A Programming Environment for On-Demand Service Overlays

by

Rahul Agrawal

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical [Computer] Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 20, 2004

Author_____
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by_____
Umar Saif
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Programming Environment for On-Demand Service Overlays
by
Rahul Agrawal

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 2004

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer [Electrical] Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Recent years have seen a growing interest in overlay networks. Overlay networks simplify the complications of creating and connecting distributed applications on the network by optimizing the underlying IP routing architecture. This thesis presents a programming environment (PEON) for automatically generating, monitoring and adapting an overlay network in response to application requirements and changing network conditions. The system automatically generates and maintains an overlay of routing modules to satisfy the application requirements. The runtime environment employs various constraint-satisfaction-programming (CSP) heuristics to allocate resources between competing applications with varying requirements of bandwidth, latency, and packet loss. The system additionally offers a "Reflection API" that allows an application to monitor and adapt the internal structure of its overlay to fine-tune its parameters.

Thesis Supervisor: Dr. Umar Saif
Title: Research Scientist

Acknowledgements

I would like to thank Umar Saif, my thesis advisor, for the tremendous time and effort he put into helping me design, implement, complete, and write my thesis. In addition, I would like to thank Steve Ward for his support and for providing me with the opportunity to research and work with network architectures. I would also like to acknowledge the O2S group at MIT. Without all of you, I would have never completed this thesis. Lastly, I would like to thank my parents for their continued support in each endeavor I pursue.

Table of Contents

1 Introduction **8**

1.1 Overview 8

2 Design Goals **14**

2.1 Problem Statement 14
2.2 Categorization of Problems 15
2.2.1 Same Source, Same Demand (SSSD) 16
2.2.2 Same Source, Different Demand (SSDD) 17
2.2.3 Different Source, Different Demand (DSDD) 18
2.3 Goal-Oriented Solutions 19

3 System Architecture **21**

3.1 Formulating the Problem 21
3.2 Demand Algorithms 22
3.2.1 Blocking Island Algorithm 23
3.2.2 SSSD Algorithm 25
3.2.3 SSDD Algorithm 27
3.2.4 DSDD Algorithm 30
3.2.5 ADD Algorithm 31
3.3 Overlay Architecture 32
3.3.1 Split Nodes 32
3.4 The Pebbles System 33
3.4.1 Pebbles System Overview 33
3.4.2 Initializing a New Node 34
3.5 Reflection API 35

4 Implementation **37**

4.1 Language Choice 37
4.2 API for QoS Overlay 37
4.2.1 API Overview 37
4.2.2 API Usage Example 37
4.3 Overlay Pebble 39
4.4 Lisp Server 40

5 Evaluation **43**

6 Conclusions and Future Work **46**

6.1 Deployment on PlanetLab 46

7 References **47**

Table of Figures

Figure 1 Constructing an overlay to provide CNN's telecast in Chinese 9
Figure 2 a) Nodes in bold are the nodes used in overlay creation. b) Arrows indicate nodal connectivity to send messages from Node A to Node I. 11
Figure 3 a) The physical connections PEON makes to connect the host node to the university nodes. b) Overlay abstraction of host node connected to university nodes. 12
Figure 4 a) The physical connections PEON makes to connect the host node to the university nodes via an intermediary node. b) Overlay abstraction of host node connected to university nodes via an intermediary node. 13
Figure 5 a) Overlay that satisfies three SSSDs. b) PEON adding a new branch to the routing tree to accommodate the addition of Node E. 17
Figure 6 An example of PEON solving a SSDD 18
Figure 7 The blocking island hierarchy for resource requirements {64, 56, 16}. The weights on the links are their available bandwidth. (a) the 16-BIG (b) the 56-BIG (c) the 64-BIG (d) the network. [1] 25
Figure 8 Network A – Contains 7 nodes {A, B, C, D, E, F, G} and 8 links {L1, L2, L3, L4, L5, L6, L7, L8} 26
Figure 9 The network connections created by the new multicast algorithm for demands {A, D, 5000000, -1, -1}, {A, E, 5000000, -1, -1}, and {A, F, 5000000, -1, -1} 27
Figure 10 Solutions generated by the blocking island heuristic for demands {A, D, 5000000, -1, -1}, {A, E, 10000000, -1, -1}, and {A, F, 5000000, -1, -1}. A routing loop from node A to node C exists. 29
Figure 11 Final network layout produced by SSDD after PEON prunes the routing tree. 30
Figure 12 Description of how the Pebbles System installs a pebble on a new node, connects two different nodes, and monitors each node. 35
Figure 13 The average amount of time PEON requires for calculating routing solutions when 0, 1, 2, and 3 intermediary nodes are in an overlay. 44
Figure 14 The average amount of time PEON requires for constructing an overlay when 0, 1, 2, and 3 intermediary nodes are in an overlay. 45

1 Introduction

1.1 Overview

The Internet was designed as a generic message relay, independent of application-level functions. Over the years, however, there has been a growing desire to *program the Internet*, either to ameliorate the austerity and quirks of IP-routing or to enhance the scalability, performance, and availability of widely-distributed applications.

This thesis addresses the problem of problem of dynamically composing widely-distributed applications across the Internet. Previous point examples of such applications range from firewalls, network address translation, proxy-caches, application building blocks, as well as more sophisticated wide-area applications like content-distribution networks, file-sharing applications, application-level multicast, secure service access, and robust routing architectures. A system which permits wide-area composition of distributed modules will permit more sophisticated applications to be constructed from such simpler components. For instance, consider a scenario in which a user wants to connect to CNN's webcast news but wants a Chinese subtitle to the added to the webcast stream. To receive the webcast, the user requires a connection that supports a bandwidth of 10 Mbps. This application involves two widely-distributed components: one that multicasts CNN's web content and the second component which adds Chinese subtitles to the webcast. Given such a request, our system sets-up a set of intermediary components which forward the network messages in accordance with the application QoS requirements (figure 1). To the user, the connected components appear as a single service which provides CNN's webcast in Chinese at 10 Mb/sec.

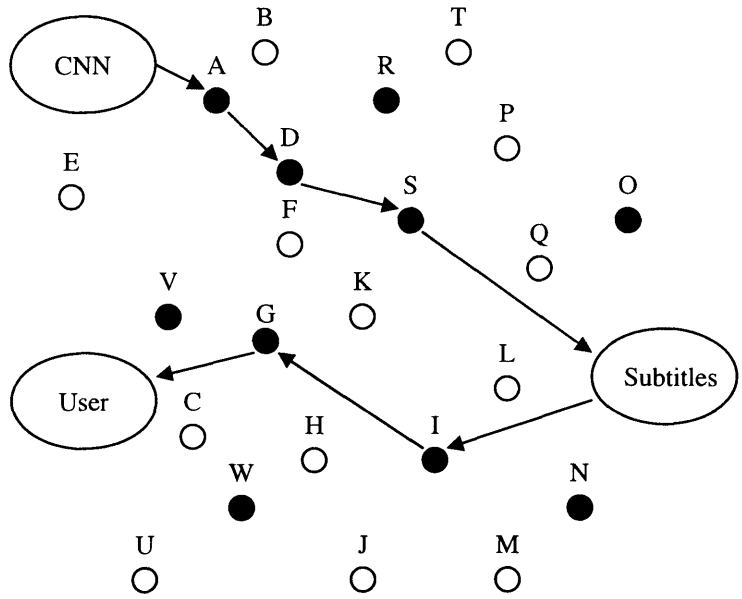

Figure 1 Constructing an overlay to provide CNN's telecast in Chinese

Overlay networks have emerged as a practical solution to control the semantics of message-forwarding on the Internet by punctuating its path with application-specific processing. Examples of such *overlay network architectures* include commercial virtual private networks (VPNs) [11] and IP tunneled networks, such as M-Bone [10]. Figure 2 shows a graphical depiction of the use of overlay networks for composing widely-distributed application. The nodes in bold indicate the set of nodes that form the overlay network.

This thesis provides an application programmer with a Programming Environment (PEON) so that he can compose widely-distributed applications from application components distributed over the Internet. The programmer specifies high-level connectivity requirements between the application components and stipulates QoS demands on each high-level connection. In the CNN example, the programmer specifies a high-level connection with QoS requirement of 10 Mbps between the CNN and the

Chinese subtitle service to create an overlay that produces CNN's webcast subtitled with chinese.

PEON connects widely distributed application components using the Internet to compose services desired by the programmer. To create these services, PEON automatically generates a topology for an overlay that supports QoS constraints required by the inter-component connections. Going back to the CNN example, based on the programmer's connectivity requirements, PEON generates a topology that links CNN's webcast component to the Chinese subtitle service at a bandwidth of 10 Mbps. By connecting to this topology, the user receives CNN's telecast subtitled with Chinese.

Overlays created in PEON satisfy programmer's QoS requirements despite changing characteristics in the network. If application components fail or bandwidths on connections change, PEON restructures the overlay such that the overlay continues to meet the programmer's demands. This scheme, therefore, permits a programmer to construct an overlay based only on high-level connectivity requirements of the application components.

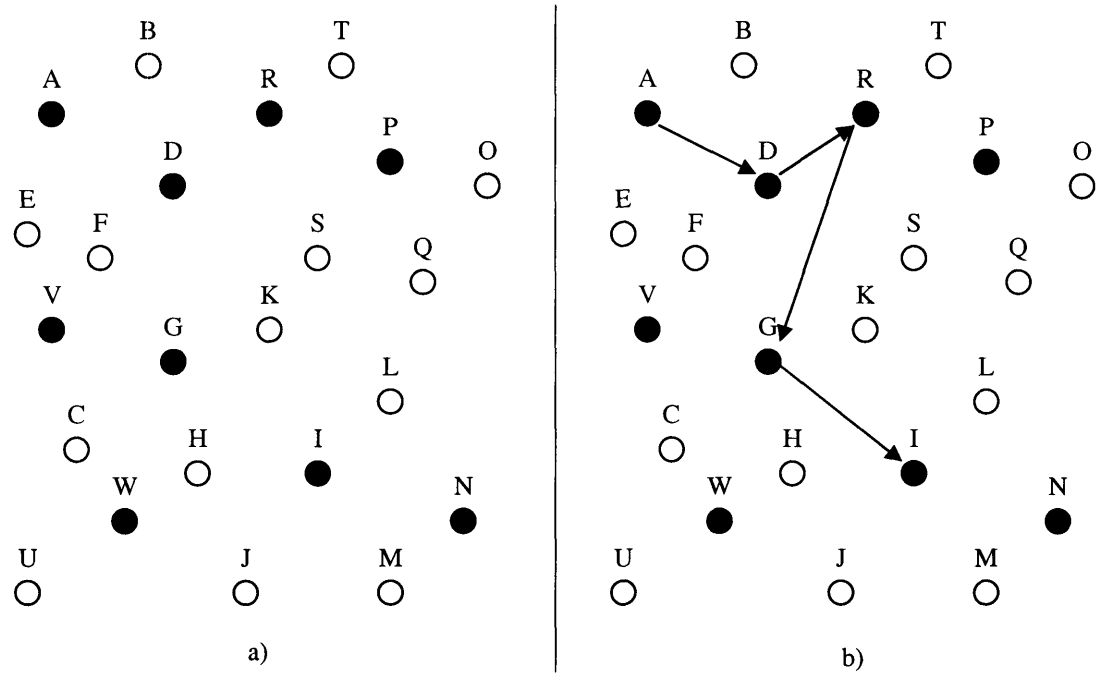

Figure 2 a) Nodes in bold are the nodes used in overlay creation. b) Arrows indicate nodal connectivity to send messages from Node A to Node I.

PEON allows the programmer to place QoS demands on an overlay network in the following ways: 1) *nodes*, which consist of the individual members of the overlay network 2) *links*, which connect the nodes and 3) *graph topology*, which define how PEON lays the nodes in the network. The QoS demands placed on the links include bandwidth, latency, and packet loss limitations. After the programmer specifies his QoS demands, PEON creates an overlay and then monitors and restructures the overlay if network conditions change. The following scenario exemplifies these ideas.

Suppose a programmer wants to send a multicast to three universities (Node A, Node B, Node C) from a source node (Host Node) at a rate of 5 Mbps. The maximum outgoing bandwidth of the source node is 5 Mbps. If the system were to create the overlay network so that each of the universities were connected directly to the host machine, then each university node would receive the data at a rate less than 5 Mbps. The data received by each university would be at a rate of 1.67 Mbps. Assuming the

destination nodes receive an equal share of the source node's bandwidth, the three destination nodes equally divide the total outgoing bandwidth of the source node, yielding a rate of 5 Mbps/3 = 1.67 Mbps. See Figure 3.

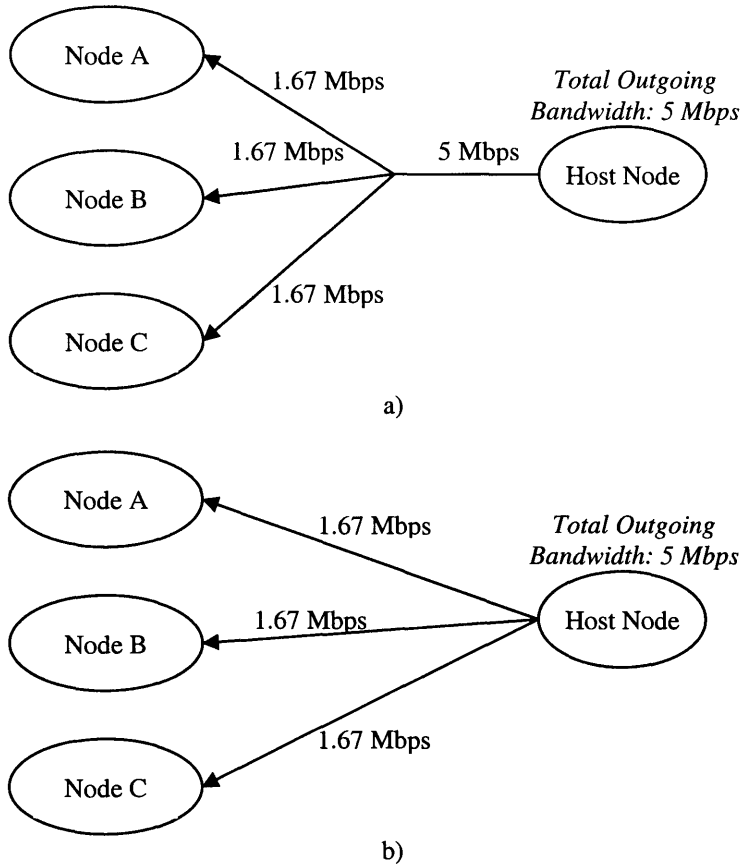

Figure 3 a) The physical connections PEON makes to connect the host node to the university nodes. b) Overlay abstraction of host node connected to university nodes.

To calculate the minimal outgoing bandwidth of the host node, we simply sum the receiving bandwidths of the destination nodes. In this case, since the programmer wants three destination nodes to receive data at a rate of 5 Mbps, the required outgoing bandwidth of the host node must be at least 3x5Mbps=15 Mbps. Unfortunately, the host node cannot support an outgoing bandwidth of 15 Mbps. To solve this problem, PEON finds a node that supports an egress bandwidth of 15 Mbps and an ingress bandwidth of 5 Mbps. PEON connects the host node to this *intermediate* node, and then connects the

three university nodes to the intermediate node. The intermediate node replicates the data it receives from the source node three times. It sends a copy of the data to each one of the university nodes. This new topology satisfies the programmer's QoS requirements. See Figure 4.

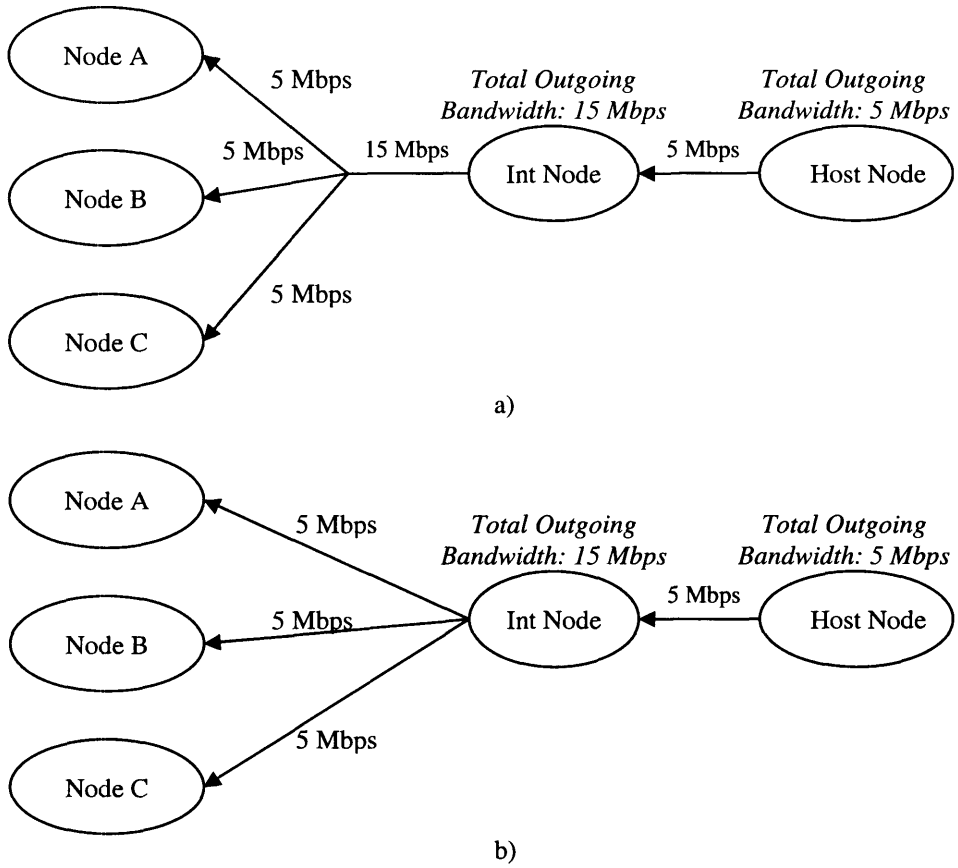

Figure 4 a) The physical connections PEON makes to connect the host node to the university nodes via an intermediary node. b) Overlay abstraction of host node connected to university nodes via an intermediary node.

This thesis describes the problems PEON proposes to solve, PEON architecture, and the algorithms used to implement PEON. Section 2 describes the goals that PEON satisfies. Section 3 describes the PEON architecture. Section 4 discusses the PEON implementation. It provides a full description of each PEON algorithm. Section 5 evaluates PEON performance and Section 6 is a summary of our presented work.

2 Design Goals

2.1 Problem Statement

In 1996, Jon Crowcroft and Zheng Wang published a paper stating that multi-metric constraint routing is a NP-hard problem in the general case [2]. The computation complexity of determining a route that satisfies a user's QoS requirements are determined by the composition rules of the metrics [2]. We define these composition rules as follows:

Definition: Let $d(i, j)$ be a metric for link (i, j). For any path $p = (i, j, k, \ldots, l, m)$, we say a metric d is *additive* if

$$d(p) = d(i, j) + d(j, k) + \ldots + d(l, m)$$

A metric d is *multiplicative* if

$$d(p) = d(i, j) * d(j, k) * \ldots * d(l, m)$$

A metric d is *concave* if

$$d(p) = min[d(i, j), d(j, k), \ldots, d(l, m)] \text{ [2]}$$

From the composition rules, delay is an additive metric, packet loss is a multiplicative metric, and bandwidth is a concave metric.

Wang and Crowcroft prove in their paper that given a network G, any n additive or m multiplicative constraints imposed on any path in G is a NP-hard problem [2]. The authors prove this theorem in their paper by reducing the problem to a Set Partition Problem, a well-known NP-hard problem [4]. More importantly, Wang and Crowcroft argue that when a concave metric is specified as one of the QoS requirements in

conjunction with either an additive or multiplicative constraint, then the problem of finding a path that satisfies the programmer's QoS demands is no longer NP-hard [2].

However, if multiple demands are specified, we can reduce the problem to a SAT and prove that the problem is still NP-hard, even if all demands have concave metrics. The problem that we address is satisfying multi-metric concave QoS demands. Given a set of nodes, we allow a programmer to deploy multiple applications on top of these nodes with a set of QoS demands.

We assume the programmer will always specify a concave metric, bandwidth, as at least one of his QoS demands. We note that most practical applications include a bandwidth constraint as part of its connectivity requirements.

PEON is designed for deploying applications for a small set of users. Such applications include chat applications and video teleconferencing. PEON creates each overlay on top of a cluster no larger than a hundred nodes. Each overlay will have no more than fifteen to twenty connected users. As such, PEON will not need to calculate more than a few solutions from a small group of nodes. Consequently, scalability with large networks or a large number of users is an issue that this thesis does not need to address.

2.2 Categorization of Problems

We have identified three cases PEON needs to solve to dynamically compose widely-distributed components: 1) Same Source, Same Demand 2) Same Source, Different Demand and 3) Different Source, Different Demand. To calculate a set of solutions for a multicast problem, PEON first determines which category the multicast

problem falls into. PEON then executes the algorithm for that multicast category to produce a solution.

Before describing each multicast category, we first define the terminology used in this thesis: Demands will be specified as $\{x, y, \beta, \lambda, \rho\}$ where x is the source node, y is the destination node, β is the bandwidth requirement, λ is the latency constraint, and ρ is the packet-loss restriction. PEON does not consider any constraints set to -1 (i.e. $\beta = -1$, $\lambda = -1$ or $\rho = -1$) when it is calculating a set of routing solutions.

The following sections describe each of the multicast categories in more detail.

2.2.1 Same Source, Same Demand (SSSD)

In a SSSD, the goal of the application programmer is to send one stream of data from one source to any number of destination nodes with the same QoS demands. For example, suppose four nodes, A, B, C, D, exist in a network. An application programmer wants to send one stream of data from node A to nodes B, C, and D. He wants each destination node to receive the data stream at a bandwidth of 5 Mbps and a latency of less than 2 milliseconds.

To solve this problem, PEON uses successive transitive closures to incrementally build a routing tree. PEON first calculates a solution for each demand. It then determines all unique links among the set of solutions and establishes connections based on these links. Figure 5a shows the overlay that satisfies these demands.

To accommodate a new SSSD, PEON first finds a solution that satisfies the new QoS demand. PEON then determines which links in the new solution do not already exist in the network. It establishes connections based on those links that do not exist. Instead of

calculating a set of new solutions for both the existing and new demands, PEON modifies the existing routing tree in this fashion. The newly established connection creates a new *branch* to the routing tree. See Figure 5b.

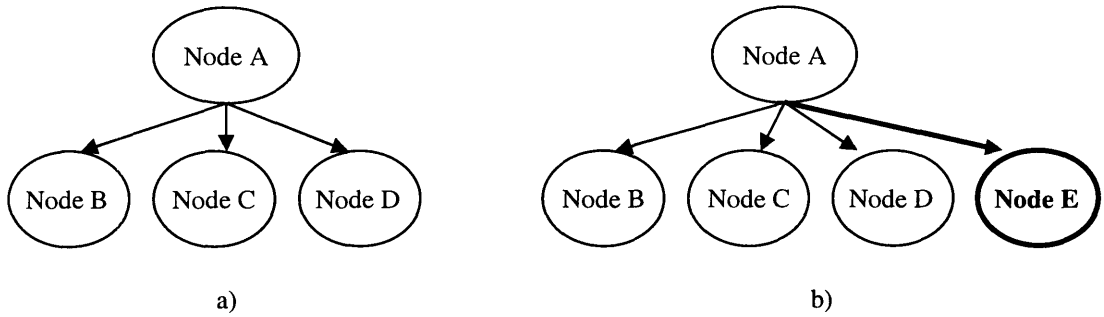

Figure 5 a) Overlay that satisfies three SSSDs. b) PEON adding a new branch to the routing tree to accommodate the addition of Node E.

The addition of a new branch to the routing tree prevents disruption from already connected nodes receiving data and leads to efficient routing paths.

2.2.2 Same Source, Different Demand (SSDD)

In a SSSD, the application programmer imposes the same QoS constraints on all paths from the source host to the destination hosts. In a SSDD, the application programmer imposes different QoS constraints on all paths. There still exists only one source host responsible for sending data to all destination hosts, but the demands placed on the paths are different from one another.

For example, consider a network consisting of three nodes, A, B, C. The application programmer specifies the demands {A, B, 5000000, -1, -1} and {A, C, 10000000, -1, -1}. To find a solution to this SSDD, PEON first calculates solutions for each individual demand. In a SSSD, PEON determines all unique links from the set of solutions and makes the connections based on the unique links. PEON cannot perform the

same algorithm for a SSDD. The set of solutions produced by PEON for a SSDD might introduce a loop in the network. As a result, PEON executes an extra step to prune the routing tree to eliminate loops in the routing solution[1]. PEON determines the eliminated routes by establishing which routes cannot satisfy all of the programmer's QoS requirements.

Figure 6a shows a routing solution to satisfy the demand {A, B, 5000000, -1, -1}. When the programmer specifies a new demand {A, C, 10000000, -1, -1}, the solution yielded by PEON creates a redundant path from nodes A, E, and D, as shown in Figure 6b. PEON eliminates this loop by eliminating the links that cannot satisfy both sets of demands. After pruning the routing tree, the final set of routing solutions is shown in Figure 6c.

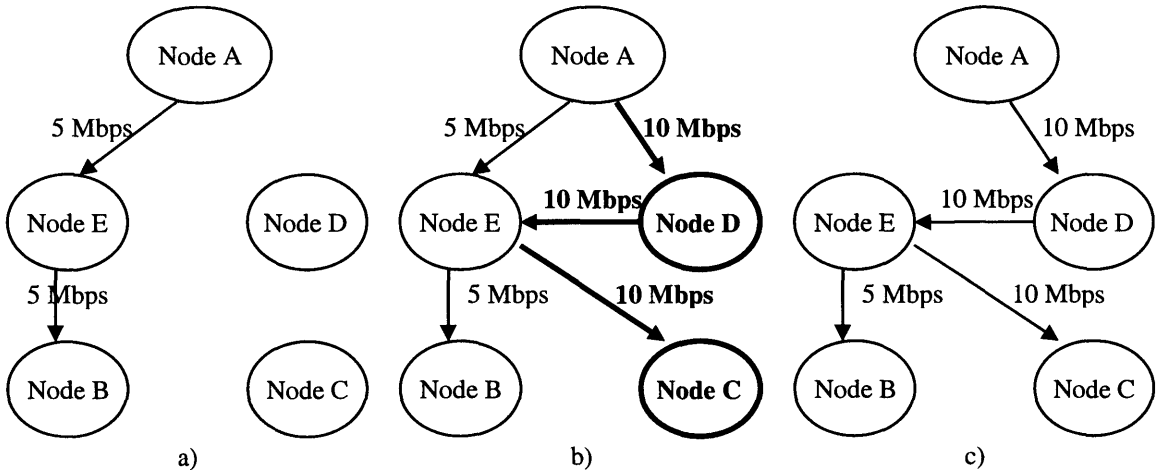

Figure 6 An example of PEON solving a SSDD

2.2.3 Different Source, Different Demand (DSDD)

[1] The PEON prunes the routing tree only if the new demand varies in the same metric as the already existing demands. For example, if the existing SSDDs vary only in latency, then the new demand must only vary in latency as well, otherwise the PEON does not prune the tree.

When an application programmer wants to create multicasts originating from different nodes, he is trying to solve a DSDD. The QoS demands can greatly vary between the different multicasts the programmer deploys. An application programmer specifying a DSDD imposes the harshest constraints on resource allocations. Despite the stringent constraints, PEON will always find a solution to the programmer's demands if one exists. We explain the details of the architecture and the implementation of these algorithms in Sections 4 and 5.

2.3 Goal-Oriented Solutions

After PEON determines the routes that satisfy the programmer's demands, the system establishes the connections and monitors the changing conditions of the network. If conditions in the network change, PEON will restructure the overlay to account for the altered characteristics of the network. This abstraction of goal-oriented programming allows the programmer to focus his efforts on network connectivity rather than on the low-level planning of such a network.

PEON uses the Pebbles System architecture [5] as the underlying technology for deploying and managing on-demand service overlays. PEON installs the Pebbles System on each node in each overlay network to facilitate and monitor the connectivity between each set of nodes, the availability of new nodes, and the disconnection between nodes in the case of nodal failure. In other words, the Pebbles System is responsible for determining when network characteristics change and for informing PEON about these changes. When conditions change, PEON recalculates new routes to reorganize the

overlay so that it continues meeting the programmer's QoS demands. We describe the Pebbles System in more detail in Section 3.

3 System Architecture

3.1 Formulating the Problem

The PEON runtime system models the network as a set of transitive closures to allocate demands. The resource allocation in network problem is defined as follows:

> ***Given*** *a network composed of nodes and links, each link with a given resource capacity, and a set of demands to allocate,*
>
> ***Find*** *one route for each demand so that the bandwidth requirements of the demands are simultaneously satisfied within the resource capacities of the links.*

By taking successive transitive closures, PEON enables the use of constraint satisfaction [6] techniques to allocate resources between overlay nodes. We formulate the resource allocation problem as a constraint satisfaction problem (CSP) in the following way: *variables* are demands, the *domain* of each variable is the set of all routes between the endpoints of the demand, and *constraints* on each link must ensure that demands routed do not exceed the resource capacities of each link. A *solution* is the set of routes, one for each demand, respecting the capacities of the links.

We propose the idea of using transitive closures to cluster nodes that have similar bandwidths. By grouping nodes in this manner, we open the problem of allocating QoS demands to the myriad of CSP heuristics available. We can use the ideas of constraint propagation, forward checking, value ordering, variable ordering, and other CSP heuristics to determine the set of routes to solve QoS multicast demands.

PEON solves multiple QoS multicast demands by first determining which category the set of demands fall into. It then executes one of three algorithms based on the type of demands specified by the programmer. By formulating the problem into a CSP problem, the PEON can use a variety of CSP heuristic to find a set of routes that satisfy the QoS demands. PEON then makes the connections based on the solutions returned by the CSP heuristic.

3.2 Demand Algorithms

An application programmer specifies his QoS requirements in PEON when he wants to create a constraint-based overlay network. As mentioned earlier, there are three types of multicast problems: SSSD, SSDD, or DSDD. When calculating a set of solutions, PEON first determines which category the QoS demands fall into. After evaluating the programmer's demands, PEON executes the algorithm specific to that category for resolving which resources to allocate for creating the overlay network.

When the programmer introduces a new demand, PEON determines whether the new demand falls into the same multicast category as the pre-existing QoS demands. If it does, PEON simply executes the same demand algorithm to satisfy the new demand. If the demand falls into a different category from the pre-existing QoS demands, then PEON executes the Addition of Different Demands (ADD) algorithm to determine a solution.

Each algorithm uses a CSP heuristic to calculate a set of solutions. PEON uses the Blocking Island abstraction proposed by Frei and Faltings [1]. Before discussing how the SSSD, SSDD, DSDD, and ADD algorithms are designed, we first present a brief

overview of the Blocking Island abstraction. The architectures of the rest of the demand algorithms are explained thereafter.

3.2.1 Blocking Island Algorithm

3.2.1.1 Blocking Islands

Frei and Faltings define a β-blocking island (β-BI) for a node x as:

The set of all nodes of the network that can be reached from x *using links with at least* β *available resources, including* x. [1]

Figure 7d shows all 64-BIs for a network [1].

β-BI have some fundamental properties. Blocking islands partition the network into equivalence classes of nodes given any resource requirement. Blocking islands are unique and identify global bottlenecks, which are effectively inter-blocking island links. These inter-blocking island links could be links for which there is no alternative route with the desired resource requirement if the inter-blocking island links are links with low-remaining resources.

In addition, blocking islands guarantee that at least one route satisfying the bandwidth requirement of an unallocated demand d = (x, y, β) exists if and only if its endpoints x and y are in the same β-BI. Furthermore, all links that could form part of such a route lie inside this blocking island [1].

3.2.1.2 Blocking Island Graph

Blocking islands are used to build the β-blocking island graph (β-BIG), a simple graph representing an abstract view of the available resources. Each β-BI is clustered into

a single node and there is an *abstract link* between two of these nodes if there is a link in the network joining them. Figure 7c is the 64-BIG of the network of Figure 7d. The abstract links denote the *critical links*, since their available bandwidths do not suffice to support a demand requiring β resources.

Frei and Faltings explain that to identify the bottlenecks for different βs, a recursive decomposition of the BIGs is built in decreasing order of the requirements. This layered structure of the BIGs is a *Blocking Island Hierarchy*. See Figure 7.

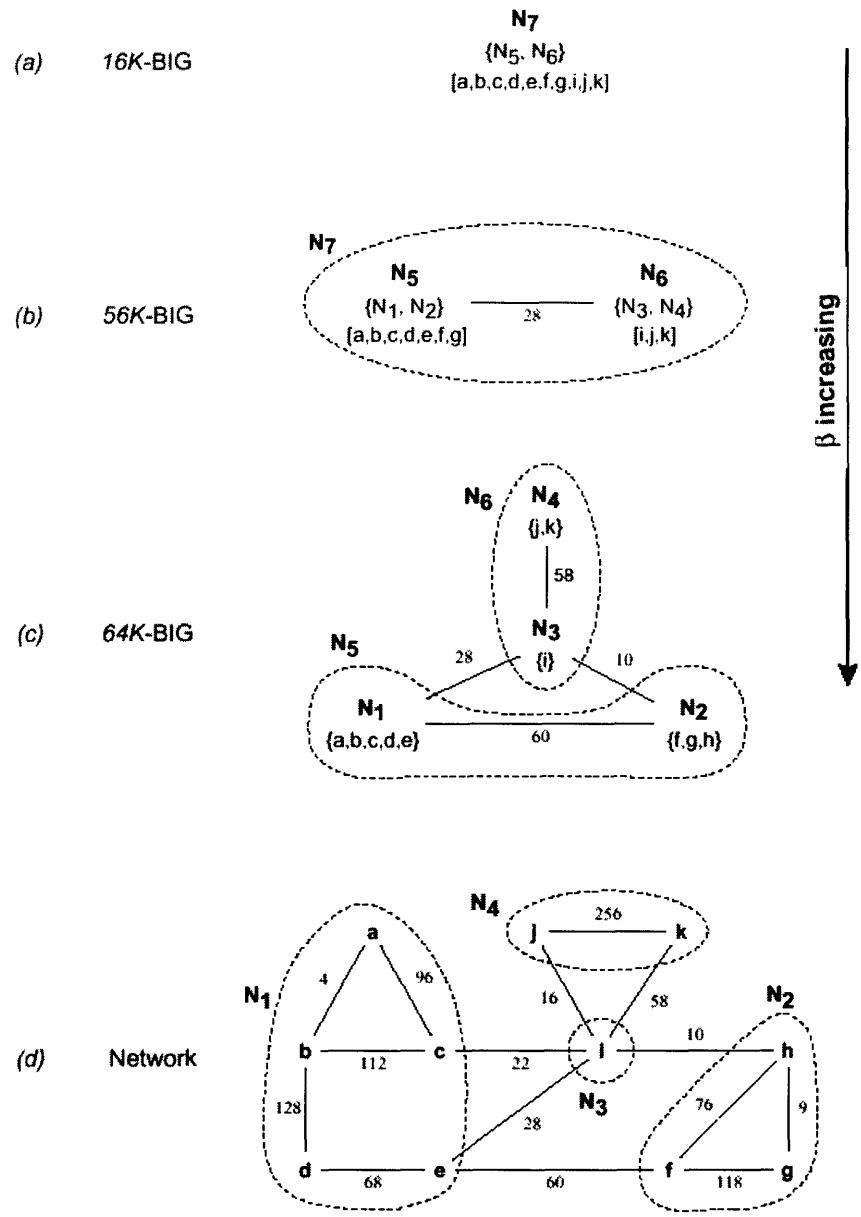

Figure 7 The blocking island hierarchy for resource requirements {64, 56, 16}. The weights on the links are their available bandwidth. (a) the 16-BIG (b) the 56-BIG (c) the 64-BIG (d) the network. [1]

3.2.2 SSSD Algorithm

3.2.2.1 SSSD Algorithm Description

The SSSD algorithm uses the blocking island algorithm to find a solution to each of the programmer's demands separately. If the blocking island algorithm cannot provide a solution to each of the programmer's demands, then PEON returns to the programmer that it cannot produce a solution that meets each of the programmer's constraints. If the blocking island algorithm can provide a solution for each of the programmer's demands, then PEON retrieves all the unique links of each solution and establishes the connections between the nodes in the network based on the unique links.

3.2.2.2 SSSD Example

Suppose a programmer wants to create an overlay in Network A, shown in Figure 8. He specifies three demands: {A, D, 5000000, -1, -1}, {A, E, 5000000, -1, -1}, and {A, F, 5000000, -1, -1}. Since all demands have the same source host (node A) and the same demands (Bandwidth = 5 Mbps, Latency = nil, Packet Loss = nil), PEON will execute the SSSD algorithm to satisfy the programmer's QoS requirements.

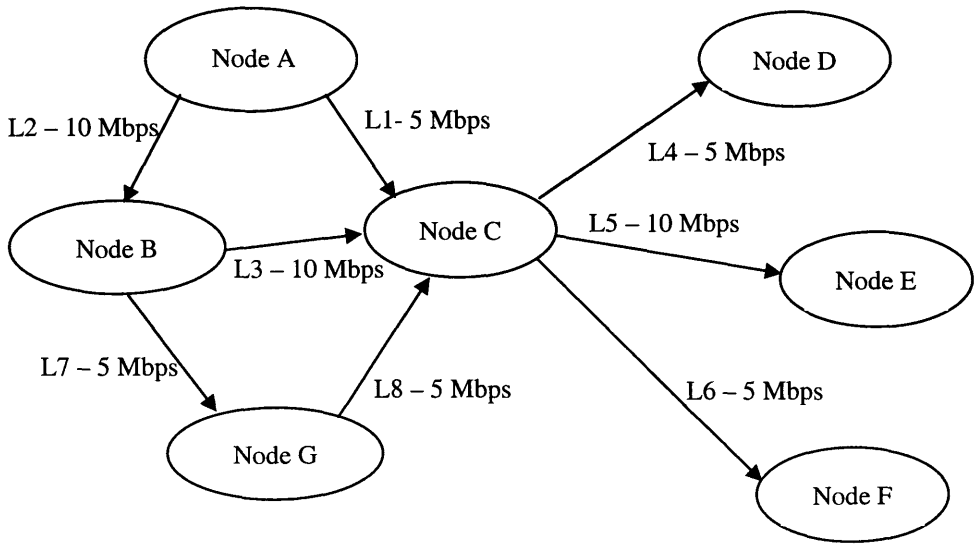

Figure 8 Network A – Contains 7 nodes {A, B, C, D, E, F, G} and 8 links {L1, L2, L3, L4, L5, L6, L7, L8}

The SSSD algorithm first uses a CSP heuristic, in this case the blocking island algorithm, to find a solution for each demand. The blocking island algorithm will return solutions of links {L1, L4} for demand {A, D, 5000000, -1, -1}, links {L1, L5} for demand {A, E, 5000000, -1, -1}, and links {L1, L6} for demand {A, F, 5000000, -1, -1}. The PEON retrieves all the unique links {L1, L4, L5, L6} among the set of solutions. It then establishes the connections between the nodes in Network A based on the unique links.[2] Figure 9 shows the resulting network connections based on the SSSD algorithm.

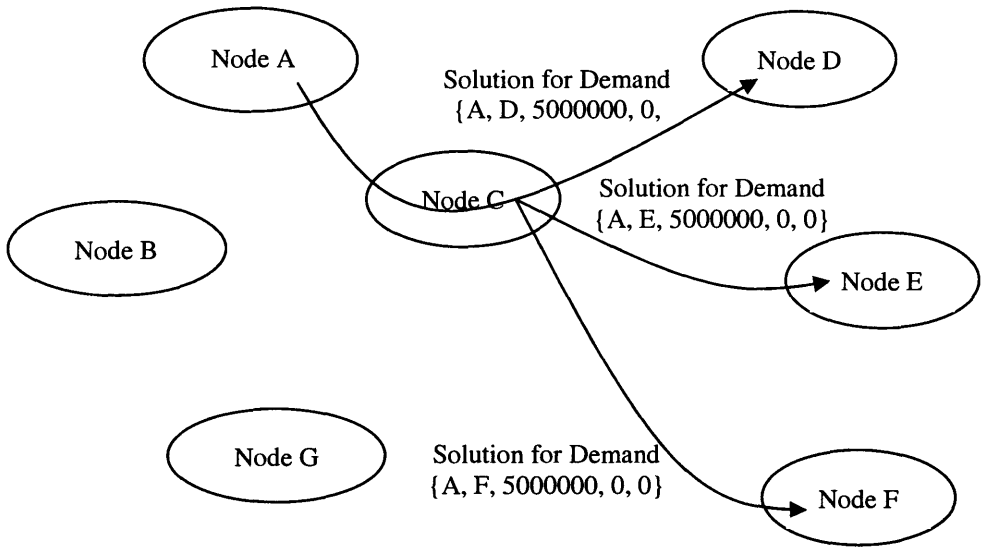

Figure 9 The network connections created by the new multicast algorithm for demands {A, D, 5000000, -1, -1}, {A, E, 5000000, -1, -1}, and {A, F, 5000000, -1, -1}

3.2.3 SSDD Algorithm

3.2.3.1 SSDD Algorithm Description

[2] The system installs split nodes in the network if it needs to replicate data. See section *Split Nodes* for understanding where the system places split nodes in the network and how split nodes work.

The SSDD algorithm is similar to the SSSD algorithm. SSDD first finds a solution for each QoS demand specified by using a CSP heuristic. After finding a set of solutions, PEON calculates all unique links. The difference between SSDD and SSSD is that SSDD performs an extra step to prune the routing tree. SSDD eliminates any potential *routing loops* that may exist in the set of solutions. Routing loops exist in a solution when a node has multiple input links. These routing loops form redundant paths in the network that result in inefficient use of network resources.

To eliminate routing loops, PEON first determines the set of nodes that have multiple input links. To resolve which set of input links to eliminate, PEON evaluates the characteristics of each of these links and eliminates those links that cannot satisfy all specified QoS demands. The following example illustrates how the SSDD algorithm functions.

3.2.3.2 SSDD Example

An application programmer is creating an overlay on Network A (Figure 8). He specifies three demands: {A, D, 5000000, -1, -1}, {A, E, 10000000, -1, -1}, and {A, F, 5000000, -1, -1}. Though all of these demands have the same source node, one of the demands has a different bandwidth requirement. As such, PEON will execute the SSDD algorithm to find a solution to satisfy these demands. After running the blocking island CSP heuristic on each demand, PEON receives solutions of {L1, L5}, {L2, L3, L5} and {L1, L6} for demands {A, D, 5000000, -1, -1}, {A, E, 10000000, -1, -1}, and {A, F, 5000000, -1, -1} respectively. PEON calculates the unique set of links to be {L1, L2, L3, L4, L5, L6}.

If PEON established connections on these set of links, all of the QoS demands are satisfied. Unfortunately, making these set of connections result in an inefficient use of resources due to the routing loop in the network. The routing loop causes the source node to send the same data to another node across two distinct paths in the network. In this case, node A sends the same data to node C across two different paths. See Figure 10.

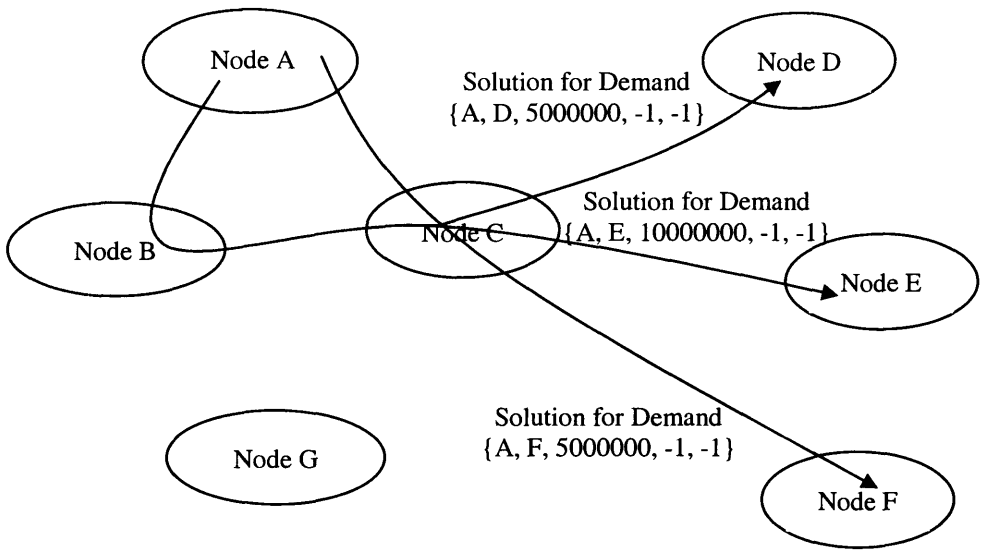

Figure 10 Solutions generated by the blocking island heuristic for demands {A, D, 5000000, -1, -1}, {A, E, 10000000, -1, -1}, and {A, F, 5000000, -1, -1}. A routing loop from node A to node C exists.

To eliminate this routing loop, the SSDD algorithm performs an optimization function. PEON first determines all nodes that have multiple input links. Of the set of links coming into a node, PEON only retains one of these links and eliminates all others. Since all demands have the same QoS metric, PEON retains the link whose characteristics best satisfy the specified demands. PEON eliminates all other links coming into the node. This elimination process removes redundant network connections and still meets the multicast QoS demands.

In this example, node C has multiple input links, L1 and L3. Since L1 has a lower bandwidth than L3, PEON eliminates L1 and retains L3. Node C becomes a spilt node and splits the data it received from node B to nodes D, E, and F. Figure 11 shows the final routing paths after PEON executes the SSDD algorithm.

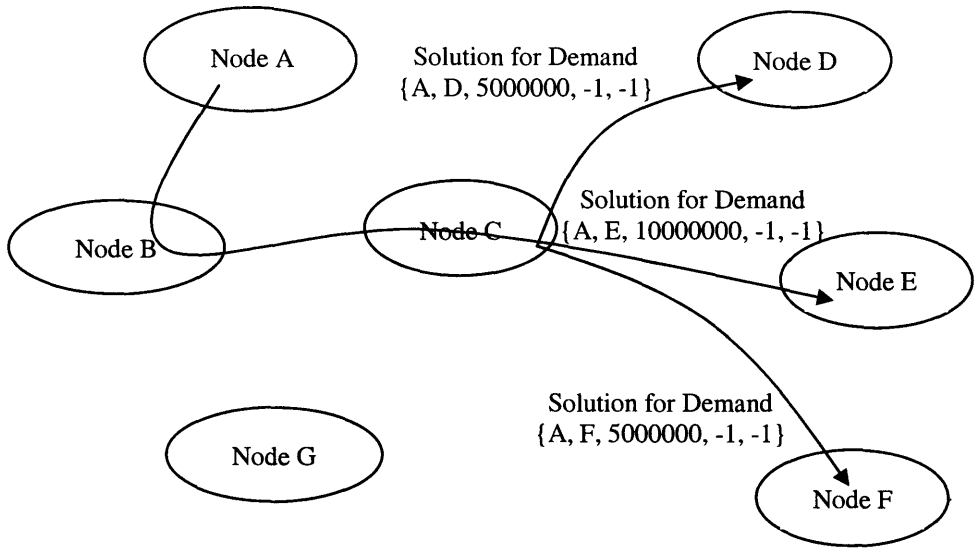

Figure 11 Final network layout produced by SSDD after PEON prunes the routing tree.

3.2.4 DSDD Algorithm

If a programmer wants to create an overlay network with multiple source hosts, PEON will separately allocate resources in the network for each demand. For example, if a programmer wants to send two streams of data across a link *l* at 5 Mbps, PEON cannot allocate 5 Mbits of bandwidth to meet these QoS demands. Instead, PEON must allocate 10 Mbits of bandwidth across *l*. Since the programmer is sending two different streams of data, each stream requires a bandwidth of 5 Mbps, yielding a total of 10 Mbps across *l*.

The DSDD algorithm is radically different from the SSSD and the SSDD algorithms. Rather than finding a solution for each demand separately, PEON uses CSP

heuristics to find solutions for all demands concurrently. The results produced by the CSP heuristics are the final connections the PEON must establish to fulfill the QoS demands.

3.2.5 ADD Algorithm

3.2.5.1 Purpose of ADD Algorithm

Oftentimes, a programmer specifies new QoS demands after the PEON constructs an overlay. To satisfy this new demand, PEON first determines if the new demand falls into the same multicast category as the pre-existing QoS demands. For example, if PEON constructed an overlay by executing the SSSD algorithm, then the new demand must also fall into the SSSD category. If it does, PEON simply executes the algorithm associated with this multicast category. In this example, PEON would execute the SSSD algorithm.

If the new demand falls into a category different from the pre-existing QoS demands, then PEON executes the ADD algorithm. Rather than re-categorizing pre-existing demands with the new demand and then determining a new set of solutions, the ADD algorithm makes the minimal number of changes to the existing overlay network to fulfill the new demand. This prevents disconnecting users in the overlay and prevents a complete reorganization of the overlay network.

3.2.5.2 ADD Description

The ADD algorithm is designed to disrupt the minimal number of connected users in the overlay network when satisfying the new demand. We also designed ADD to find a solution to satisfy all QoS demands (including the new demand) if one exists.

When the application programmer specifies a new QoS demand d, PEON allocates all pre-existing n demands. PEON uses a CSP heuristic to find a solution for d with the n allocated demands. If the CSP heuristic cannot find a solution, then PEON allocates $n - 1$ demands and de-allocates $n - (n - 1) = 1$ demand. With these new allocations, PEON uses a CSP heuristic to find a solution for d and the de-allocated demand. If PEON cannot find a solution, it tries all possible combinations of allocating $n - 1$ demands, de-allocating one demand, and using a CSP heuristic to find a solution for d and the de-allocated demand. If PEON still cannot find a solution, PEON allocates $n - 2$ demands, de-allocates $n - (n - 2) = 2$ demands, and uses the CSP heuristic to find a solution for d and the two de-allocated demands. Again, PEON tries all possible combinations of allocating $n - 2$ demands, de-allocating two demands, and using a CSP heuristic to find a solution for d and the de-allocated demands. This process continues until PEON can no longer allocate any demands or until the CSP heuristic produces a solution.

If the CSP heuristic produces a solution at any point during ADD execution, the ADD algorithm immediately terminates. PEON disconnects all connections that are not present in the newly produced solution. It establishes any connections that appear in the new solution that did not previously exist in the overlay network.

3.3 Overlay Architecture

3.3.1 Split Nodes

After PEON calculates the set of routes to create an overlay network, PEON determines the location of *split nodes* in the network. Split nodes are responsible for

replicating data at different points in the network. They also distribute the replicated data to multiple nodes. Split nodes take the data received at their input connector and replicate it *n* times, where *n* is the number of outgoing links the split node contains. In the solution to the SSSD example in Network A, node C acts as a split node. Node C replicates the data it receives from node A and distributes the data to nodes D, E, and F.

When PEON creates a connection between a source node and a destination node, PEON first checks whether the source node contains an *overlay pebble*[3]. If the source node does not contain an overlay pebble, then PEON installs the overlay pebble on the source node. PEON then creates an output connector on the pebble to create a link between the source and destination nodes. Each node only has one input connector, so any node that receives data through its input connector instantly sends the data to all its output connectors. This design easily replicates and sends data to various nodes.

3.4 The Pebbles System

We use the Pebbles System architecture [5] as the underlying technology for deploying and managing on-demand service overlays. An *overlay manager* installs the Pebbles System on each node in each overlay network to facilitate and monitor the connectivity between each set of nodes, the availability of new nodes, and the disconnection between nodes in the case of nodal failure.

3.4.1 Pebbles System Overview

[3] Overlay pebble is discussed in the section *The Pebbles System*

The designers of the Pebbles System [5] formalize the needs of users as *goals*. To achieve these goals, the Pebbles System separates the planning code from the components so that it can address the repertoire of different resources. To meet user goals, the Pebbles System searches for different techniques or methods to achieve each objective. Each technique is evaluated and compared against the goal of the user to determine whether the objective can be met. At the same time, the Pebbles System considers alternative methods at lower-levels when breaking the user-goal into sub-goals. Put more succinctly, resolving the goal typically takes the form of a goal tree, choosing the best implementation to reach the user goal at each decision node.

3.4.2 Initializing a New Node

When the programmer introduces a new node to an overlay network, an overlay manager installs pebbles on the new node to facilitate monitoring and connecting to other nodes in the network. The overlay manager will use an *overlay pebble* to establish a link between the new node and another overlay node to expand the entire overlay. Once PEON establishes a link, the overlay manager will monitor this newly added node using the pebbles installed on the new node. The overlay manager can use this information to determine whether new resources have been added to the overlay network or if certain nodes in the network have been disconnected. As network resources change, the overlay manager can use this information to reroute links to optimize the overlay network. See Figure 12 below.

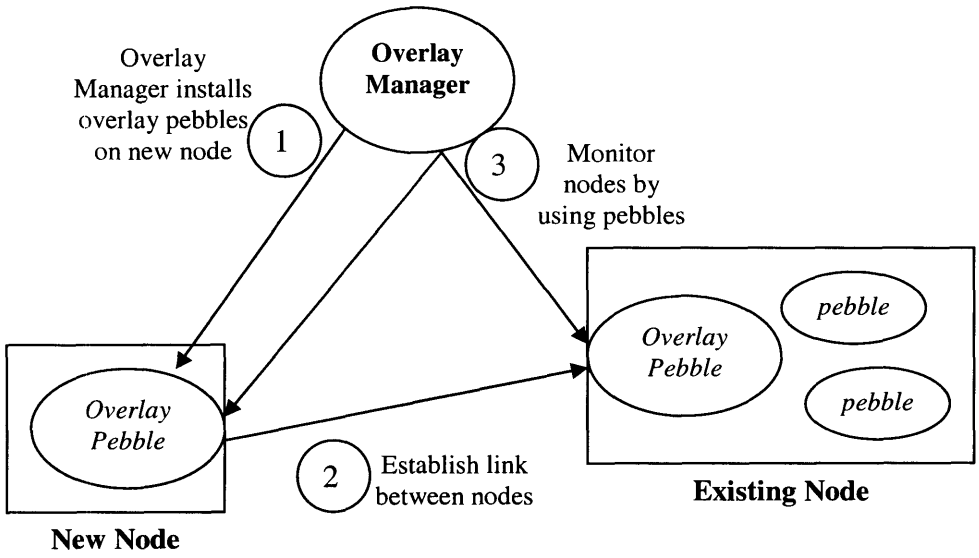

Figure 12 Description of how the Pebbles System installs a pebble on a new node, connects two different nodes, and monitors each node.

3.5 Reflection API

PEON has a centralized form of control when it creates an overlay network. PEON is solely responsible for calculating and instantiating the links between the nodes in the network. Maintaining this centralized form of control after overlay creation can be detrimental if PEON fails, since the overlay network will immediately be eradicated. To avoid this, we designed the overlay network to move to a more distributed form of control after its creation. This decentralized form of control prevents overlay annihilation if parts of the network fail.

To move the network to a distributed form of control, we present a Reflection Application Programming Interface (API) to the programmer. This API allows the programmer to stipulate individual QoS demands on specific parts of the network. After the programmer specifies his constraints, an overlay manager will monitor the individual

QoS demands to ensure that the specified constraints are satisfied. If at any point the network changes, the overlay manager will restructure the overlay so that the PEON will fulfill the programmer's demands.

For example, a programmer uses the Reflection API if he wants the bandwidth between two nodes to be at least 10 Mbps. Once specified, the overlay manager will monitor the set of links connecting these two nodes. If at any point the bandwidth capacities of any of the links fail to meet the programmer's QoS demand, the overlay manager will reroute the set of links so that the PEON will still meet the programmer's demands.

4 Implementation

4.1 Language Choice

Our language choice to write PEON is python. To create an overlay network, an application programmer specifies his demands in PEON. The simplest language to do this in is python. We chose python because python is the easiest language to integrate with the Pebbles System. This is because the Pebbles System is written in python.

4.2 API for QoS Overlay

4.2.1 API Overview

We present an API to the application programmer that enables him to specify his QoS demands when creating an overlay network. The API permits the programmer to dictate which nodes he wishes to include in the overlay, the source nodes that send data, the destination nodes that receive data, and the QoS demands on the routing links. After the programmer specifies his QoS demands, PEON constructs the overlay network.

The API we present to the programmer is written in python, so the programmer must specify his QoS constraints in python as well. The classes and methods that the programmer uses to build an overlay network are found in Appendix A. The next section presents an example that demonstrates how we intend the programmer to use the overlay API.

4.2.2 API Usage Example

Suppose an application programmer is attempting to create an overlay network. He wants to send a CNN telecast to three university nodes: MIT, Stanford, and Berkeley. The telecast he wants to send is a live video telecast, so he decides he needs to impose two QoS demands. To ensure adequate quality of the telecast, the programmer decides he needs a bandwidth of 5 Mbps and a latency less than 300 milliseconds.

To construct this overlay, the programmer must first executes the *init()* function to set the IP address and the port number of the Lisp Server.[4] He then retrieves the hosts that will later become the source and destination nodes of the multicast. To obtain a host, the programmer uses the method *getHost()* and passes in the URL and the port number of the host he wants to retrieve.

After obtaining the hosts, the programmer creates a node on each one of these hosts by instantiating the *Node* class. The programmer passes in a host and the URL of an overlay pebble to the constructor of the *Node* class. The *Node* constructor will create a node on the host passed in by installing the overlay pebble on the host.

Depending on the function of each node, the programmer then makes either an *output* or *input* connector on each of the nodes. The programmer creates an output connector on a node if the node sources data. He creates an input connector on a nodeif the node receives data. In this example, the programmer creates an output connector on the node that has CNN has a host, and he creates input connectors on the hosts that have MIT, Stanford, and Berkeley as their hosts.

After creating the overlay node, the programmer specifies his QoS requirements by instantiating the *VirtualPath* class. For a bandwidth of 5 Mbps and a latency less than

[4] The Lisp Server contains the code for executing the blocking island algorithm. The Lisp Server is talked about in more detail in the *Lisp Server* section.

300 milliseconds, the programmer simply passes in "Bandwidth>5000000" and "Latency<0.3" to the *VirtualPath* constructor.

After the programmer specifies all nodes and QoS demands, the programmer calls *Connect()* to create the overlay. He passes in the QoS demands, the data output connector, and all data input connectors to the *Connect()* constructor. PEON then creates the overlay based on these set of commands.

The following code exemplifies how the programmer uses the API to create the overlay network:

```
# Gets the URL of the Overlay pebble to install on the hosts
OVERLAY_PEB = "http://o2s.lcs.mit.edu/myPebble.py"

# Runs the init function to access the Lisp Server
init()

# Obtains all the source and destination hosts
source = getHost("www.cnn.com", "8439" )
mit = getHost("web.mit.edu", "7743")
stanford = getHost("www.stanford.com", "4854")
berkeley = getHost("www.berkeley.edu", "8328")

# Creates a node on each host obtained
cnnNode = Node(source, OVERLAY_PEB)
mitNode = Node(mit, OVERLAY_PEB)
stanfordNode = Node(stanford, OVERLAY_PEB)
berkeleyNode = Node(berkeley, OVERLAY_PEB)

# Creates an input connector on the source host and output connectors on the
#destination hosts
cnnCon = Connector(cnnNode, "output")
mitCon = Connector(mitNode, "input")
stanfordCon = Connector(stanfordNode, "input")
berkeleyCon = Connector(berkeleyNode, "input")

# QoS requirements specified
constraints = VirtualPath("Bandwidth>5000000", "Latency<0.3")

# Overlay network created with the specified nodes and QoS requirements
overlay = Connect(constraints, cnnCon, mitCon, stanfordCon, berkeleyCon)
```

4.3 Overlay Pebble

The application programmer installs an overlay pebble on each node he creates. PEON also installs an overlay pebble on all intermediary nodes it uses. The overlay pebble is designed to replicate all data that comes through its input connector to all its output connectors. To replicate data, the overlay pebble reproduces the input data *N* number of times, where *N* is the number of output connectors the overlay pebble contains. It then sends each copy of the data to each of its output connectors.

In some cases, the programmer might need to process the data that comes through each intermediary node. We designed the overlay pebble with an *AppCallback()* function that allows the programmer to manage each piece of data received by the pebble. Rather than requiring the programmer to write his own overlay pebble, the programmer subclasses the overlay pebble and rewrites the *AppCallback()* function. This gives the programmer complete control in handling each piece of data that comes through each node.

4.4 Lisp Server

Christian Frei and Boi Faltings chose to write and implement their blocking island algorithms in Common Lisp Object System (CLOS). Since both the language used to write PEON and the language the application programmer uses to specify his QoS demands are python, we had to implement an interface between python and CLOS for PEON to access and execute the blocking island algorithms.

We considered several options when we decided what type of interface to implement between the two languages. For one, we wanted to isolate the blocking island code from PEON code so that if another and more efficient CSP heuristic became

available, we could easily replace the blocking island code. Secondly, rather than installing the blocking island code on each node, we wanted to have only one copy of the blocking island code available for the entire system. If we ran multiple instances of PEON, we preferred each version of PEON running the same copy of the blocking island code rather than each PEON running their own copy of the blocking island code. With this design, if we ever needed to change the blocking island code, we would only need to change one copy of the code rather than several versions of it.

Considering these design goals, Extended Markup Language—Remote Procedure Call (XML-RPC) is the best implementation to interface the two languages. XML-RPC allows an application to call code that we can place on a computer different from the computer running the application. As such, we use a Lisp Server [7] to place the blocking island code on a computer different from the computer running the code to create the overlay network.

There is one drawback to using the Lisp Server. Each call made to the server can potentially take a large amount of time before the Lisp Server returns the results back to the application. This is because each call to the server and the results returned from the server must traverse the internet. If the network is highly congested, long latency delays can prolong the rate at which the application receives solutions.

Despite these delays, the greatest number of times the application accesses the Lisp Server is n times, where n is the number of QoS demands. In the case where all demands have different sources, the application accesses the Lisp Server only once.[5]

[5] For a more detailed explanation on how many times the application accesses the Lisp Server, see section on *Demand Algorithms*.

Consequently, the delay incurred by accessing the Lisp Server is not substantial, even if the internet becomes congested.

5 Evaluation

The performance characteristics are dependent upon the number of nodes required to construct the overlay network. The time taken to construct the overlay increases as PEON uses more intermediary nodes to construct an overlay network.

We performed four different tests to assess how quickly PEON calculates the solutions to satisfy multiple QoS multicast demands. We also performed tests to assess the amount of time PEON requires to construct an overlay network. Each test consisted of using the SSSD algorithm, since SSSD takes longer to execute than the DSDD algorithm. The DSDD algorithm accesses the Lisp Server only once, whereas the SSSD algorithm accesses the Lisp Server for each QoS demand specified. Each test consisted of one source node and three destination nodes. The number of intermediary nodes required to satisfy the QoS demands varied between the different tests. Figure 13 shows the average amount of time PEON requires to calculate solutions for overlays with varying numbers of intermediary nodes.

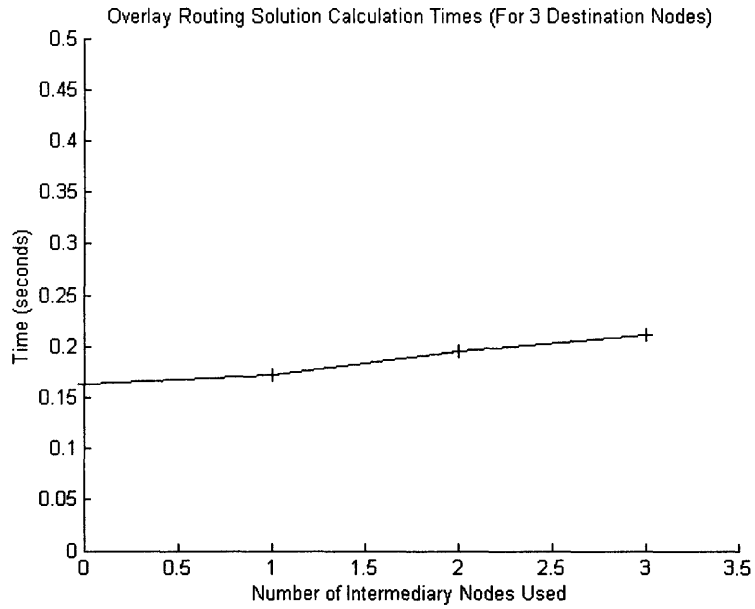

Figure 13 The average amount of time PEON requires for calculating routing solutions when 0, 1, 2, and 3 intermediary nodes are in an overlay.

From this graph, we can deduce that the time required calculating solutions increases linearly by about 15 milliseconds per intermediary node.

Figure 14 shows the amount of time PEON requires to create the overlay network with varying numbers of intermediary nodes.

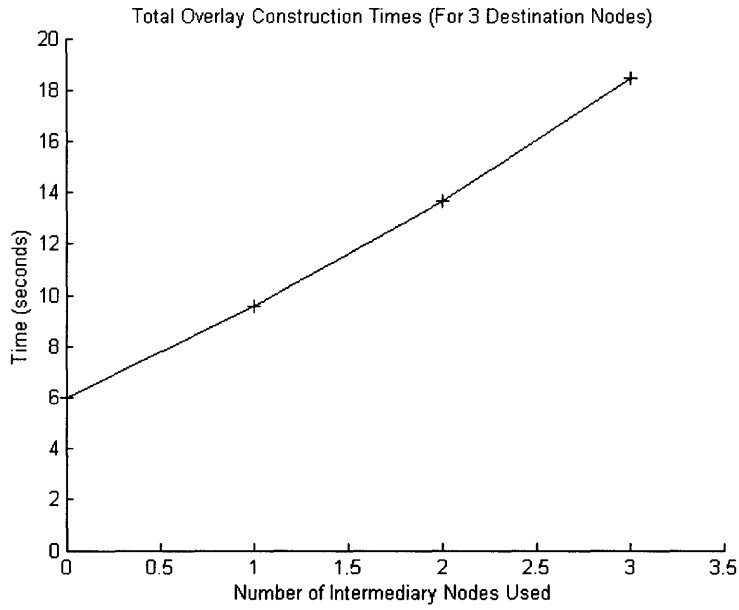

Figure 14 The average amount of time PEON requires for constructing an overlay when 0, 1, 2, and 3 intermediary nodes are in an overlay.

From this figure, we can deduce the time required for PEON to construct the overlay network increases linearly by about 5 seconds for each intermediary node introduced in the overlay. We attribute the increase in overlay construction time to the Pebbles Environment, since PEON constructs the overlay in the Pebbles Environment.

It is easy to see that the total time required to construct an overlay network is dependent upon the number of intermediary nodes that need to be connected, rather than the time required for executing the PEON algorithms.

6 Conclusions and Future Work

Developing and deploying overlay network architectures are an effective solution to the growing demand for programming the Internet. Providing programmers with a set of tools to develop their applications for the Internet and to allow a significant client population to utilize their services presents the right opportunity for the innovation of novel ideas. We can rapidly program these ideas with PEON without having to compete with the existing infrastructure of another network. This system goes one extra step by promoting on-demand service overlays that are dynamically deployed and managed by end-host applications to account for changing demands and network conditions. This system will comply with clients' requests by connecting clients to overlay networks that are specific to their needs.

6.1 Deployment on PlanetLab

We have tested all algorithmic implementations on networks that are no larger than eight to ten nodes. The next phase of evaluation and testing is to implement PEON on a larger test-bed of nodes. This will determine how well these algorithms scale with the network. One such network where testing of this sort can be done is PlanetLab [8]. Currently, PlanetLab has over 350 nodes available that we can use to construct overlays for different types of applications.

7 References

[1] C. Frei and B. Faltings. Resource Allocation in Networks Using Abstraction and Constraint Satisfaction Techniques. In *Principles and Practice of Constraint Programming*, 204-218, 1999.

[2] Z. Wang and J. Crowcroft. Quality of Service Routing for Supporting Multimedia Applications. *IEEE Journal on Selected Areas in Commuincation*, September 1996.

[3] A. Janardhanan. Bandwidth Allocation Planning in Communication Networks: Christian Frei, Boi Faltings. University of Nebraska-Lincoln.

[4] H. Zhu and O. Ibarra. On Some Approximation Algorithms for the Set Partition Problem. *The 15th Triennial Conference of International Federation of Operations Research Society (IFORS)*. August 1999.

[5] U. Saif, H. Pham, J. M. Paluska, J. Waterman, C. Terman, S. Ward. "A Case for Goal-Orientated Programming Semantics." System Support for *Ubiquitous Computing Workshop at the Fifth Annual Conference on Ubiquitous Computing* (UbiComp'03).

[6] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, UK, 1993.

[7] S. Autexier, S. Hess, M. Pollet. XML-RPC for Allegro Lisp. http://www.ags.uni-sb.de/~pollet/software/allegro-xmlrpc.html. March 28, 2002.

[8] PlanetLab. http://www.planet-lab.org/. April 27, 2004.

[9] J. Touch. Dynamic Internet Overlay Deployment and Management Using the X-Bone. *Computer Network*, pp117-135. July 2001.

[10] Eriksson, H., "MBone: The Multicast Backbone," Commuincations of the ACM, Aug. 1994, pp. 54-60.

[11] Scott, C., Wolfe, P., Erwin, M., *Virtual Private Networks*, O'Reilly & Assoc., Sebastapol, CA, 1998.

Appendix A: API for QoS Overlay

Class NetHost

Constructor Summary
NetHost(String ip, String port) Creates a host where pebbles can be installed on. The host has an ip address of ip and a port of port.

Method Summary	
string	**getPort**() Returns the port of the host.
string	**getIP**() Returns the IP address of the host.
string	**getName**() Returns the name of the host.
string	**getBIName**() Returns the blocking island name of the host. The blocking island name translates all ":" to "^".
Node[]	**getNodes**() Returns all nodes that installed on the host.
void	**addNode**() Adds a new node to the host
void	**eraseNode**(Node node_to_erase) Deletes node_to_erase from the array of installed nodes only if node_to_erase is installed on the host.
void	**eraseAllNodes**() Removes all the nodes from the host.

Class Node

Constructor Summary
Node(NetHost host, String pebble) Creates a node that is installed on host with pebble.

Method Summary	
PebbleHandler	**getPebHandler**() Returns the pebble handler of the pebble installed on the node.
Host	**getHost**() Returns the host of the node.
string	**getPebble**() Returns the pebble installed on the node.
string	**makeConName**() Returns a unique name for a connector that is instantiated on the node

	with an overlay pebble.

Class Connector

Constructor Summary	
Connector(Node node, String name) Creates an output connector on node with the name name.	
Method Summary	
Node	**getNode**() Returns the node on which the connector is installed.
string	**getName**() Returns the name of the connector.

Class VirtualPath

Constructor Summary	
VirtualPath(String *args) Creates a set of restrictions that can be imposed on a path in an overlay network. Currently, three constraints can be placed on a path – bandwidth, latency, and packet loss. Each constraint must have the name of the constraint passed in first, then the level of the constraint, and lastly the value of the constraint. For example, a bandwidth constraint of 5 Mbps must be specified as "Bandwidth>5000000".	
Method Summary	
Int	**getBWConstraint**() Returns the value of the bandwidth constraint specified. 'False' is returned if no bandwidth constraint is specified.
string	**getBWLevel**() Returns the bandwidth level of the bandwidth constraint. 'False' is returned if no bandwidth constraint is specified.
float	**getLatConstraint**() Returns the value of the latency constraint specified. "-1" is returned if no latency constraint is specified.
string	**getLatLevel**() Returns the latency level of the latency constraint. "-1" is returned if no latency constraint is specified.
float	**getPackLossConstraint**() Returns the value of the packet loss constraint specified. "-1" is returned if no packet loss constraint is specified.
string	**getPackLossLevel**() Returns the packet loss level of the packet loss constraint. "-1" is returned if no packet loss constraint is specified.

Class Connect

Constructor Summary
Connect(VirtualPath virtPath, Connector outputCon, Connector *inputCons) Creates an overlay network with the QoS requirements specified in virtPath. outputCon is the output connector of the source host from which the data originates. *inputCons is a list of input connectors of the destination hosts to which the data is ultimately sent to.

Method Summary	
VirtualPath	**getVirtPath**() Returns the virtual path imposed on the overlay network.
Connector	**getOutputCon**() Returns the output connector of the source host of the overlay network.
Connector[]	**getInputCon**() Returns an array of all the input connectors of all the destination hosts of the overlay network.

Method init

Method Summary	
void	**init**() Performs all the necessary initialization functions to create a QoS constrained overlay network, for example, setting the IP address of the Lisp Server that contains the blocking island code. Must be executed before an overlay can be created.

Method getHost

Method Summary	
NetHost	**getHost**(String ip, String port) Returns the NetHost object that matches the ip address and the port. An error message is returned if no host matches the arguments passed in.