OPENET: An Open and Efficient Control Platform for ATM Networks[*]

Israel Cidon[†‡], Tony Hsiao[#], Asad Khamisy[#], Abhay Parekh[¶], Raphael Rom[†‡] and Moshe Sidi[‡]

Abstract

ATM networks are currently moving from the experimental stage of test-beds to a commercial state where production networks are deployed and operated. The ATM Forum PNNI (Private Network to Network Interface) standard introduces an architecture suited for an internetwork which, in principle, can also be used as an intra-network nodal interface. However, the current PNNI falls short in providing an acceptable solution due to severe performance limitations in intra-network operation, limited functionality and the lack of open interfaces for functional extensions and services.

OPENET is a common, open and high-performance network control platform based on performance and functional enhancements to the PNNI platform. It is designed to address the issues of interoperability (being vendor independent), scalability (in terms of network size and volume of calls), high performance (in terms of call processing latency and throughput) and functionality. OPENET is mainly an intra-networking extension to current PNNI. It is compatible with PNNI in the internetwork environment where large networks must be partitioned according to natural topological or organizational boundaries.

The major novelty of the OPENET architecture (compared to the current PNNI) is its focus on network control performance. A particular emphasis is given to the increase of the overall rate of connection handling, to the reduction of the call establishment latency and to the efficient utilization of the network resources. These performance enhancements are achieved by the use of a native ATM distribution tree for the dissemination of utilization update, light-weight call setup, take down and modification signaling, the use of fast setup and takedown with the future option to implement them in hardware, and extensive use of caching and pre-calculation for route computation. OPENET also extends PNNI in terms of functionality. It utilizes a new signaling paradigm that better supports fast reservation and multicast services, a rich control communication infrastructure which enables the development of augmented services leveraging the existing functions, messaging system and information of the network control platform.

† Sun Microsystems Labs, 901 San Antonio Rd., Palo Alto, CA 94305.
MMC Networks, Inc. Sunnyvale, CA.
¶ FastForward Networks, San Francisco, CA.
‡ Department of Electrical Engineering, Technion, Haifa 32000, Israel.

[*]This work was performed in and sponsored by Sun Microsystems Labs.

1. Introduction

Network control is the set of mechanisms and processes that handles the routing, establishment, admission, modification, maintenance and termination of connections. The actual information that is exchanged among end-users is transparent to network control. Efficient network control is critically important in the performance of high-speed real-time applications, and is a key factor in justifying the cost of evolving ATM technology [1]. New interactive applications (such as browsers) open and terminate, in quick succession, many connections of different types such as text, image, video, audio. The fast handling of large number of connections is likely to be essential to the success of ATM. Another important requirement of network control is that it enables the network to operate at high levels of efficiency, so that given a topology and link capacities, many calls can be supported with acceptable levels of quality of service. A close look at network control architectures reveals that low-level functions such as the dissemination of topology and utilization updates are extremely important in ensuring efficient and reliable network control [2].

The quality of the network control platform is measured by its *performance, reliability and functionality*. Performance is measured by the network control throughput, i.e., the maximal rate at which the network can process new calls, the call processing latency and the link utilization efficiency. Network control reliability is judged by the way faults such as link and nodal outage, control message errors and soft errors are handled. Since the ATM network control actually reserves resources such as bandwidth and VC identifiers, unused resources must always be expeditiously released after call termination (including call termination that is forced by an outage).

The network control functionality is judged by the level of service that the network control provides to users. Functions such as QoS negotiation, supporting multi-level call priority, third party setup, policy routing, mobile naming support and many more, may be needed by particular network owners. For example, a company that provides a personal (mobile) communication service via its private ATM terrestrial infrastructure must include on-line efficient tracking of mobile users as part of its network control. This calls for the ability to add functions to the network control according to the intended network use, utilizing an open interface through which such functions can be added

In this paper we present the OPENET architecture designed and implemented at Sun Microsystems [3]. OPENET is an open and high-performance network control platform based on performance and functional enhancements to the PNNI standard [4]. It is designed to address the issues of interoperability (being vendor independent), scalability (in terms of network size and volume of call establishment and call property changes per link or switch), high performance and functionality. In addition, OPENET provides the ability to efficiently integrate customer specific network control functions and value added services through open interfaces (APIs) which enable the programmer to use the basic communication functions as well as the information resources of OPENET.

Performance critical elements are designed with the option to port them to hardware implementation in the future. Such elements are designed using light-weight and simple protocols and as much as possible using unicell messages. In addition, OPENET can be easily partitioned among multi-processors (such as the combination of link, switch and external general purpose processors). OPENET is an extension to PNNI so it is compatible with PNNI in the internetworking environment where large networks need to be partitioned according to natural boundaries. Finally, OPENET makes several functional improvements over PNNI such as the ability to setup whole multicast trees as a single routing and setup operation, the ability to negotiate setup parameters with the network and an efficient mechanism of fast modification of existing connection parameters without terminating and setting it up again (on the fly reservation).

2. Key Design Ideas and Rationale

OPENET can be viewed as both a complement and enhancement to PNNI. PNNI was originally targeted at inter-networking control (and therefore seems to fall short in term of efficiency for intra-network control). OPENET is targeted at intra-networking control. Within a network (such as a campus intranet backbone) connection setups are expected to be rather frequent. Therefore, network control efficiency is a key design issue of OPENET. In addition, OPENET is focused on being an open architecture, by providing an extensive support for new services and network control extensions through open APIs.

Efficiency of network control is measured by three basic quantities: (i) Call throughput which describes the maximal rate at which the network can process new calls or change the parameters associated with existing calls. Larger throughput implies that more control operations can be performed per unit of time, or alternatively, more users can be supported. (ii) Call latency which expresses the time elapsed between the request to establish a call or change its parameters and its completion. Smaller latencies yield faster setups and are key to modern applications (multimedia, web, network computing). (iii) Resource utilization efficiency which measures how well the network control exploits the available resources to satisfy user requests. This quantity can be expressed in terms of blocking probability or overall (maximum) network revenue.

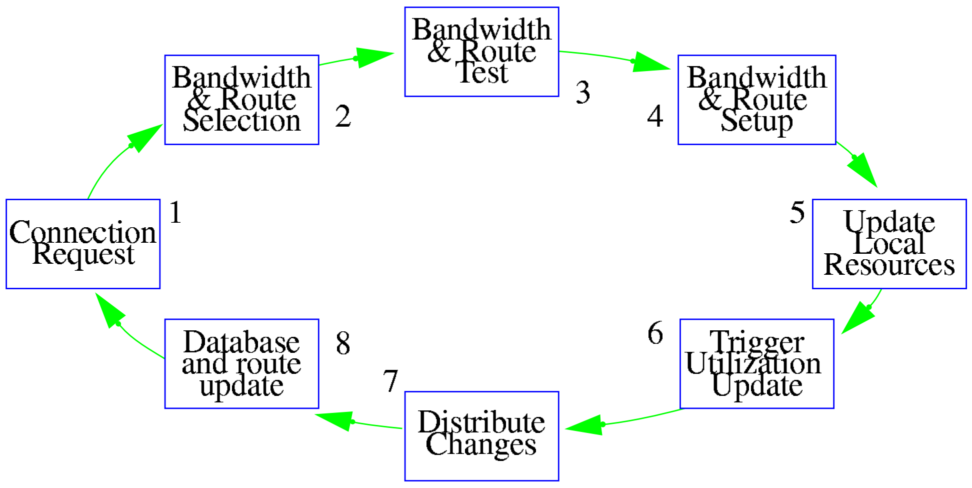

Figure 1: Network Connection Control Cycle.

Let us examine the typical setup sequence of operations of a modern control platform such as PNNI [4], NBBS [2], APPN [8] and OPENET. These networks are typically based on maintaining at every network node an updated view of the network topology and load. They employ a mechanism known as *link state algorithm* [11] to maintain and update this view via dissemination of local information (topology and utilization update [2]). We describe the process of establishing connections and updating the routing data-bases as a connection control cycle depicted in Figure 1. Our starting point is the event of a user requesting to open a connection; the destination address is first resolved in order to locate its hosting switch and port (step 1). Next a route is calculated using the local topology view (step 2). Then, an attempt to setup the call is made using a distributed setup procedure (setup signaling). Note that since route calculation can be done concurrently at several sources there is a possibility of "collision" among concurrent request which will result in some of the resources taken away before the setup signaling can lock them. Therefore, availability of resources is checked again (step 3) and then reservation is confirmed and locally setup (steps 4–5). If any of the previous steps fails, the connection is rejected. At that point if some link utilizations have changed enough to trigger a topology database update, a broadcast action is taken (steps 6–7). When this information is recorded at all nodes and is ready to be used for other call setups (step 8) we have completed the process of setting up the call including the process of making the network fully updated again for the route calculation of new calls. Call termination process is similar but most of the functions (such as route calculation, admission etc.) are skipped.

The number of operations (or CPU cycles) which are required to process all calls defines the call throughput of the network control platform. The time it takes to pass from step (1) to step (5) is the call setup latency. The time it takes to complete the whole cycle (steps (1) to (8)) and the accuracy of the triggered update have a major impact on the network resource utilization efficiency. This is because the fast completion of the control cycle, reduces the "collision" time window among different calls starting at different nodes which attempt to utilize the same link resources. The accuracy of the update increases this efficiency in particular at high loads, by enabling sources to "squeeze" links up to the maximum usage. Similarly, the route computation process quality has a major impact on the maximal number of calls that can be fit at the same time. Consequently, in order to improve (all three components of) the network control efficiency, the control cycle should be processed as fast as possible, utilizing as little CPU cycles as possible, while keeping the accuracy and promptness of updates and the quality of the routing procedure at high level.

Our general approach in designing OPENET is "optimistic", namely, we assume that most of the time the network is reliable and control messages do arrive at their destinations. This is true since topological changes (such a link failures) are infrequent compared to the time scale by which control tasks are completed. Based on this assumption, we attempt to optimize the system for the most frequent events, so that most of the time, the control will be very efficient. Handling exceptional and rare events, such as loss of control messages or their out of order arrivals (which are addressed in our design) in a very efficient way is often very complex. In our design we trade performance for simplicity in such rare cases.

OPENET uses source routing i.e., the source computes the entire route based on the data available to it. This relieves intermediate nodes along the path from performing additional route computations. The quality of the route clearly depends on the accuracy of the global topology and utilization data available at the source. OPENET delivers this data efficiently and promptly. The premise of this efficiency is the separation of topological and utilization data. To update the topology database at each node, OPENET uses an OSPF-like algorithm that essentially floods any message carrying topological change.

Since topological changes are infrequent and need to overcome unpredictable topology outages, flooding is a reasonable approach. However, utilization updates are very frequent. Therefore, use of flooding is inefficient here because of the following. First, flooding implies that nodes get the same information many times over multiple links (actually over all links) and therefore waste a large amount of CPU cycles. Second, the forwarding of the flood message is done in software at rather higher level of the protocol (as topology sequence numbers need to be checked) resulting in excessive delays. Third, since the forwarding requires software involvement the node needs to react as fast as possible to each flood message which results in a large number of interrupts (and therefore CPU cycles). Batching such updates (say, polling them periodically) will introduce additional latency in the flood progress. Last, PNNI or OSPF flood requires the use of large messages (in PNNI it is several hundreds bytes) despite the fact that the utilization information is fairly short. Hence, in OPENET we devised a novel construct, called the *distribution-tree*, that is a multi-point to multi-point tree on which utilization updates are distributed (see Section3.2). The distribution-tree proves to be a very efficient mechanism to distribute utilization information as it allows using single cell update messages, the use of hardware forwarding and the ability to batch updates together.

To further expedite route selection at the source and reduce setup latency, OPENET uses extensively *route caching*. Each source computes and caches continuously best routes for several QoS classes. This implies minimal on-demand route computation and short waiting before the source route can be used.

An additional key design issue in OPENET insisting on *light-weight* signaling and protocols. This is important for achieving the goals of low latencies and low computational overhead per call. It uses single cell messages to signal frequent operations, such as reservation changes and connection take down. Error recovery is mostly performed on an end-to-end basis rather than hop-by-hop to reduce the number of costly timer operations for a single call setup or change. In particular, OPENET handles the modification of an existing call parameters not as a new call but as a fast reservation operation along the existing VC, thus, eliminating extra operations such as route computation and the process of source routing. When hop-by-hop error recovery is required for correctness (in case of connection take-down), special attention is given to its efficient implementation using unification of the timer operation for all take-down processes taking advantage of the fact that no ordering need to be maintained across take-downs of different connections.

3. OPENET Network Model

OPENET views the network as a collection of *nodes* interconnected by high-speed *links* and also attached to external *users*. A node is comprised of two functional parts: the *Switch*, and the *Control Point* (CP). The switch contains a switching fabric and its low level control, typically implemented in hardware. The CP is responsible for all other control functions such as initializing and maintaining the switching tables and coordinating activities with other switches. The CP is typically implemented in software. CPs exchange data with their peers through designated VC termed *control VC(s).* Typically (but not always) transfer of information to non-neighboring CPs is accomplished hop-by-hop over single link control VCs.

3.1. CP Functions

The CPs perform those functions necessary to make a group of nodes operate as an integrated network. Adopting the PNNI terminology we distinguish between OPENET signaling which is the set of protocols used to setup, takedown and maintain connections and OPENET routing which is the set of messages and procedures used for computing the routes and gathering the information necessary for this computation.

3.1.1. OPENET Signaling

User Support: The CP is responsible to interpret user requests received via the User Network Interface (ATMF UNI or Q9321 [5]), and translate them into internal actions that provide the user the service it requested. OPENET also supports an extended set of functions we have found important (Section 5.2).
VC Set-Up: Once a user request is received and a plausible route is computed an attempt to establish the connection is made. A special type of VC is the point to multipoint multicast VC. Such VCs are organized in a tree structure and hence present additional complexity in its computation and in setting it up. OPENET provides several new useful setup functionalities. OPENET allows the user to specify a range of acceptable bandwidth and returns the highest the network may support. It allows the establishment of a VC from a point which is not the source (third party setup). It allows the computation and establishment of a complete multicast tree as a single operation, whereas the current ATM standard only supports adding connections one at a time. OPENET also provides new multicast connections (termed distribution trees) which allow multiple sources and destinations.
VC Take-Down. Once the VC is no longer needed (as a result of user indication or a failure), it should be taken down efficiently and orderly so that released resources may be reallocated as fast as possible.
VC Maintenance. This function typically manages changes in the connection allocated resources during the VC's lifetime, such as, an increase or decrease of the VCs bandwidth parameters.

3.1.2. OPENET Routing

Topology Update: OPENET conducts the computation of the complete route at the source. Therefore, each node maintains a global view of the complete network and this view is maintained using a link state update algorithm.
Utilization Update: To calculate efficient routes every node maintains a utilization view of the network, i.e., the level of bandwidth reservation for all QoS classes of every link in the network. OPENET efficiently updates this view using an ATM *distribution tree* described later in detail.
Route Computation: In order to decrease the setup latency, OPENET employs extensive route caching so that calls do not need to wait for a complete route computation. This also enables the CPs to utilize otherwise idle periods for the pre-computation of such routes.

3.2. Connection Types

User and control information are carried over connections. The current ATM standard recognizes two types of connections. A point to point connection (unicast) where a single source (*calling party*) exchanges information (information may flow in both directions) with a single destination (*called party*). A point to multi-point (unidirectional multicast) connection over which a single source may send the same cells to multiple destinations. Such a connection is established by maintaining a directed tree structure (termed a *destination tree*) over which cells are being sent from the root to the leaves. At every junction in the tree the switch must copy the each cell to multiple output ports and perform a VC label swap operation specific to each output port. In both connection types only a single source is multiplexed at a receiver.

In [9], another possible structure we term a *source tree* (termed sink tree there) is described. This connection carries multiple streams of cells from multiple sources to a common destination over the same tree and its VC labels. This implies that an adaptation layer that relies on the VC value only (AAL5) cannot be applied to a source tree. Therefore, the tree can be used to deliver unicell messages or support an adaptation layer that supports sub-VC multiplexing (AAL3/4).

A *distribution tree* is a multicast construct with several sources and several destinations. The number and identity of the sources and destinations does not have to be the same. A cell sent by any of the sources on the distribution tree will arrive at all the designated destinations. Similar to the destination tree case, a distribution tree may be used for unicell messages or under an adaptation scheme which supports sub-VC multiplexing. In the following we describe two approaches for establishing a distribution tree, the load-balanced tree and the pivot based approaches. These trees are used to facilitate the utilization update in OPENET. In addition they are offered as a new user service.

3.2.1. The Load-Balanced Distribution Tree

A tree spanning all of the sources and destinations is constructed such that each link of the tree is bidirectional. Thus, two VCIs are defined between any two neighboring nodes of the distribution tree–one in each direction. Cells are then broadcast on this tree, i.e., each node associates every incoming VCI of the destination tree to all the outgoing VCIs on the links of the distribution tree *except* in the return direction (i.e., excluding the link of the incoming VCI).

As an example consider the network of Figure 2. In the example the circles are nodes whose names are marked with upper-case letters, the bold lines are the links of the designated tree and the port numbers are marked with numerals (by convention the local CP is always designated as port 0 and not shown in the figure). The VC tables of all the nodes appear in Figure 3. For example, a cell injected to the network at node C (port 0) is switched to both port 4 (towards node F) with a VCI 18, and port 1 (towards node E) with a VCI 30. This cell, upon arrival at node E (on port 3) with that VCI will be forwarded appropriately towards nodes A and B and towards node's E own CP. As is evident from the example every cell will traverse every link of the tree exactly once thereby balancing the load on these links.

3.2.2. A Pivot Based Distribution Tree

In some cases, it may be beneficial to set up the distribution tree about a specially designated node called the pivot. Each source forwards all of its cells to the pivot, which then forwards these cells to each destination. Thus, the pivot-based tree is composed of a combination of a source tree (described in [9]) and a destination tree (a regular ATM multicast tree) in which the same node serves as the root node of both trees. A special mapping of the VC labels at the root switch directs the traffic multiplexed from all sources (over the source tree) to be multicast over the destination tree to all its destinations.

Consider a set of sources s_1, s_2 , . . . , s_n and a set of destinations d_1, d_2 , . . . , d_m. Choose a *pivot* switch p and construct two directed trees one consisting of the switch p along with all the s_i switches (the source tree) and the other consisting of the switch p with all the d_i switches (the destination tree). The source tree is directed towards p and the destination tree is directed away from p. The construction of the trees is done according to the normal way of VC construction. Thus, a cell entering at any of the switches s_i carrying the VCI of the source tree will end up at the pivot switch. Using the multicast capability, every

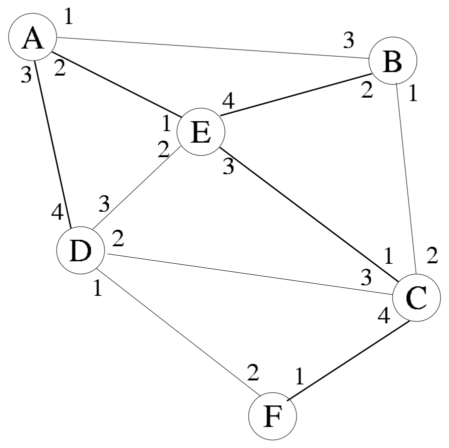

Figure 2: An example of a utilization tree.

cell entering the pivot with the VC identifier of the destination tree will be switched and distributed to all destinations d_i. By appropriately setting the VC-table at the pivot so that the source tree VCI and the destination tree VCI are linked together, the distribution tree is constructed. When the same nodes are both sources and destinations it may be simpler to construct a single tree that serves both as the source and destination trees.

A drawback of the pivot-based approach over the load-balanced tree approach is that the traffic load on the links is not completely balanced. For instance, in the example above if node C transmits a cell it will wind its way to node E (the pivot) and back towards node F thus traversing the link CE twice, once in each direction. This happens for every link shared by the source and the destination trees.

4. Topology/Utilization Update and Call Routing

Among the key OPENET novelties is the way it handles utilization update and route caching. Therefore our detailed description will focus on these two subjects first.

The topology and utilization update mechanism is designed to maintain a replicated database at all CPs so as to enable each CP to compute an effective route for new arriving calls or whenever rerouting is necessary. The database must therefore include all the relevant information for such routing and potentially for other network control functions. Examples of information items included in the database are: the set of switches, the set of communication links and the manner in which they are interconnected, the ID of CPs, link and switch capacities, link and switch current utilization levels, etc. Other information, which we will term *attributes*, can also be included. These might be information regarding reliability, cost, length or security properties of links and switches as well as user reachable addresses or domains (in the absence of other directory services [2][8]).

Input Designation		Output Designation			
P	VCI	P	VCI	P	VCI
0	25	2	23	3	24
3	18	2	23	0	24
2	21	3	24	0	24

Input Designation		Output Designation	
P	VCI	P	VCI
0	20	2	27
2	23	0	20

Input Designation		Output Designation			
P	VCI	P	VCI	P	VCI
0	13	1	30	4	18
4	31	1	30	0	18
1	22	0	18	4	18

Input Designation		Output Designation	
P	VCI	P	VCI
0	20	4	18
4	24	0	11

Input Designation		Output Designation					
P	VCI	P	VCI	P	VCI	P	VCI
0	19	1	21	3	22	4	23
3	30	1	21	0	12	4	23
4	27	1	21	3	22	0	12
1	23	0	12	3	22	4	23

Input Designation		Output Designation	
P	VCI	P	VCI
0	31	1	31
1	18	0	12

Figure 3. VC Tables for the load-balanced tree example.

In OPENET, the contents of database are logically divided into quasi-static and dynamic portions, according to the expected rate of its change. Quasi-static information is that which changes quite infrequently and is related to topology changes such as when a failure occurs or a planned network reconfiguration takes place. Dynamic items are those which change frequently, for example, with the establishment and termination of connections.

Our "optimistic approach" to network control optimizes the system for the most frequent events and trades performance for simplicity in the case of exceptional and rare events. Therefore, for performance reasons, we have separated the mechanisms for exchanging quasi-static information (*topology update*) from that of exchanging dynamic information (*utilization update*). The utilization update is optimized for both update-latency and computational overhead by assuming a fixed topology. This results in a lightweight procedure that can sustain a higher rate of updates at a low latency. For the topology information we adopt a conservative and standard mechanism that can cope with topological changes.

In the normal mode of operation, the quasi-static information is synchronized and the utilization update mechanism may assume a fixed topology. With such a setting we use a distribution tree, thereby making use of hardware based ATM multicast to accomplish the task of utilization update. This update functions properly as long as no change occurs that disrupts the distribution tree. Thus, most topological changes (those which do not impact the multicast tree) will have no effect on the utilization update mechanism and the topological update mechanism ensures that these changes are disseminated to all CPs.

When a major topological change occurs that disrupts the utilization update mechanism more massive operations must take place. The topology update mechanism is activated and disseminates the updated information to all CPs. Once this is done, the utilization update mechanism must be set-up anew, and only then can normal operation resume. To minimize the impact on the utilization information promptness, topology update messages also include utilization information for the items reported.

Since OPENET is designed as a PNNI extension, we use the standard PNNI topology update methods which is based on OSPF [11],[12],[13]. It is implemented using a reliable hop by hop flooding algorithm for the propagation of the topology database items. Because the OSPF method is well known and documented, we focus the rest of this section on the newly devised utilization update mechanism.

At both PNNI and OPENET, a single CP (termed peer group leader - PGL) is selected to perform the logical peer group functions of PNNI and other OPENET functions such as the manager of the distribution tree. The leader election process in both platforms uses the topology update procedure. Our algorithm requires much fewer messages than the original PNNI leader election algorithm. The algorithm use the Propagation of Information with Feedback (PIF) algorithm described in [16] and its details appear in [3].

4.1. Utilization Update

The utilization update is an iterative process triggered by a major change in a utilization-related database item. Clearly, the latency of this information update process reflects on the currency of the database and hence the quality and promptness of the information used for call routing. Similarly, the maximum rate of the utilization updates that the system can process has a major impact on the precision of the information maintained in the nodes as well as on the network size that can be supported. As the precision and number of links increase so does the rate of utilization update that needs to be handled by each node.

Using a flooding algorithm of the OSPF type is inefficient for utilization update purposes. A flooding algorithm implies that update messages are received in duplicates from all neighbors entailing redundant overhead. It also entails a store-and-forward operation at every hop which forces the protocol to read with no delays each received update in order to check whether forwarding is required so that information dissemination is not delayed unnecessarily. This excludes the possibility of processing a large number of duplicates in batch (since doing so will increase the latency of the update). Therefore, our architecture uses a network-wide spanning distribution tree for the task of utilization update (see Section 3.2). The same distribution tree can be used by all nodes to broadcast local changes to all others using unicell updates. Such a scheme uses native ATM cell-switching, delivers only a single copy of the information to each CP, utilizes the switch hardware multicast where possible and uses a small and fixed amount of VCIs. Since no forward operation is required at the CP it also allows batch processing of multiple utilization messages. One might add that a single network-wide spanning tree is advantageous over, say, a collection of trees for its scalability and management properties.

Utilization update is triggered whenever there has been a substantial change in the utilization from the values previously reported or if enough time has elapsed since the last update. The method to quantify a significant change is based on percentage change of the available bandwidth over a link. The rate of utilization update is expected to be orders of magnitude higher than the rate of the topology update. On the receiving side, the rate of utilization updates received can burden the receiving CP. This statement addresses the fact that handling each utilization update individually and instantaneously may result in too many system calls (interrupts) and hence an unacceptable nodal overhead. Periodic polling of the incoming cells queue (which is identified by a particular VC value) appears to be a better practice in terms of system performance.

We have quantified the major impact of utilization update mechanism on the ability of the CP to process utilization changes. When a CP processes a communication interrupt it needs to first service the interrupt routine, transfer the data to main memory, invoke basic driver processing, perform a context switching to the appropriate process (say “link state protocol”) handle the message and send a possible response. Typical numbers for this process are 9ms (Sparc 1) and 4ms (DECStation)[14]. Recall that a switch embedded CP may not be the most powerful RISC engine.This flood update processing time sets a limit to the number of flood updates that can be processed to somewhere around hundred or less (taking into account that other processes needs to be run as well).

We have simulated the update rate of the link triggering update mechanism of PNNI in a network of 60 nodes and 240 links at average loads of 0.6–1.0. Our calls were divided to 3 classes in terms of calls duration (1–15 hours, 1 minute - 1 hour, and 0.5 second to 10 seconds, assuming uniform duration distribution within the range) and bandwidth range (1–20Mbps, 64Kbps–10Mbps and 1–10Mbps, respectively).

ACR_PM	Load				
	0.6	0.7	0.8	0.9	1.0
30%	.3439	.3980	.4208	.4670	.4912
40%	.2796	.3275	.3503	.3974	.4162
50%	.2284	.2703	.2915	.3320	.3515
60%	.1879	.2243	.2442	.2810	.2984
70%	.1537	.1850	.2038	.2365	.2527
75%	.1385	.1675	.1922	.2158	.2306

Table 1: PNNI - Average updates rate, per link per second.

The classes represent backbone connections (PVCs and VPs), real-time video and audio and bursty data connection (WWW, ftp, NFS) respectively. The distribution among classes was 0.25, 0.25 and 0.5, respectively. PNNI uses an incremental reporting algorithm (using the parameter ACR_PM) which triggers a new update only if the change in the available bandwidth (ACR) since the last update is at least ACR_PM% (however not less than 3% of the raw link capacity). The results of Table 1 are in terms of average number of updates per link per second. For example for a load of 0.8 and an ACR_PM of 60% the total number of updates triggered is $0.2442 \times 240 = 58.5$. If some of the backbone switches in the network have a link degree of 8 this amounts to the reception of 438 updates/second on average. For a more accurate (but still quite coarse) ACR_PM=30% the number rises to 808. Finally, recall that the average is not the worst case load. Given the possible correlation between the updates (a large call is being setup or released via multiple links) there is a non-negligible probability (0.02) for periods of update rates which are twice or more higher (see Figure 4). Therefore, a clear conclusion is that the PNNI utilization update mechanism is very inefficient and might result in periods of excessive CP loads even under moderate network sizes and update accuracies. A clear evidence is the recommendation of the PNNI standard to set ACR_PM at the level of 75%, which results in almost no meaningful utilization information. In contrast in OPENET, utilization update is received only once at a node and can be batched at any granularity level. Therefore, no matter how many such updates are received the CP may deal with all updates at its own convenient times.

Utilization update is performed over the distribution tree structure. The elected leader of the peer group is responsible for setting up and maintaining the distribution tree. When a topological change that affects the connectivity of the distribution tree occurs (such as link or node failure), the leader will send a new *tenure* message (defined in PNNI) to all other nodes within the peer group. Every node that receives this

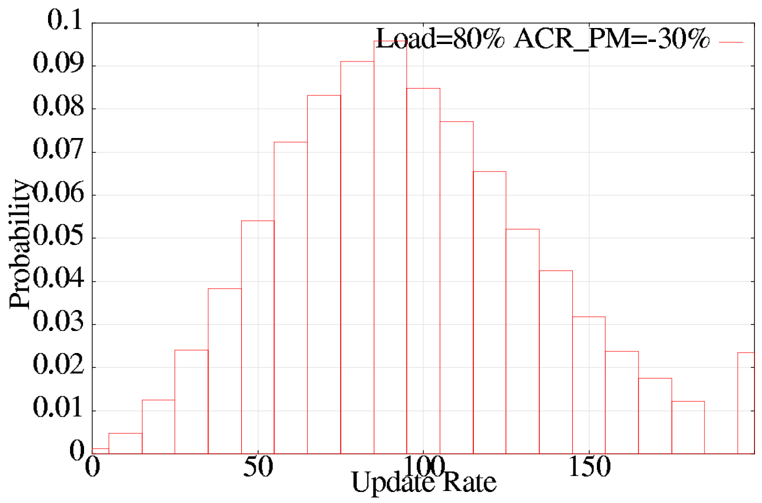

Figure 3: Utilization rate distribution.

message will locally release the current distribution tree thereby stop the forwarding of cells on this tree. Since the tenure message is sent using a reliable flooding mechanism, every node in the peer group will eventually receive this message and release the distribution tree. After the leader sent the tenure message, it will compute a new distribution tree and set it up.

4.2. Route Computation

Route computation is used in the setup process of new connections or the rerouting of existing connections. It is performed at the source CP of a connection or by another CP for third party setup. Its goal is to obtain a low cost (in terms of administrative weight) simple path (unicast) or tree (multicast) in the network which meets the bandwidth and other QoS constraints (such as delay and loss) of the connection.

A setup request for a unicast connection contains the bandwidth requirements and QoS class of the connection in both directions. The bandwidth requirements (i.e., the peak and sustainable cell rates) along with the current link state parameters (i.e, the ACR, CRM and VF) are used by the GCAC algorithm in [4] to determine whether a given link can be considered in the route computation. The QoS class determines the values of the MCTD, MCDV and MCLR0 parameters. We use the The MCDV and MCLR0 attributes of a given link to determine whether this link can be considered in the route computation. The route computation then proceeds by finding the minimum weight path (according to the AW parameter) under the constraint that the sum of the MCTDs along the path does not exceed the requested MCTD of the connection. A fast setup process is extremely important for the success of ATM networks as discussed in earlier. An on-line computation of the route upon the receipt of a setup request can extensively delay the setup process. Hence, our route computation process described below, consists of an off-line computation

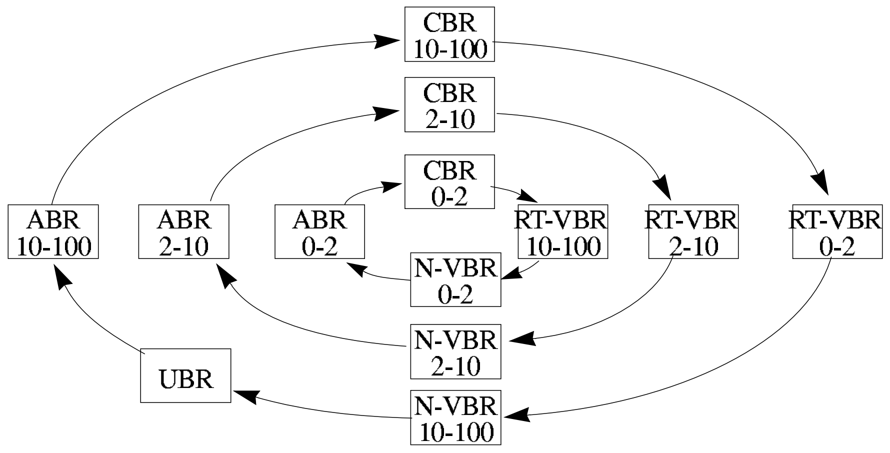

Figure 4: Route computation instances.

of routes, where the routes for different QoS classes and route capacities are pre-computed and stored in a route database. This allows for background computation of routes, using idle processor periods for route computation. Off line route calculation is particularly advantageous for a network with a high number of calls each requesting only a small portion of typical link capacities. (In a well designed network, the number of calls requesting a large portion of the capacity cannot be too large). In this case, it can be assumed that the same routes can be used multiple times as single calls do not saturate the links immediately. It might also be reasonable to use both precomputed routes for frequent small bandwidth calls as well as on demand computation for large bandwidth calls.

OPENET computes 13 different instances of routes divided according to three bandwidth ranges and the QoS class of calls as depicted in Figure 5. Computation starts generally from high-bandwidth calls (as these routes can also be used to carry low bandwidth call) to low-bandwidth ones. Arriving calls are matched to one of the route instances. The cached route is checked and validated against the latest topology and utilization data. If no valid route is found, an on-demand computation is carried on for that call.

Another aspect of the routing algorithm is the way it adapts to changes of topology and utilization. OPENET has taken the approach that a single instance of route computation will not be interrupted in the middle due to new topological changes (preventing "livelocks" as a result of excessive updates). Instead, the current computation instance will be completed and only then the new topology information will be integrated. The computation will continue retaining the same round robin order until all instances are derived using the new parameters. In general, routes for high bandwidth calls are always computed first. The latter approach guarantees that even in a frequently changing network, the routing algorithm will be able to make progress and all QoS classes will be served by it.

Our algorithm is a heuristic one based on several applications of Dijkstra's shortest path algorithm each with a different set of edge weights. Each application looks for an optimal set of paths with respect to one criterion using the others as constraints. The execution of the complete algorithm results in a set of feasible routes in terms of the delay requirements. This set of routes is stored as an ordered set of subtrees (which is a natural outcome of Dijkstra's algorithm).

Let D^{CBR} denote the delay constraint of CBR connections and let W_i and D_i denote the weight and delay of link i, respectively. Let $\overline{W}$ denote a vector whose components are W_i. Similarly denote $\overline{D}$. Denote by $\mathrm{DIJ}\,(\overline{\mathrm{X}})$ the application of Dijkstra's algorithm to our network when the link weights equal $\overline{X}$; Let $\mathrm{DIST}\,(\mathrm{T},\,\overline{\mathrm{X}})$ be a vector whose i-th component is the distance of node i from the root of tree T when $\overline{X}$ is taken as the link-weight vector of T. The algorithm uses a scalar constant $c > 1$ and proceeds as follows:

1. Compute $T^1 = \mathrm{DIJ}\,(\overline{\mathrm{W}})$, $\overline{\mathrm{W}}^1 = \mathrm{DIST}\,(\mathrm{T}^1,\,\overline{\mathrm{W}})$, $\overline{\mathrm{D}}^1 = \mathrm{DIST}\,(\mathrm{T}^1,\,\overline{\mathrm{D}})$. Remove all nodes i for which $D_i^1 > D^{\mathrm{CBR}}$.

2. Compute $T^0 = \mathrm{DIJ}\,(\overline{\mathrm{WD}})$, $\overline{\mathrm{W}}^0 = \mathrm{DIST}\,(\mathrm{T}^0,\,\overline{\mathrm{W}})$, $\overline{\mathrm{D}}^0 = \mathrm{DIST}\,(\mathrm{T}^0,\,\overline{\mathrm{D}})$. Remove from T^0 all nodes i for which $D_i^0 > D^{\mathrm{CBR}}$ (those cannot be assigned a path in this round of computation) or those for which $W_i^0 > c \times W_i^1$.

3. Define $\overline{X}(p) = \dfrac{p}{\max\limits_i \{W_i^0\}}\overline{W} + \dfrac{(1-p)}{\max_i \{D_i^0\}}\overline{D}$.

4. Perform the following for $p = 0.7,\, 0.4,\, 0.1$:

   Compute $T^p = \mathrm{DIJ}\left(\overline{X}(p)\right)$, $\overline{W}^p = \mathrm{DIST}\,(T^p,\,\overline{W})$, $\overline{D}^p = \mathrm{DIST}\,(T^1,\,\overline{D})$. Remove from T^p all nodes i for which $D_i^1 > D^{\mathrm{CBR}}$ or those for which $W_i^p > c \times W_i^1$.

All destinations not included in any of the trees are considered non-reachable under the current constraints. For routing to destination node i chooses the first path found in the trees as T^p varies from 1 down to 0.

Clearly, a better algorithm would perform step 4 by searching more values of p at a higher computation cost. For bidirectional connections where both directions use the same undirected path, we modify the above algorithm in the following manner. In step 1 we replace the link vector $\overline{W}$ by a vector whose components are the sum of the link weights in both directions. The definition of $\overline{X}(p)$ in step 3 uses the maximum value of both directions.

For ABR and UBR connections the route computation is less complex due to the fact that no additive constraints are required. In this case the shortest path algorithm with links weights equal to the administrative weights will do (after trimming those links where the loss, rate and other constraints are not met).

In the multicast case we first construct a subgraph of the network which contains only those links which can support the requested bandwidth and QoS requirements. An approximated minimum weight Steiner tree which contains the source and the destinations is then computed (see, e.g. [17]), with links weights equal to the sum of the weights in either or both directions. Then, the delay feasibility of the tree is checked, and if unfeasible we construct a shortest path tree (using the delay as the weight parameter) from the source to the destinations using a collection of individual unicast routes as explained earlier.

5. OPENET Signaling

5.1. Forwarding modes

OPENET defines low level communication mechanisms termed *forwarding modes*, by which CPs forward control and other messages to other non-neighboring CPs. The *source route* and VC *traversal* forwarding modes described in this section allow a CP of, say, switch s_1 to forward messages to the CPs of switches $s_2, \ldots, s_n$ in a given order. The forwarding modes are used by both OPENET signaling as well as other network control applications.

5.1.1. Source Route Forwarding Mode

In the source route forwarding mode a message traverses a path from CP to CP according to a route that is explicitly contained within the message [10]. The path is described by a sequence of local link/port IDs. CP along the path not only forwards the message to the next CP along the path but also performs an individual function that is also indicated in the list. Such a message is used, for example, by the VC set-up mechanism in which the typical function would be bandwidth reservation. Source route forwarding is also used for general datagram delivery between CPs, i.e., delivery of messages without connection setup.

Not all CPs along the path have to perform the same function nor must the path be a simple one. Thus, for example, one can construct a path from, say, CP_1 to CP_2 and back to CP_1 in a way that different functions or none are performed by the CPs in the forward and backward direction. For example, one can use such a message for the construction of a VC in which the initiator of the setup operation is not the source of the VC (third-party setup). The source routing mode also allows to trace back the path the arriving message took. The reverse path is accumulated by replacing the current forward path port by the backward ones at each hop. In this case, an arriving message can be responded to over the same path it has arrived on.

5.1.2. VC Traversal Forwarding Mode

In the VC traversal forwarding mode a message traverses a path hop-by-hop from one CP to another along the route of a previously established VC, i.e., the control message is forwarded from a CP to a CP along the VC route without having its cells being carried over that VC. A VC traversal message can traverse any type of VC, including multicast VCs. Since the CP keeps a version of the local VC table(s), it is possible for CPs to exchange control messages along VC path by examining the entries of its VC table and swapping the appropriate VC identifiers, emulating in software the actions performed by the switch itself.

The method of forwarding control messages from a CP to a CP along the path of an existing VC described above is referred to as the *Forward VC Traversal* or simply *VC Traversal*. VC traversal can also be carried in the opposite direction based on the output labels. This is referred to as the *Reverse VC Traversal*. The reverse VC traversal method might be less efficient since a search through an unsorted table may be necessary. The reverse VC traversal method is sometimes necessary to forward control messages along a unidirectional connection or a partially established connection (see below).

A VC traversal message can also traverse the tree of a multicast connection in a way similar to traversing the path of a unicast connection. The difference here is that each entry in the VC table of the CP may refer to more than one output port/label pair, depending on the structure of the tree. It is also possible to use the forward VC traversal mode to send control messages back to the root for an established source tree.

5.2. Connection Control and Maintenance

The main tasks associated with connection control are connection setup, connection take down and connection maintenance. The OPENET connection setup uses the source route forwarding mode and includes the update of routing tables and the bandwidth reservation along the chosen route. It utilizes end-to-end error recovery in order to lower the overhead in typical error-free cases. The connection can be either a unicast, a single source multicast, or a multi-source multicast (distribution tree). Connection clearing (take down) is used for bandwidth and label release upon the termination of calls, or upon topological outages that disconnect the connection. Connection maintenance is needed to modify the parameters of existing connections and to insure "soft state" operation of the network to release of unused resources in cases of undetected errors. Call signaling have a major impact on the overall performance of the system. OPENET uses light-weight protocols based on the optimistic approach guideline. In the following we outline the main ideas behind these operations, more details can be found in [3].

5.2.1. Unicast Connection Setup and Clearing

A unicast connection setup is initiated by a user by sending to its CP a UNI SETUP message [5] that includes the destination address, the class of service it requires, and other parameters as specified by the standard. The CP maps the request into one of the internal classes of service. In OPENET traffic intensity is represented by two values: MAX and MIN, indicating a region of acceptable values. One of the main reasons for such a setting is allowing the user to make more flexible decisions based on network state. The CP selects a feasible route for the corresponding class of service and requested bandwidth (as described in Section 4.2.) and constructs an N_SETUP message (defined by OPENET [3]), using the source route forwarding mode. The message contains, in addition to its type and route, the MIN and MAX parameters, the function to be performed by every link and *label* fields that are swapped by the corresponding CPs along the route. It also carries the relevant end-to-end parameters that were part of the user UNI SETUP.

The N_SETUP message is then forwarded *downstream* from the source CP to the destination CP. Each CP that receives the N_SETUP message checks whether the corresponding switch can accommodate the connection. If the available bandwidth (termed M) for the requested class is greater than the current MAX value, a quantity MAX of the bandwidth is reserved, and the N_SETUP message is forwarded to the next CP on the route. If M is between the MIN max MAX values, then a quantity M is reserved and N_SETUP is forwarded with the MAX value being set to M. If M is smaller than the MIN value, then the connection setup process is aborted and the connection clearing process is started from this point back to the source.

One of the functions of the set-up procedure is to set the appropriate VC identifiers. To that end the N_SETUP message carries two labels, a *forward* and a *backward* one, corresponding to the two directions of the VC. Upon receiving the message, the CP records the backward label that appears in the message and chooses one (unless one already exists) for the backward direction which it puts in the message before it is being forwarded. The forward label is also included if it is already available in the VC table (used in cases of N_SETUP retransmissions so as not to setup the same connection twice). At this point these labels are logical and are used only for VC traversal; They will become physical labels upon the receipt of the N_SETUP_ACK.

When an N_SETUP message arrives at the destination CP, it sends a UNI SETUP to the destination host and generates a N_SETUP_ACK message that is forwarded upstream back to the source CP along the same route of the established connection. The N_SETUP_ACK message, which is of the forward VC traversal type, contains the final MAX value contained in the N_SETUP message just arrived. The message also completes the exchange of VC identifiers for the forward direction (source to destination).

Upon receiving an N_SETUP_ACK message, each CP along the reverse path updates the bandwidth reserved for the connection (its original reservation was at least that value), updates the label fields and records it in the switch and sends the message upstream. When the message arrives at the source CP, the connection is now ready. However, because the destination may require some time before agreeing to accept the connection (based on UNI specifications), data cannot yet flow through the connection. If N_SETUP_ACK (or CONNECT) are not received in the source after a specified time-out, it will re-send the N_SETUP message again.

When the destination host responds with a UNI CONNECT to the destination CP, the CP sends N_CONNECT upstream using the forward VC traversal mode. When the N_CONNECT reaches the source CP it is converted back to a UNI CONNECT and delivered to the source host. The message N_CONNECT_ACK is sent back to the destination host using the forward VC traversal mode for error recovery purpose. Both N_CONNECT and N_CONNECT_ACK are of the forward VC traversal type and do not impose any additional processing at the intermediate nodes. When the N_CONNECT_ACK reaches the destination CP it delivers a UNI CONNECT ACKNOWLEDGE message to the destination host. The setup procedure is now complete.

In normal operation of the network, clearing (take down) operations are needed to release reserved bandwidth upon the users requests to terminate connections. In addition, clearing operations are needed to release reserved resources when failures cause the disconnection of the source from its destinations. The clearing protocol is responsible for returning the used VC labels as well as bandwidth to the unused pool.

Ours is a 'one pass release protocol', based on the assumption that errors occur very infrequently. The clearing protocol is initiated by a CP (the 'originating CP') based on a user's request or some error condition. The originating CP generates and forwards an N_RELEASE message (defined by OPENET) which is of the VC traversal type. Each CP that receives the message releases the corresponding bandwidth at the appropriate switch, erases the corresponding entry from the VC table and forwards the message. Only a single N_RELEASE message will be transmitted from the originating CP to the other end of the connection.

5.2.2. Unicast Connection Maintenance

OPENET supports changing the reserved bandwidth in a connection. The N_CHANGE message is used to change the reserved bandwidth along a path when so is desired by the source user. It is always possible to lower a bandwidth requirement but the increase in bandwidth is on an availability basis. The N_CHANGE message (using VC traversal) contains the MAX and MIN bandwidth desired for this connection and a refresh period value. The N_CHANGE message is acknowledged by the N_CHANGE_ACK message, which contains the final negotiated value and is also of the forward VC traversal type. The N_CHANGE message can be used in combination with the N_RELEASE message to provide highly reliable mechanism for the return of critical resources used by large bandwidth connections.

Connection maintenance is also important as a "garbage collection" mechanism that prevents deadlock of resources following undetected messages and memory errors. These events are expected to be rare, and therefore the connection maintenance we use is based on the source CPs sending at a very low rate periodic N_CHANGE messages in each direction of every established connection with no change in the bandwidth parameters ('refresh messages'). These refresh messages allow the intermediate CPs along the connection to slowly age the connection and eventually take it down if not refreshed within the connection refresh period. The refresh period is not uniform: high bandwidth connections may use more frequent refresh messages than light connections.

We expect that typical connections may have multiple N_CHANGE exchanges between their setup and termination. Therefore, in many cases this procedure will dominate the network control signaling. Therefore, it is particularly important to design these messages using light weight protocol and a hardware implementation hooks. The N_CHANGE and N_CHANGE_ACK messages are simple and contained within a single cell and can be easily mapped into hardware implementation.

5.2.3. Multicast Connection Setup

A multicast connection setup is initiated by a user when it needs to communicate with several other users in the network. The host sends its CP a request that contains the destination addresses, rate and the QoS it requires. As in the unicast case, the CP maps this to a network class of service along with a MIN-MAX representation of traffic intensity. The CP then computes low cost feasible routes for the corresponding constraints. These routes form a tree rooted at the source host and each of its leaves is a destination hosts (or destination switch port).

OPENET supports two ways to set up a multicast VC. The first method, called *complete setup*, is based on a single setup message that traverses the whole tree and creates a multicast tree using a single message. The second, termed *incremental setup*, is based on several unicast like setup messages (that may partially overlap). The incremental method allows the root to add destinations to the multicast tree one at a time as in UNI 4.0.

In the complete setup case, the source CP computes a traversal path of the tree. This path starts and ends at the source CP, and traverses each link of the tree exactly twice. The setup process is achieved by an N_SETUP message of the source traversal route type which include the traversal list, the function to be performed by every link, the label field and the MIN and MAX values, much like the unicast N_SETUP message. For a complete list of the possible functions in the multicast case see [3].

A CP that receives the N_SETUP message for the first time (no backward labels assigned yet) checks if the requested service can be supported. As in the unicast case, If the resources cannot be secured an N_RELEASE message is sent back towards the source CP. In other cases the MAX field of the setup is updated, the bandwidth reserved. If the entry in the VC table does not exist yet the VC table is updated accordingly and a backward label is chosen and forwarded (a CP is always visited for the first time in the down-stream direction). If this message also includes a forward label (in addition to a new backward one - indicating first upstream arrival) the CP creates a new pair of (associated) entries in both directions that corresponds to this VC. If this is a branching point the CP then allocates the backward label field before forwarding the N_SETUP message while associating it with the same VC. Any time a N_SETUP message carries a backward label that is already known at that CP it is interpreted as a retransmission. Note that these label exchanges always properly define the associations between the label entries and informs the CP about whether the setup is new or a retransmission.

When the N_SETUP message completes the traversal back at the source CP its MAX value is the final acceptable value at all nodes in the multicast tree. The source CP issues an N_CHANGE message to notify all CPs of the final accepted bandwidth using the VC traversal mode. Each CP then adjusts the reservation to the final accepted value. The N_CHANGE message is acknowledged by all leaf CPs with the N_CHANGE_ACK message, which confirms and completes the multicast connection setup.

5.2.4. The Incremental Multicast Setup

Based on the routes it determined, the host CP computes a traversal of the tree by several (partially overlapping) linear paths. Each of these paths ends at one of the destination CPs, and traverses part of the tree from the root down to a leaf. Current UNI signaling for multicast connection require the addition of end-users one at a time using a separate ADD PARTY message for each one. Note that it is possible to perform the incremental setup either synchronously or asynchronously. In the synchronous approach, a setup message must be acknowledged by the destination before the next one is sent. In the asynchronous approach, the setup messages can be transmitted in parallel without having to wait for the acknowledgment of other setup messages. To result in a single multicast connection the setup messages must be identified with the set of entries in the VC table corresponding to the multicast connection. This is done with the forward and backward label fields in the N_SETUP message similar to the complete setup case. Generally, the setup operation is similar to the unicast setup where the overlapping parts are considered as retransmissions and reservation is only conducted once. Details and an example are provided in [3].

5.2.5. Multicast Connection Clearing

Unlike unicast connection, a multicast connection may be partially taken down, for example, when one end-user decides to remove itself from the multicast or when a failure caused the tree to be partitioned such that a part of it remains operational. There are two basic clearing operations. A complete take down of the entire connection is termed *connection release*. A partial take-down of a connection is referred to as a *user drop* operation, which can be a root-originated or a leaf-originated request. It is required (at both UNI and OPENET) that the originating user stays connected at all time.

Releasing a multicast connection is initiated by the source using the N_RELEASE message as in a unicast connection. That is, the N_RELEASE message traverses the multicast tree using the VC traversal mode and releases the bandwidth and labels of the connection. Drop requests are handled in the opposite direction, that is, bandwidth is release step by step from the leaf being dropped until the first junction node. According to this process, the CP sends an N_RELEASE message on the upstream direction of the multicast tree using the VC traversal mode. This message will release the bandwidth and the labels up to the corresponding first branching node. The branching node, upon receipt of this message will send an N_DROP_REPORT message (of the VC traversal type) to the source.

Another common operation is a source-triggered dropping of a party. Because it is more convenient to conduct the dropping of a party from the dropped party backward, this operation is done in two steps. The source node sends an N_DROP_REQUEST message which contains a list of end point IDs of the parties to be dropped. This message is of the VC traversal type, and it traverses the multicast connection and arrives at all destination CPs. All destination nodes whose end point reference is included in this message will start a drop party process as described before. As a result, an N_DROP_REPORT message will be delivered to the source for each dropped party, completing the drop operation.

6. Summary

The OPENET architecture offers a performance oriented and open infrastructure for building large and efficient ATM networks. It leverages the PNNI standard as much as possible without sacrificing these essentials.

The performance enhancements make use of several novel mechanisms which address the key performance bottlenecks of network control. OPENET uses hardware base multi-point to multi-point unicast to for a major facilitation in the dissemination of utilization update. It employs off-line route computation to reduce routing latency. It utilizes light-weight hardware compatible signaling with end-to-end error recovery to both reduce signaling latency and to reduce overall signaling computation load.

OPENET offers standard APIs and access to its routing modes and information to future network control applications making these applications switch independent.

References

[1] R. Handle, M. Huber and S. Schroeder, "ATM Networks, Concepts, Protocols, Applications," second edition, Addison-Wesley, 1994.

[2] I. Cidon, I. Gopal, M. Kaplan and S. Kutten, "A Distributed Control Architecture of High-Speed Networks," *IEEE Trans. on Communications*, Vol. 43, No. 5, pp. 1950–1960, May 1995.

[3] I. Cidon, T. Hsiao, P. Jujjavarapu, A. Khamisy, A. Parekh, R. Rom and M. Sidi, "The OPENET Architecture," SUN Microsystems Laboratories report, SMLI TR–95-37, December 1995.

[4] PNNI SWG, "PNNI Draft Specification" ATM Forum contribution 94–0471R7.

[5] ATM Forum, "ATM User-Network Interface Specification", version 3.0, Prentice Hall 1993.

[6] I. Cidon, I. Gopal and R. Guerin, "Bandwidth Management and Congestion Control in plaNET," *IEEE Communication Magazine*, Vol. 29, No. 10, pp. 54–63, October 1991.

[7] I. Cidon, I. Gopal, P. Gopal, R. Guerin, J. Janniello and M. Kaplan, "The plaNET/ORBIT High Speed Network," *Journal of High Speed Networks*, Vol. 2, No. 3, pp. 171–208, 1933.

[8] A. Baratz, J. Gray, P. Green, J. Jaffe and D. Pozefsk, "SNA Networks of Small Systems," *IEEE Journal on Selected Areas in Communications*, Vol. SAC–3, No. 3, pp. 416–426, May 1985

[9] R. Cohen, B. Patel, F. Schaffa and M. Wiilebeek-LeMair, "The Sink Tree Paradigm: Connectionless Traffic Support on ATM LANs," *Proceedings of IEEE INFOCOM'94*, pp. 821–828, Toronto, Canada, June 1994.

[10] I. Cidon and I.S. Gopal, "PARIS: An Approach to Integrated High-Speed Private Networks," *International Journal of Digital and Analog Cabled Systems*, Vol. 1, No. 2, pp. 77–86, April-June 1988.

[11] J. Moy, "OSPF Version 2", Network Working Group, Internet Draft, September 1993

[12] J. McQuillan, I. Richer and E. Rosen, "The New Routing Algorithm for the Arpanet," *IEEE Trans. on Communications*, Vol. 18, No. 5, pp. 711–719, May 1980.

[13] D. Bertsekas and R. Gallager, "Data Networks," second edition, Prentice Hall, 1992.

[14] R. Bettati, D. Ferrari, A. Gupta, W. Heffner, W. Howe, M. Moran, Q. Nguyen and R. Yavatkar, "Connection establishment for Multi-Party Real-Time Communication," in *Proceedings of Fifth International Workshop on Network and Operating Systems Support for Distributed Audio and Video*, Durham, NH, April 1994

[15] B. Awerbuch, I. Cidon and S. Kutten, "Communication-Optimal Maintenance of Replicated Information," *Proceedings of 31st Ann. Symposium on Foundation of Computer Science*, (St. Louis, MO), October 1990, pp. 492–502.

[16] A. Segall, "Distributed Network Protocols," *IEEE Trans. on Information Theory*, Vol. IT–29, No. 1, January 1983.

[17] F. K. Hwang and D. S. Richards, "Steiner Tree Problems," *Networks*, Vol. 22, pp. 55–89, 1992.