 set capacity_ 1Mb </pre>
<b>nestVN.tcl</b>	<pre> Set virtual_resource [new VirtualLinkAgent/VLAgg] \$virtual_resource attach-src \$node1 \$virtual_resource attach-sink \$node2 \$virtual_resource set capacity_ 10Mb  Set nested_(1) [new VirtualLinkAgent/VLAgg] \$nested_(1) set capacity_ 2Mb  \$virtual_resource create-nested-agg \$nested_(1) </pre>
<b>muxVN.tcl</b>	<pre> set virtual_resource [new VirtualLinkAgent/VLAgg] \$virtualnetwork_link1 attach-src \$node1 \$virtualnetwork_link1 attach-sink \$node2 \$virtualnetwork_link1 set capacity_ 10Mb  set virtualnetwork1_link [new VirtualLinkAgent/VLMixed] # attachments for a virtual link \$virtualnetwork_link1 attach-src \$node1 \$virtualnetwork_link1 attach-sink \$node2 \$virtualnetwork_link1 set capacity_ 5Mb  set virtualnetwork2_link [new VirtualLinkAgent/VLMixed] </pre>

	<pre> # attachments for a virtual link \$virtualnetwork_link2 attach-src \$node1 \$virtualnetwork_link2 attach-sink \$node2 \$virtualnetwork_link2 set capacity_ 2Mb  set virtualnetwork3_link [new VirtualLinkAgent/VLMixed] # attachments for a virtual link \$virtualnetwork_link3 attach-src \$node1 \$virtualnetwork_link3 attach-sink \$node2 \$virtualnetwork_link3 set capacity_ 1Mb  # all attached to the same aggregate object \$virtualnetwork_link1 attach-aggr \$virtual_resource \$virtualnetwork_link1 attach-aggr \$virtual_resource \$virtualnetwork_link1 attach-aggr \$virtual_resource </pre>
<pre> topologyVN.tcl </pre>	<pre> #tree topology if (\$topology == "tree") { \$sim duplex-link \$node_(0) \$node_(3) \$param_(bw) \$param_(dly) DropTail \$sim duplex-link \$node_(3) \$node_(1) \$param_(bw) \$param_(dly) DropTail \$sim duplex-link \$node_(3) \$node_(2) \$param_(bw) \$param_(dly) DropTail } else if (\$topology == "triangle") { # triangle topology \$sim duplex-link \$node_(0) \$node_(1) \$param_(bw) \$param_(dly) DropTail \$sim duplex-link \$node_(1) \$node_(2) \$param_(bw) \$param_(dly) DropTail \$sim duplex-link \$node_(2) \$node_(0) \$param_(bw) \$param_(dly) DropTail } # virtual resources do not depend on the physical topology set virtual_resource_(1) [new VirtualLinkAgent/VLAgg] \$virtual_resource_(1) attach-src \$node_(0) \$virtual_resource_(1) attach-sink \$node_(1)  set virtual_resource_(2) [new VirtualLinkAgent/VLAgg] \$virtual_resource_(2) attach-src \$node_(1) \$virtual_resource_(2) attach-sink \$node_(2)  set virtual_resource_(3) [new VirtualLinkAgent/VLAgg] \$virtual_resource_(3) attach-src \$node_(2) \$virtual_resource_(3) attach-sink \$node_(0) </pre>

<b>virtuosity.tcl</b>	<pre> Set maestro [new VirtualLinkAgent/Maestro] #assuming resources and hoses as previously defined \$resource_(1) attach-maestro \$maestro \$resource_(2) attach-maestro \$maestro \$resource_(3) attach-maestro \$maestro \$maestro HB # instantiate broker 1 Set brok_(1) [new VirtualLinkAgent/Broker] \$brok_(1) attach-res \$resource_(1) # \$vpapp attach-brok \$brok # \$brok attach-maestro \$maestro # instantiate a delegate Set delegate_(1) [new VirtualLinkAgent/Delegate] \$delegate_(1) attach-res \$resource_(1) \$maestro attach-delegate \$delegate_(1) # \$delegate attach-maestro \$maestro # \$delegate_(1) attach-res \$resource_(1) \$brok_(1) attach-delegate \$delegate_(1) #representing the customers Set vpapp_(1) [new Application/VNGenerator] # \$maestro attach-app \$vpapp \$vpapp_(1) attach-brok \$brok_(1) \$vpapp_(1) attach-maestro \$maestro \$delegate_(1) attach-app \$vpapp_(1) \$vpapp_(1) attach-res \$resource_(1) # generation parameters \$vpapp_(1) set avgenerationtime_ 900.0 \$vpapp_(1) renegotiation \$vpapp_(1) exp \$vpapp_(1) set single_capacity_ 1Mb </pre>
<b>nestprovision.tcl</b>	<pre> set nested_(1) [new VirtualLinkAgent/VLAgg] \$resource_(1) create-nested-agg \$nested_(1) \$nested_(1) set capacity_ 5Mb  set nested_(2) [new VirtualLinkAgent/VLAgg] \$resource_(3) create-nested-agg \$nested_(2) \$nested_(1) set capacity_ 1Mb  \$resource_(1) attach-maestro \$maestro_(1) \$resource_(3) attach-maestro \$maestro_(1)  set hose_(1) [new VirtualLinkAgent/Hose] \$hose_(1) set-source \$node_(0) \$hose_(1) endpoint \$node_(1) \$hose_(1) endpoint \$node_(2) \$hose_(1) attach-resource \$nested_(1) \$hose_(1) attach-resource \$nested_(2)  \$nested_(2) attach-maestro \$maestro_(2) \$nested_(1) attach-maestro \$maestro_(2) \$hose_(1) attach-maestro \$maestro_(2) </pre>



**Daniel Villela** (dvillela@comet.columbia.edu) received the degree in Electrical Engineering in 1997 and the 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 the 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 IEEE, IEEE Communications Society and the Brazilian Computer Society (SBC).



**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 and QOS research. He is currently the co-chair for the IEEE Conference on Open Architectures and Network Programming (OPENARCH 01). Andrew T. Campbell received his Ph.D. in Computer Science in 1996 and the NSF CAREER Award for his research in programmable mobile networking in 1999.



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