Managing Spawned Virtual Networks

Andrew T. Campbell[1], John Vicente[2] , and Daniel A. Villela[1]

[1] Center for Telecommunications Research, Columbia University
[2] Intel Corporation

Abstract. The creation, deployment and management of network architecture is manual, ad hoc and slow to evolve to meet new service requirements resulting in costly and inflexible deployment cycles. In the Genesis Project (genesis@comet.columbia.edu), Columbia University, we envision a different paradigm where new network architectures are dynamically created and deployed in an automated fashion based on the notion of "spawning networks", a new class of open programmable networks. Spawning networks support a virtual network operating system called the Genesis Kernel that is capable of profiling, spawning, architecting and managing distinct virtual network architectures on-the-fly. In this paper, we describe a kernel plug-in module called "virtuosity" for the management of multiple spawned virtual networks. Virtuosity exerts control and manages multiple spawned virtual network architectures by dynamically influencing the behavior of a set of resource controllers operating over management-level timescales.

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 [8] a different paradigm where a communication middleware platform is capable of profiling, spawning, architecting and managing distinct virtual network architecture on-the-fly. We call our vision Genesis and summarize here the Genesis Kernel, a virtual network operating system.

We believe that the design, creation and deployment of new network architectures should be automated and built on a foundation of *spawning networks*, a new class of open programmable networks. Spawning networks represent a new approach to the field of programmable networking where the network environment is capable of dynamically creating new network architectures on-the-fly. The Genesis virtual network kernel represents a next-generation approach to the development of programmable networks building on our earlier work on open programmable

[1] Daniel Villela is a CNPq-Brazil Scholar

[2] John Vicente is a Visiting Researcher at Columbia University

broadband [15][16][7] and mobile networks [25]. The Genesis Kernel has the capability to spawn child network architectures that can support alternative architectures 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". 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.

At the lowest level of the Genesis Kernel architecture [8], a *transport environment* 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 forms a virtual network topology. Each virtual network kernel can create a distinct *programming environment* that supports routelet programming and enables the interaction between distributed objects that characterize the spawned network architecture. The programming environment comprises a metabus[3] that partitions the distributed object space supporting communications between objects associated with the same spawned virtual network. Each virtual network has its own metabus. A *binding interface base* [1] supports a set of open programmable interfaces on top of the metabus, which provide open access to a set of distributed routelets and virtual links that constitute a virtual network architecture. The metabus and binding interface base also support a set of *life cycle services*, enabling the *profiling, spawning and management* of child virtual networks. For full details on the Genesis Kernel see [8].

Within Genesis, resource management of spawned virtual networks is handled by *virtuosity* [9], a Genesis Kernel plug-in. The virtuosity architectural model (see [9] for complete architectural details) comprises a number of distributed elements. These elements are instantiated as part of the child virtual network kernel during the spawning phase [8] and are deployed as a set of distributed plug-in objects. Virtuosity 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 slow time-scale *dynamic provisioning*, *capacity classes*, which provide general purpose 'resource pipes', *inheritance* and *autonomous virtual network control.*

In this paper, we present the elements of the virtuosity system, a next-generation architecture for virtual network resource management. In Section 2, we present the *maestro*, a central controller responsible for managing the global resource policy within the virtual network. In Section 3, we introduce the *auctioneer*, which implements an economic auctioning model for resource allocation. An *arbitrator*, presented in Section 4, represents an abstract virtual network capacity 'scheduler'. We summarize and conclude in Section 5.

---

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

2 Maestro - Distributed Virtual Network Control

At the core of the virtuosity resource management architecture is the maestro, a key controller that oversees the resources[4] (i.e., virtual links that interconnect routelets) of the managed virtual network domain. Virtuosity, through maestro, manages and controls virtual network resources on a slow performance management [13] timescale that operates on the minutes / tens of minutes period. We argue that this is a suitable timescale for virtuosity to operate, while allowing virtual networks to perform dynamic provisioning, as needed. Maestro coordinates virtual network control through distributed virtuosity components performing virtual network monitoring, economic-based resource allocation and capacity-based scheduling all of which operate or exert control on management-level timescales.

Using fundamentals of distributed management design the maestro manages global resource policy within a virtual network and its (parent) allocated virtual network resources. In a fully distributed manner, the maestro maintains global state of its virtual network. Maestro uses dynamic provisioning of virtual network resource capacity to meet the changing needs of its child networks (captured in child per-virtual network policy) and to react to changes in its global state. That is, a maestro may need to respond to dynamic changes in its own virtual network (e.g. changes in the resource needs of its local clients/users) and child network resource needs, as well as adjusting to changes imposed on it by the underlying parent network resource availability. In addition, maestro coordinates and influences child network behavior through the integration of monitoring-based feedback and economic factors which are being driven by subscriber service demands and cost potential. Maestro establishes resource policies, coordinates policy distribution and enforces policy through capacity scheduling and policing. A delegate, acting as a proxy agent, serves maestro by promoting decentralized coordination and localized communications. Delegates handle all local resource interactions and control mechanisms on the virtual network domain-specific routelets by interfacing with the other virtuosity elements supporting resource allocation and virtual network scheduling.

Maestro interacts with its child networks to promote the efficient use of its global resources while ensuring that the resource needs of its child networks and its own virtual network users are being met. The maestro can influence the way in which resources are allocated to its child networks by setting optimal market pricing [20] and resource allocation strategies, e.g., under provisioning its own virtual link resources but overbooking resources to child networks to maximize revenue for the controlled capacity traffic.

---

[4] Although we restrict the virtual network resource to link bandwidths, we feel that the virtuosity model can be easily extended to support other router resources or by partitioning router resources proportionally based on virtual network link aggregate demands.

2.1 Maestro Design

During the spawning phase of a child network the maestro conducts a virtual network admission control test based on the resources requested by the child network topology. If the test is positive, then the parent provider network admits the child network and allows it to become a participant in the auctioning process controlled by the auctioneer and governed by the child virtual network's policy. Admission is coordinated by the parent maestro using its virtual network hierarchy tree. The parent maestro receives a *ReqSpec* for admission from a child virtual network and determines if sufficient resources are available within the context of its own available network resources 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. Admission is based on evaluating the *ReqSpec* target capacity class resource provisions (viz. rate_quantity) against aggregate capacity class policies and aggregate resource usage. By monitoring the available capacity along all of its virtual links, the maestro determines if resources allocated along its virtual links are also underutilized. Based on this measurement state and capacity threshold violations it can allocate under utilized resources based on the capacity class, bandwidth and policy requested by the new virtual network. If capacity is available, the child network is immediately

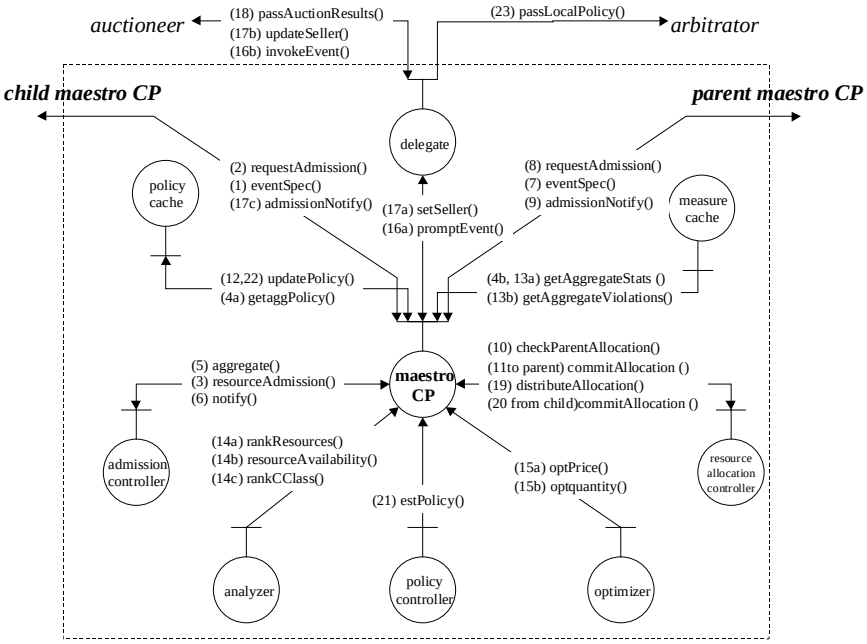

Fig. 1. Maestro Object Model

admitted and the child network is allowed to participate in the auction process. In the case that the parent has insufficient resources to accommodate the new child network then it needs to renegotiate its provisioning needs with its own parent (and hence its

provider) at the next level down its virtual network inheritance tree [8], [9].The provisioning request enters the parent auctioning process following a successful admission control sequence by traversing the hierarchy tree through several levels until a provider is found that can accommodate the requested demands.

Through slow timescale resource allocation, the maestro invokes the auctioning process on a periodic or static deadline basis. This period is driven (again) by slow timescale consideration allowing the auctioning process to reach equilibrium and maintain constant services over longer timescales, e.g., tens of minutes. Resource pricing and quantity announcements to child networks are set such that the parent can achieve more effective utilization and revenue gain. The maestro uses two variables for resource auctioning. These are a price quote, $\boldsymbol{Q}_{ij}$, and a rate quote, $\boldsymbol{R}_{ij}$, (where i = virtual resource; j = capacity class) which the *delegate* element relays to the *seller object* of the resource allocation system for appropriate auctioning. The auctioning process (which is discussed in the next section) requires a recursive, distributed algorithm and global consensus in order to reach steady state [20]; this constrains the static or dynamic invocation process to a lower periodic bound for recurring resource allocations. Upon reaching auctioning equilibrium [20], the maestro receives the results of the auctioning process, calculates local resource policies and stores the resource allocation policy results for child networks in the policy cache.

Also depicted in Figure 1, we now illustrate the behavior of the admission control and the resource allocation process from the perspective of the maestro system and its objects (viz., *maestro control point (CP), resource allocation controller, admission controller, optimizer, analyzer, policy controller, policy cache, measurement cache, and delegate*) embedded between the maestro CP's of the child and parent virtuosity systems. The process begins with the child maestro CP submitting an *eventSpec* (1) notifying the maestro CP that the child is requesting admission or an extended resource capacity request. This is immediately followed by a r*equestAdmission()* (2) of the associated provisioning specification. The maestro CP then invokes multiple *resourceAdmissions()* (3) methods from the admission controller for requested capacities on parent resources. The admission controller responds, *notify() (6),* after testing admission per resource against existing aggregate (4a) provisioning policies (i.e., child networks and local provisioning policy) and aggregate (4b) resource measurements (i.e., resource availability) to determine if the child requested increase or admission specifications exceeds the composite (5) provisioned policies and resource availability. The admission test is based on a rule-based policy that is per capacity class and per class resource usage in the determination of available capacity. The result (in this case) is admission failure along with the failure code and failure specification structure, specified per resources in list form. In turn, the maestro CP must then send an *eventSpec (7)* and *requestAdmission (8)* to the parent maestro CP to request additional capacity resources to extend its currently allocated provision, on behalf of the child's request. In this case, admission is successful upon completion of the parent's admission control, and the resource allocation (auctioning) procedures follows with notification through the parent maestro CP (9) with the admissible and allocated provisioning specification.

Prior to commitment (11) to the parent allocation, the allocation is checked (10) by a local resource allocation controller and stored (12) in the policy cache, if acceptable. The analyzer object is then invoked to assess the balance of global resource

consumption (13a,b,c), across the virtual network resources (14a,b) and the capacity classes (14c). The analyzer results are used by the optimizer object to establish the optimal price (15a) and quantity (15b) values per capacity class per managed resource for appropriate auctioning. The auctioneer is event notified (16a,b) for next provisioning interval synchronization and relayed with the optimal per class per resource (17a, b) auctioning variables. At this point, the child maestro CP is then notified (17c) of successful admission and prepares itself for auctioning with the local auctioneer. It is anticipated that reaching auctioning equilibrium will occur on the order of several minutes or longer as the number of competing child subscribers, parent resources and capacity classes increase. Nevertheless, we argue that the extended auctioning period is well in line with the necessity to maintain network stability through management timescale control, furthermore, we believe that this trade-off is offset with the resource efficiency gains that are achievable with slow timescale dynamic provisioning. Upon reaching auctioning equilibrium, auctioning results (18) are submitted (through the maestro CP) to the resource allocation controller, which then proceeds to distribute (19) resource allocations to the child network auctioning 'buyers'. It coordinates with the child's resource allocation controller to gain final commitment (20) on the allocation. If unsuccessful, the resource allocation process may re-cycle through these same steps, until reaching firm commitments by all child network subscribers and their requested resources capacities. If successful, the policy controller object establishes local policies (21), and the maestro CP updates (22) the policy cache with child network provisions and local resource policies. Finally, the delegate object passes (23) the required capacity scheduling and policing policies to manage and enforce the child allocations.

3 An Auction-based Resource Allocation System

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 (parent virtual network) and subscriber (child virtual network) behaviors, and correspondingly their 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 through effective price-based, load balancing and the addition of multiple virtual network subscribers. 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.

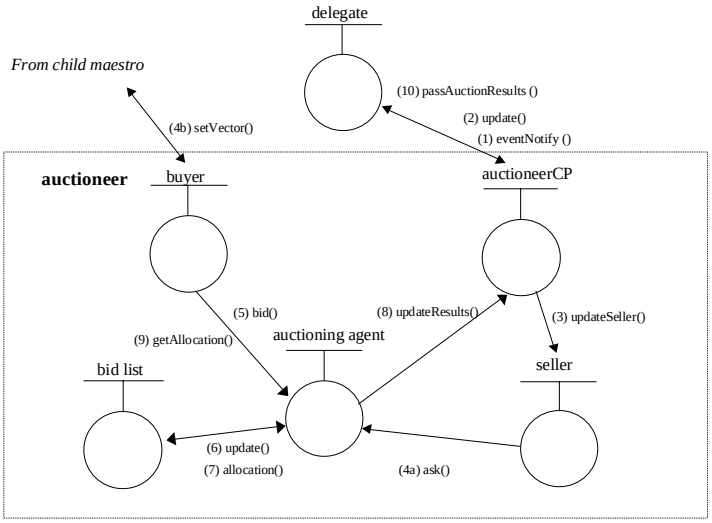

Fig. 2. Auctioneer Object Model

In this paper we consider a strategy known as Progressive Second Price (PSP) [20] that aims to provide high resource efficiency (e.g., cost, utilization) via a competitive, market-driven auctioning process. Auctioning occurs between a set of buyers (child virtual networks) and a seller (parent virtual network). Within a competitive bidding process a successful allocation conclusion for a particular buyer may not be attainable if the buyer is unwilling to pay the market value for the resource capacity or does not offer provisioning alternatives. The auctioning process is designed to follow a bidding procedure, allowing, for example the auctioning of best-effort classes would generally follow the more stringent capacity classes.

In our extended auctioning model we introduce two key contract variables: *contract_duration* and *contract_maxcost*. These variables represent important provisioning options which allow child subscribers to make long-standing contracts with the parent provider; in this sense, these variables represent a way for a child to avoid the normal open-market competition of the auctioneer.

3.1 Auctioneer Design

The auctioneer object architecture is illustrated in Figure 2. It comprises several objects: *auctioning agent*, *seller*, *buyer*, *bid list*, and *auctioneerCP* (auctioneer control point). The auctioneer is present at each routelet auctioning resources to a number of child network subscribers for its virtual link resources. The auctioneerCP object acts as a proxy that exchanges necessary information with the maestro (through the delegate) to receive updated parameters for interval-based auctioning. The seller object represents the provider in the auctioning system. Its task is to specify the quantity of the resource that is available for auctioning and the price for the resource. The seller will announce what is the current market price and availability for individual resources and capacity classes via the `ask()` method. These variables are

optimally determined based on what drives the provider market towards the desired revenue objective as well as resource gain efficiency, e.g., preferring controlled capacity to constant capacity. On the other hand, the buyer object plays the subscriber or child network role. Several buyers are allowed to be present within an auctioneer, bidding on behalf of the virtual network they represent. Buyer instantiation is a result of admission control; created and enabled when the buyer object participates in the resource allocation process. Once the buyer enters the auctioning process, it requests of the auctioning agent a position in the bid list that contains updated information (bids, allocation, quantities) about each buyer participating in the ongoing process. The buyer can bid for a resource by specifying its required quantity and the price expected to pay (upper bound) for individual capacity classes. The buyer object seeks to minimize cost and maximize rate quantity for each capacity class. The auctioning agent maintains the bid list object and stores updated information (current state) about the auctioning process. By accessing a bid list, the buyer can retrieve its current allocation and find whether it has been granted better conditions throughout the auctioning process.

When equilibrium has been reached, the auctioning agent updates the results from the process to the auctioneerCP for forwarding to the maestro. By definition, equilibrium is achieved when all buyers cannot improve their allocations. This can be implemented using a timeout condition to signal that no more changes will occur. Once the buyer reaches a position that it cannot improve, it will cease bidding or notify the child virtual network maestro to seek a provisioning alternative and renegotiate for the resource. The process is considered to be in equilibrium when all buyers are satisfied.

The dynamics of the auctioning process is illustrated in step-wise form in Figure 2. A delegate, operating on behalf of maestro, notifies the auctioneerCP that an event is taking place (1). It then updates the optimal price ($\boldsymbol{P_{ij}}$) and rate quantity ($\boldsymbol{R_{ij}}$) as provided by the maestro for appropriate auctioning (2), refreshing the seller state (3). The seller object then announces (4a) the available resource quantity and associated pricing for buyer bidding. In parallel, buyers will receive (4b) from children's virtual network maestros the desired strategy (allocation and cost) for their bidding (5). The auctioning agent mediates the auctioning process between buyers and sellers seeking successful auctioning equilibrium and optimal resource allocations. It maintains a bid list via the `update()` method (6) for allocations resulting from buyers' bidding, according to their resource valuations and recent bidding. The allocations can be retrieved at any time (7) by the auctioning agent to keep buyers informed. The auctioning agent then updates the results (new allocations and costs) at the AuctioneerCP (8). Information about allocations is also available to buyers through requests (`getAllocation()` method) (9). Delegates will then receive agreed price and rate allocations (10) and will pass the information to the maestro for child networks' policies according to their provisioned share of the parent resource. The maestro seeks closure by communicating the allocations (or denials) to child virtual network maestros through its delegates for final consideration. If agreed allocations are not satisfactory for any subscriber child network, the child maestro may invoke the auction process (again) with alternate specifications, and the previous child allocations and policies are voided by the parent maestro. Further details about the PSP auctioning model can be found in [20].

4 Virtual Network Capacity Scheduling

With the arbitrator, we introduce the notion of virtual link capacity-based scheduling driven by a parent-child hierarchy of virtual network provisioning policies. The arbitrator receives virtual network policy from the maestro over a slow timescale provisioning interval upon completion of the resource allocation process. The virtual link arbitrator manages the access and control of the parent link packet scheduler based on policy-driven virtual network capacity.

Leveraging similar ideas of flow quality of service semantics [5], (e.g., deterministic, statistical and best effort) we abstract the QOS class differentiation concepts and apply it to capacity-based 'provisioning' classes. The intent here is to provide more provisioning flexibility and control to the child virtual network for maximizing resource efficiency and QOS control and reduce the burden on the parent to manage the child domain and maintenance of low-level QOS service level agreements (SLA). Rather, moving the provider (parent) and subscriber (child) service models to be based more on provisioning SLAs with flexible capacity classes. Therefore, virtual network SLA maintenance is kept strictly on a virtual link capacity basis while parent policing and regulation treatment can be removed from the delay-sensitive models (to be managed autonomously by the child network services) and focused more on virtual link bandwidth sharing.

4.1 Arbitrator Design

The capacity arbitrator is based on a set of virtual network *capacity classes* and class *weight* policies that are distributed to the arbitrator component by the delegate on behalf of the maestro. Virtual network classes represent differentiated policy for provisioning capacity. Class weights are calculated (by the maestro) based on the following parameters: *rate_allocation, percentage, price* variables specified in the *AllocSpec*. The capacity classes and weights are translations of the negotiated resource allocations on individual child virtual resources during the auctioning process and are used as the virtual link scheduling policies. Capacity classes and weights are used by the arbitrator to differentiate child virtual network allocations and the ordering of packet delivery to the parent link resource.

During the spawning process, each virtual network is assigned a unique virtual network identifier to distinguish its traffic from other child network traffic. A *capacity class identifier* function is introduced into the arbitrator architecture prior to the child's link scheduler function to recognize QOS behavior treatment (e.g., best-effort, controlled load, expedited forwarding, etc.) associated with the child specific QOS architecture. This function interworks with a *switch vector* to assign each packet a stamp that associates it with a particular capacity class prior to its arrival at the routelet port packet link scheduler as illustrated in Figure 3. We refer to this procedure as capacity class mapping. Policy mappings are formed during the provisioning process and distributed during the resource allocation process to the child virtual networks. 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, 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 delay, controlled load and best-effort) 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. QOS class and capacity class mappings are considered part of the provisioning policy for child networks and stored within the policy cache of the supporting the maestro.

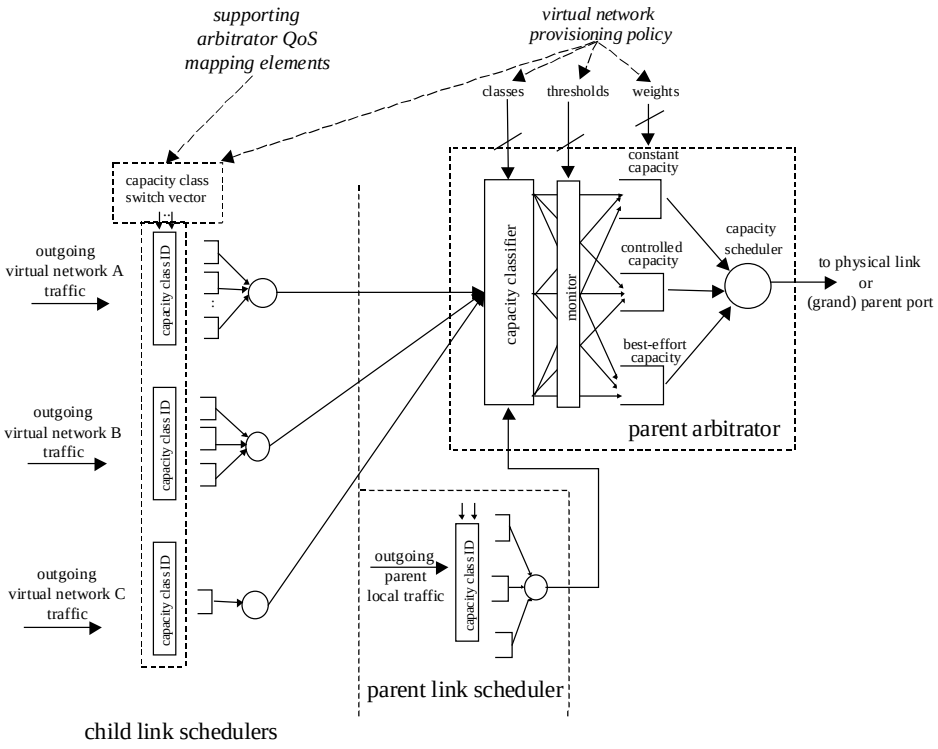

Fig. 3. Arbitrator

A *capacity classifier* is used to identify virtual networks and their capacity classes. The classifier queues incoming stamped packets (from the output of routelet port link schedulers) to the appropriate *capacity queue* structures (viz. constant, controlled, and best-effort). Individual capacity 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 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 parents output port. The capacity arbitrator leverages space (i.e., available resource bandwidth), time (i.e., provisioning interval length) and capacity-class abstractions to manage scheduling of its own user traffic and packets from child virtual networks onto the parent virtual link. The *capacity scheduler* services the capacity queues in priority order and weighted round-robin for same capacity class queues.

As illustrated in Figure 3, child network traffic is scheduled by the child's packet link scheduler associated with the routelet output port and similarly, the parent network routelet port. The introduction of the virtuosity arbitrator into the output port architecture merges both child and parent QOS scheduled traffic and provides coarse capacity scheduling of the composite traffic based on the allocated provisioning policies. It is important to note that the illustration represents the default virtual network resource management implementation, but is not the only parent option for managing child network traffic. Alternatively, the parent may override[5] the arbitrator function and integrate child network traffic through its local routelet port link scheduler.

Also illustrated in Figure 3, the monitor is central to the arbitrator, performing monitoring and policing on individual parent resources. Policing assures that child virtual networks are not consuming parent virtual networks resources above and beyond their allocation of the virtual link capacity. Policing actions (e.g., dropping, tagging or degrading to best-effort capacity class) is driven by virtual network policy.

5 Conclusion

We are implementing "spawning networks", a new class of open programmable networks. The Genesis Kernel lies at the heart of spawning networks capable of profiling, spawning, architecting and managing distinct virtual network architecture on-demand. In this paper, we have described a kernel-level plug-in module called "virtuosity" for the management of multiple spawned virtual networks. The virtuosity framework comprises a maestro, which performs distributed virtual network control; an auctioneer, which leverages economic models based on auctioning to perform resource allocation; and finally, an arbitrator which performs policy-based virtual network capacity scheduling.

6 Acknowledgement

This work is supported in part by the National Science Foundation (NSF) under CAREER Award ANI-9876299 and with 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).

---

[5] This also suggests that virtuosity, or at least key components of virtuosity are not required and may be substituted. The architectural selection and realization is based on the parent resource management policy set during the profiling phase and programmatically composed during the spawning phase of the life cycle process.

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

[3] Blake, S., et al. “A Framework for Differentiated Services”, draft-ietf-diffserv-framework-01.txt.

[4] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and Weiss W., “An architecture for differentiated services”, draft-ietf-diffserv-arch-02.txt, October 1998.

[5] Campbell, A.T., Coulson, G., and D. Hutchison, "A Quality of Service Architecture" , ACM SIGCOMM Computer Communication Review, Vol. 24 No. 2., pg. 6-27, April 1994.

[6] 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.

[7] Campbell, A.T., De Meer, H., Kounavis, M.E., Miki, K., Vicente, J., and Villela, D. A., "A Review of Programmable Networks", ACM Computer Communications Review, April 1999.

[8] Campbell, A.T., De Meer, H., Kounavis, M.E., Miki, K., Vicente, J., and Villela, D. A., "The Genesis Kernel: A Virtual Network Operating System for Spawning Network Architectures", IEEE 2nd International Conference on Open Architectures and Network Programmability (OPENARCH'99), New York, October 1998, pp. 115-127.

[9] Campbell, A. T., Vicente, J., and Villela, D. A., "Virtuosity: Performing Virtual Network Management", International Workshop of Quality of Service (IWQoS), London, June 1999.

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

[11] Duffield N., et al., “A Performance Oriented Service Interface for Virtual Private Networks”, draft-duffield-vpn-QOS-framework-00.txt. Work in progress.

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

[13] Keshav, S., and Sharma, R., “Achieving Quality of Service through Network Performance Management”, Proc. of NOSSDAV’98, Cambridge, July 1998.

[14] "The Integration of Real-Time Control with Management in Broadband Networks", Proceedings of the Workshop on Broadband Communications, Estoril, Portugal, January 20-22, 1992, pp. 193-204.

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

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

[17] Van der Merwe, J. E. and Leslie, I. M., "Switchlets and Dynamic Virtual ATM Networks", Proc Integrated Network Management V, May 1997.

[18] 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.

[19] Multiservice Switching Forum (MSF), http://www.msforum.org/

[20] Semret, N., and Lazar, A. A., “Design, Analysis and Simulation of the Progressive Second Price Auction for Network Bandwidth Sharing”, Technical Report CU/CTR/TR 487-98-21

[21] OPENSIG Working Group http://comet.columbia.edu/opensig/

[22] 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”, draft-rajan-policy-QOSschema-00.txt, October 1998. Work in progress.

[23] Rooney, S., Van der Merwe, J. E., Crosby, S. A., Leslie, I. M., “The Tempest: A Framework for Safe, Resource-Assured, Programmable Networks”, IEEE Communications Magazine, October 1998, pp 42-53.

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

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