An Efficient Signaling Structure for ATM Networks

WUCS-95-09

Dakang Wu

Department of Computer Science
Washington University
dw@dworkin.wustl.edu

April 25, 1995

Abstract

As ATM becomes widely accepted as the communication standard for high speed networks, the signaling system structure and protocols that support ATM become more and more important. To support existing, future and unknown applications, the signaling system has to be very flexible and efficient. In this paper we define the signaling problem, present several possible signaling system structures, compare the advantages and disadvantages of these systems, and then we propose a new signaling system structure. The fundamental idea of the new signaling system is the logical separation of the signaling system structure from the underlying communication network, even though they may be built on the same physical network. The proposed signaling system structure shows very promising performance in terms of signaling latency, scalability, and reliability.

1 ATM Network and Signaling Problem

In an ATM network, user data are divided into small fixed size packets, called cells. Each cell carries, in its header, a label. At each switch in the network, the incoming cell label is used as an index to access a table which translates the label into an output port number and the label for the next hop. Since all the table look-up and transmission are performed by the hardware, cells can travel very quickly through the network. This basic scheme provides great flexibility to the communication system. Cells from different sources can be interleaved to share a physical link capacity so that the link bandwidth can be efficiently used. Since it is not difficult for the hardware to make multiple copies, multicast can be naturally and efficiently supported. ATM treats different media, such as voice, video and data, in exactly the same way, so that the communications for different media can be integrated on the same ATM network.

Since ATM is basically connection-oriented, before the data communication can take place, the tables at switches have to be set up. It is the signaling system's job to maintain these tables at switches. After a connection has been accepted by the system, some *quality of services* (QOS) should be guaranteed. To guarantee the *QOS*, resource reservation is necessary. The signaling system is responsible for managing the resources, such as bandwidth and table entries, on each link to maximize the network performance. Since the telecommunication system may involve hundreds or even thousands of switches, it is not practical for all of them to be controlled by a single controller. In our model, one or more interconnected switches are grouped to form a *node* which is under the control of a single *control processor(CP)*, which could be a general purpose computer. The control processor manages the resources of the underlying switches. To the outside world, a node is a virtual switch with information storing and processing capabilities. Nodes have to communicate in order to establish and maintain connections. Briefly, signaling can be described as the "Exchange of information specifically concerned with the establishment and control of connections, and with network management, in a telecommunication network"[**?**]. These management tasks are performed at the nodes. To design a signaling system for an ATM network, we have to address the following issues: how to organize the nodes; how to distribute the information among the nodes; how the nodes pass information and what are the protocols the nodes must follow; how efficient are the protocols; how to deal with failures.

In the rest of the paper, we consider these issues. In this section, we will formally define the multipoint signaling problem and we will give some measures to evaluate the goodness of a signaling system. Section 2 briefly introduces several signaling system designs and compares their performance in terms of the measures given in section 1. Section 3 proposes a Virtually Hierarchical Signaling System(VHSS) which is very simple and performs well in terms of almost all the important measurements. The fundamental concept is logical separation of the signaling network from the underlying communication network. Like *Signaling System No.7 (SS7)* [**?**], the signaling network is independent of the communication network. Unlike SS7, our signaling network is built on the same ATM network taking advantage of the flexibility ATM provides. A virtually hierarchical signaling system does not require that the network nodes and links be hierarchically organized. A hierarchical signaling system is built on the top of an arbitrary topology network. Section 4 concludes the paper and raises some issues for further research.

1.1 Signaling Problem

A communication network that supports multi-point connections can be modeled as an undirected graph [1] $G = (V, E)$. Each link (u, v) has an integral capacity $\gamma(u, v)$. We assume that each vertex of the graph has an identifier that uniquely distinguishes itself from other vertices. A *connection request* is a tuple $[c, T, r, w]$, where c is a *connection identifier* that uniquely identifies a connection in the network, $T \subseteq V$ is a set of *terminals*, $r \in T$ is a distinguished terminal, called the root of the connection and w is the resource requirement. For simplicity, in this paper, we consider bandwidth as the only resource managed so that we use the word bandwidth and resource interchangeably.

Let $H = (W, F)$ be a subtree of G. We say that H is a *connection graph* that implements a connection request $[c, T, r, w]$ iff for any u, $v \in T$, there exists, in H, a path from u to v, and no subgraph of H has this property.

A *connection descriptor* is a pair (q, H) where q is a connection request and H is a connection graph that implements q. For a set of connection descriptors C and any link $(u, v) \in E$, define the relative load on edge (u, v) imposed by C to be

$$\lambda_C(u, v) = \sum_{\substack{([c,T,r,w],\, H=(W,F)) \in C \\ \text{Such that } (u,v) \in F}} w \;/\; \gamma(u, v)$$

We say that C is feasible if for all edges (u, v), $\lambda_C(u, v) \leq 1$.

A *Signaling system* is said to be *incremental* if end-points can be added or removed at any time and no rerouting of existing connections is allowed. Since most multipoint applications need to maintain connections dynamically, we only consider algorithms for an incremental system.

[1]More generally, we can model the network as a directed graph. To use a directed graph model, we have to distinguish sources from sinks. For simplicity, in this paper, we use the undirected model.

A *Signaling System* dynamically maintains a feasible set C of connection descriptors for a network G under the following operations.

$create_x(r, w)$ adds a connection descriptor $([c, \{r\}, r, w], H = (\{r\}, \{\})]$ to C where c is a new connection identifier.

$destroy_x(c)$ removes the connection with identifier c from C and releases all the resources taken by connection c. Only the root of a connection is permitted to issue this request.

$join_x(c, u)$ adds a new terminal u to a connection identified by c. When successful, this operation replaces the old connection descriptor $D = (q, H)$ with a new connection descriptor $D' = (q', H')$ where q' is equal to q except with u added to the terminal set, H' implements q', and H is a subgraph of H'. An unsuccessful *join* operation returns nil and no resources will be allocated.

$drop_x(c, u)$ removes an end-point u from the connection indicated by c and releases the resources related to the end-point u. This operation replaces the old connection descriptor $D = (q, H)$ with a new connection descriptor $D' = (q', H')$ where q' is equal to q except with u removed from the terminal set, H' implements q', and H' is a subgraph of H.

In each of the above operations, the subscript $x \in V$ is called the *invocation point* indicating the location at which the operation was requested and at which a response is expected. To simplify the algorithms, in the following discussion, we assume that the invocation point is the node that starts the operation. For example, the invocation point of a *create* operation is the root of the connection, and the invocation point of a *join* is the node that is to be added to the connection. It is not difficult to pass the request and response between the invocation point and the node that starts the operation.

Notice that only *join* requests really need to allocate resources. Other operations either only release resources (e.g. *drop* and *destroy*) or only record some information (e.g. *create*). A *create* request does need to allocate bandwidth on the access link. Since we are considering the problem of network signaling, we assume that only when the access link is availablecan the *create* request reach the network. From the resource management point of view, there is no reason to reject *drop*, *destroy* or *create* requests. So we can focus most of our attention on processing of *join* requests.

One constraint of the algorithm is that it must be on-line, in the sense that the decision about routing or rejecting a *join* request has to be made without any knowledge of future requests.

The goal of a signaling system design is to design both the architecture and algorithms of the signaling system so that it maintains a feasible set of connection descriptors efficiently.

1.2 Correctness and Performance Measures

To evaluate a system, some performance measures must be provided. In this section, we discuss some of the design issues and give some quantitative measurements so that we can evaluate a signaling system.

1.2.1 Correctness

The signaling system has to solve the signaling problem correctly. Correctness can be divided into two parts: safety and liveness. Safety conditions state that the result of all operations maintains or implies a global feasible set of connection descriptors as defined in section 1.1. The safety conditions do not always guarantee a useful system. A signaling system that responds to any request with "No" and does not allocate any resources is a safe system. To exclude such a trivial design, the liveness conditions require that the system do something useful. One example of a liveness condition is "whenever the system has enough bandwidth (resource), a request should be satisfied." Unfortunately, since operations may be initiated at different nodes concurrently with conflicting bandwidth requirements, this condition can be difficult to satisfy. Another way to define the liveness is that no live-lock exists in the system. In our presentation of the VHSS algorithm in section 3, we will prove the correctness of the algorithms.

1.2.2 Efficiency of The Signaling System

As for most distributed systems, time complexity and message complexity are the measures used to evaluate the efficiency of a system. In the theoretical analysis, we assume message transmission can occur instantly. Processing a message at a CP takes one unit of time. These assumptions are justified as follows.

Cells of a signaling message are transmitted on a predefined signaling connection. Compared with software message processing, hardware transmission takes very little time. Lower level network interfaces take care of segmentation and reassembly. To the CP, it sees a whole message instead of multiple cells. With more and more built-in resource checking and reservation functions to serve increasing number of service requirements, together with interrupt and context-switching incurred with each message, the time to process a message dominates the time consumed in the entire message processing.

To simplify the analysis of time complexity, we define a *single request response time* to be the number of time units that elapse from sending of an operation request to receiving the response with no other message processing in the whole system.

1.2.3 Scalability

Since telecommunication systems can become very large, scalability is a big concern. We say a system is scalable if the worst case response time increases at most logarithmically with increasing network size. Since each signaling connection at a signaling point, defined in next section, imposes some computational load on the signaling point, we require that the number of signaling connections at any signaling point should not grow with the network size. Routing is one of the biggest computational jobs in the signaling system. We say a signaling system is not scalable if the routing algorithm has to keep track of the network status for all the nodes in the network.

2 Signaling System Design

We define a *Signaling Point* (SP) to be any entity in the network that can generate and process signaling messages. A CP of a node is an SP. Notice that an operation request is initiated by the user who is not part of our network signaling system model. To unify the processing, we assume that user requests come from SPs which are outside of the signaling network through signaling messages. The results of requests are sent to the user through response messages delivered to the SPs who sent the requests.

A signaling system consists of two parts: the signaling network and the signaling protocol. A signaling network defines the topological connection of SPs, which can be different from the topology of the network it controls. For example Signaling System No. 7, which has been widely deployed, is a separate signaling network designed to control the underlying telephone network and provide data communication services. With ATM, people can setup signaling connections freely.

Two SPs that are connected by a signaling connection are considered signaling neighbors even though they may not be connected by a physical link. In this way, the signaling network can be configured with great flexibility.

Like any distributed protocol, a signaling protocol is composed of three parts: a data store that records the current state of the system; message types and formats that define the information passed among signaling points; the signaling algorithms that solve the signaling problem based on the system state and the message processed.

To decide to accept and route a *join* request or to reject it, two types of information are necessary: network status information which includes network topological information and network load information; and connection information which includes the resource requirement for each connection and the layout of the connection graphs. If these two pieces of information are available, a signaling system can make the admission and routing decision easily. Unfortunately, these two pieces of information are not always available. One of the signaling system's job is to get this information or to use incomplete information together with heuristics.

We assume that a connection identifier contains the address of the root of the connection. This root address information can be used in our routing algorithm to find a next hop towards the root. The root address information can be retrieved through a mapping $root(cid)$ where cid is the connection identifier. We assume that signaling links preserve the message order. We assume that, at each SP, there exists a resource manager which records the link capacity and usage. The resource manager provides several SP-wide accessible functions that are used to manage the resources.

- $findPath(bandwidth, sp1, s)$ finds a path from the signaling point $sp1$ to any element in a set of signaling points s with the required bandwidth. Some shortest path algorithm and cost metric can be built into this algorithm. For now we just assume that it finds a path that satisfies the load constraint. If such a path exists, the path, a sequence of SPs that control the connected segments of the network, is returned. Otherwise nil will be the return value.

- $reserve(bandwidth, path)$ reserves the *bandwidth* on all the links of the given *path*.

- $allocate(bandwidth, path)$ initializes switch tables to setup a connection with the reserved *bandwidth* on all the links of the *path*.

- $release(bandwidth, graph)$ releases the bandwidth on all the links specified in *graph* and modifies the switch tables to release the bandwidth on all the links specified in *graph*.

In this section, we give several signaling system designs and list the advantages and disadvantages of each of the designs. At the end, we give a performance comparison of the systems discussed.

2.1 Centralized Signaling System

A centralized signaling system is the simplest to design. Figure **??** shows such a system. Every user has a signaling connection to a central control processor. Every user connection request is forwarded to the central *Control Processor*, the CP. Since the CP manages all the resources and knows all the connection status, it can make routing decisions easily.

A *connection* is a data structure that stores the connection information. A *new* operation creates a new connection data object with the root as the only vertex in the connection graph. A *delete* operation removes a connection data object from the system. Assume that there exists a global function $getConnection(cid)$ which returns a connection data object if the connection does exist. Otherwise a nil is the return value. A connection object contains a graph data structure that stores the connection graph. An access function $updateGraph(cid, updateCommand, path)$ is called to update the connection graph. If the value of *updateCommand* is ADD, a path specified by the *path* parameter is added into the connection graph. If the value of *updateComand* is REMOVE, the path is removed from the connection graph. A path is a data structure that contains a list of SPs. If r is a SP in a *path*, then $L(path, r)$ returns the left link of r in the *path* or nil if r is the leftmost SP in the *path*. Similarly $R(path, r)$ returns the right link of r. A function *findPathToBranch(sp)* returns a path in the connection graph from the *sp* to the first branching point of the connection tree. A general routine $send(destination, message)$ sends the *message* to the *destination*. The connection data structure is defined in Figure **??**. Figure **??** shows the algorithm at the CP that implements the centralized signaling system.

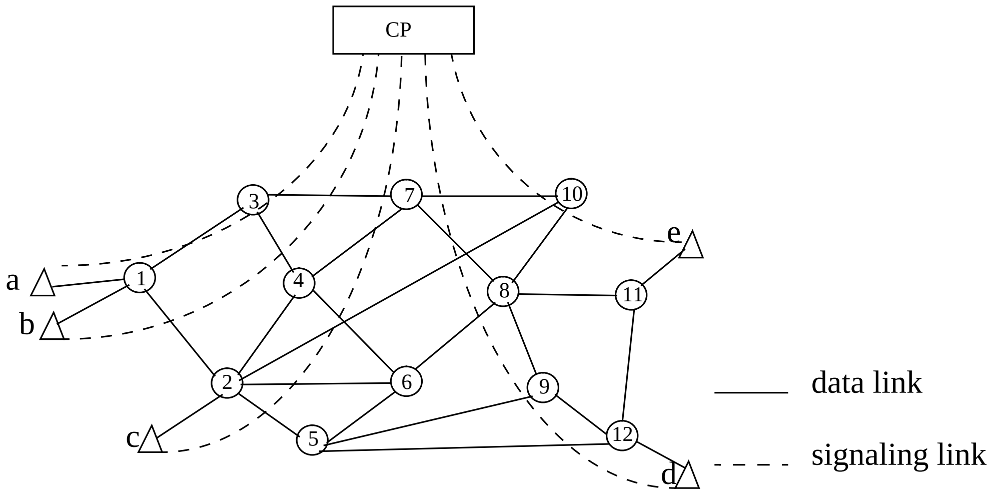


Figure 1: A Centralized Signaling System

We present the algorithms in an event processing fashion. An event occurs when an operation request message (*create, destroy, join, drop*) is received. Messages are queued. Only the one at the head of the queue causes the corresponding event to occur. Every event processing routine executes as an atomic operation.

When a *join* request comes, the CP calls the routine *findPath* to find a path to route the request from the invocation point to the connection tree. If such a path exists, the path is added to the

```
connection {
    connectionId:        cid;
    bandwidth:           bw;
    connectionGraph:     graph;
    void                 updateGraph(ConnectionId, {ADD | REMOVE}, path);
    Path                 findPathToBranch(sp);
    /* returns a path from the sp to the first branching point */
}
```

Figure 2: Connection Data Structure

```
• create(cid, bw, sp)
  new con(cid, bw)
  /* establish a connection structure with
     root(cid) as the only vertex in its graph*/
  send(sp, createResp(cid, ACK));

• join(cid,bw,sp)
  /* sp indicates the sender of the request */
  con := getConnection(cid);
  path: = findPath(bw,sp,con.graph(V))
  if (path = nil)
     send(sp, joinResp(cid, NACK))
  else
     allocate(bw, path)
     con.updateGraph(cid, ADD, path)
     send(sp, joinResp(cid, ACK));
  endif

• drop(cid,sp)
  con := getConnection(cid);
  if (con $\neq$ nil)
     path := con.findPathToBranch(sp);
     updateGraph(cid, REMOVE, path);
     release(con.bw, path);
  endif
  send(sp, dropResp(cid, ACK));

• destroy(cid)
  con := connection(cid);
  if (con $\neq$ nil)
     release(con.bw, con.graph);
     delete con;
  endif
  send(sp, destroyResp(cid, ACK));
```

Figure 3: Algorithms for Central Controlled System

connection graph and bandwidth is allocated along the path for the request. Since the capacity constraint is observed by *findPath*, no overload of links may occur. The connection graph gets updated any time a *join* is successful or a *drop* is processed, so that the connection graph of every connection is consistent with our definition of the connection graph in section 1.1. The safety condition can be easily proved by induction on the number of operations. It is easy to design *findPath* to always find a path if such a path exists (use a modified version of Dijkstra's shortest path algorithm that consider only the links with the required bandwidth for example). If *findPath* satisfies this property, the liveness condition "A *join* request will be successful whenever the network has enough bandwidth to support it" can be guaranteed.

The advantages of a centrally controlled signaling system are: it is easy to design the protocol and make it correct; there is little communication overhead.

Disadvantages of the centralized design are obvious: it is not scalable since the central controller has to maintain the whole network; the requests are processed one by one which eliminates any possibility to process requests concurrently; if the CP fails, so does the whole system.

Before we switch to other signaling system designs, it is interesting to estimate how far a centralized signaling system can go. Let d be the average path length of an operation. For each switch on the path and the user at the end of the path, the central controller has to send a request and receive a response, (to set up the switch tables and to communicate with the user), so that $2*d$ messages are needed to process one operation. Let m be the number of messages that a computer can process each second. Let n be the number of users in the system and s be the average number of operations initiated by a user in a minute during the busiest hour. Then the number of users the centralized system can support is given by following formula:

$$n \leq \frac{60 \times m}{2 \times d \times s}$$

Assume that $m = 2000$, $d = 6$, and $s = 1$. The number of users that can be supported by such a centralized system can be as much as 10000. While this calculation is somewhat simplified, it seems likely that networks with a few thousand users could be supported efficiently. This calculation shows that for a medium size network, a centralized signaling system could be a good choice.

2.2 Layered Signaling system

One way to distribute the computational tasks among CPs is to layer the network into small slices. In a layered signaling system, multiple CPs are used, each manages part of the resources. Figure ?? shows a two-layer signaling system. The bandwidth of each link (u, v) is divided into two. For example, link B-D is a 155M link. CP_a manages 80M and CP_b manages the remaining 75M. When the user starts a *create* request, (s)he has to decide which layer (s)he wants to use. If the connection is created in layer A, all the subsequent requests for that connection are routed to CP_a and use only the bandwidth managed by CP_a. In a layered system, a physical ATM network becomes multiple independent networks with smaller capacities.

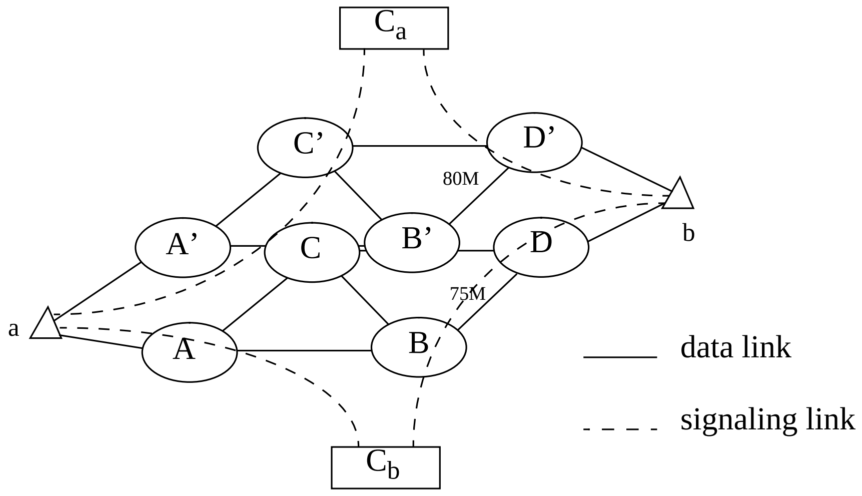


Figure 4: A Layered Signaling System

Layered design retains the simplicity of the centralized system and provides some degree of concurrency and reliability. One disadvantage is that it may unnecessarily block some requests. For example, if a *join* request comes with 20M bandwidth requirement to layer A, The residual capacity on link l is 20M and CP_a and CP_b each controls only 10M. The request has to be rejected even though the link can support that request in a centrally controlled system. Some protocols can be designed among CPs to negotiate and adjust the bandwidth managed by CPs, but that will make the system more complicated especially when the number of layers increases. A layered system is essentially a combination of centralized systems. So it suffers the same scalability problem as a centralized system. A layered system may work well when the managed links are high capacity multiple trunks. Since the bandwidth requirement of each individual request is relatively small comparing with the link capacity, fragmentation is not a serious problem.

2.3 Completely Distributed Signaling System

Both network information and connection information can be distributed in the network. In a completely distributed signaling system, every node (a virtual switch) manages the bandwidth of the links that are adjacent or internal to that node. CPs do not have global knowledge about the network status, nor do they have global knowledge about connections. They retrieve the necessary knowledge by message-passing.

When a *join* request comes to a node, the node checks if the connection exists at the node. If the connection does not exist, the request will be forwarded to the next node towards the root of the connection based on the routing table at the node. This forwarding continues until it touches a node where the connection exists or there is no link that can support the request. A response message will travel backwards to the origin of the request.

The advantages of such a system are: requests can be processed concurrently; there is no computational bottle neck as in the centrally controlled case; the number of signaling connections at each node equals the number of its external links which is much smaller than the network size; and a single node or link failure only affects the connections routed through the node or link.

The disadvantages are: messages go hop by hop so that the *single request response time* is $\theta(d)$ where d [1] is the network diameter. Efficiency of the signaling system depends greatly on the routing algorithm. If the routing algorithm does not have enough global network status knowledge, it could be very inefficient.

2.4 Traditional Hierarchical Network and Signaling

In a traditional hierarchical network like the traditional telephone network [**?**], user addresses are organized hierarchically. Each signaling point has its management domain. The domain of a higher level signaling point is the union of the domains of its *signaling children*'s, the next lower level signaling points. When a connection request comes with the user address out of its domain, the signaling point passes the request to its *signaling parent*, the next higher level signaling point. When the connection has been set up, all the data traffic goes through the same hierarchy as the control signals go.

The advantages of this design are: the number of signaling connections at a node equal the number of its signaling children plus one, which counts the connection to the signaling parent; if the branching factor is a constant B, then the single response time is $O(min\{log_B n, d\})$ where n is the network size; it is easy to design the signaling algorithm for such a network since there exists exactly one path for each pair of nodes.

The disadvantages are: it is too restrictive for the communication network so that the resources will be wasted on the way up and down the hierarchy; and some higher level signaling points may become bottlenecks for signaling processing as well as bottlenecks for resources.

2.5 Performance Comparison

Table **??** compares the performance of the signaling system designs discussed where n is the network size, L is the number of layers in a layered system, d is the diameter of the network, and B is the maximum branch factor in a hierarchical network. From the table we can see that both centralized and layered systems are not scalable since the number of signaling connections is n, as well as that their routing algorithms have to collect network status information for the whole network. Their performance could be acceptable for medium size networks. The distributed approach also suffers from the scalability problem because the single request response time is $O(d)$. The traditional hierarchical signaling system scales well, but it mixes the signaling system and the data communication system. All the data traffic has to go up through the hierarchy and then go down creating unnecessary traffic and cause possible high level bottlenecks.

[1]Assume that the routing algorithm at each node can find a neighbor which is at least one hop closer to the root of the connection than the current node is

System Type	Single Request Response Time	Number of Signaling Connections	Number of Users can be supported
Centralized	$O(1)$	n	$\leq \frac{60\times m}{2\times d\times s}$
Layered	$O(1)$	n	$\leq L \times \left(\frac{60\times m}{2\times d\times s}\right)$
Distributed	$O(d)$	$deg(v)$	any practical number
Hierarchical	$O(log_B(n))$	$O(B)$	any practical number

Figure 5: Signaling System Performance Comparison

3 Virtually Hierarchical Signaling System

3.1 Virtually Hierarchical System

The main disadvantage of a traditional hierarchical network is that it restricts the network topology to be hierarchical which is not desirable in general networks. We don't want to put any restriction on the topology of the communication network. On the other hand, hierarchies are natural in the communication world. General communication patterns show great locality geographically and organizationally. There are much more local calls than long distance calls, more inter-state calls than international calls. There are much more intra-organization calls than inter-organization calls, etc. Accordingly, the user address usually reflects this hierarchy. The administration domain also suggests some natural hierarchies.

In this paper, we propose a *Virtual Hierarchical Signaling System*(VHSS) in which a hierarchically organized signaling network controls a communication network with arbitrary topology. Like Signaling System No 7 (SS7) [?], a signaling network is independent of the communication network so that all the advantages of common channel signaling apply here. Unlike SS7, where a complete physical network could be built just for signaling, in our proposal, the signaling system is built on the same ATM network so that the cost of the signaling system can be greatly reduced.

Figure **??** shows a hierarchical signaling network with three levels of hierarchy that controls an underlying general communication network. The hierarchical signaling network is composed of signaling points and signaling links. A circle node in the picture represents a node in the communication network with its CP. To make it more general, we call it a level 1 SP. Each level i SP, except the highest level SP, has a signaling connection, the dotted line in the figure, with a level $i+1$ SP. The level $i+1$ SP becomes the signaling parent of the level i SP. A SP is a logical concept. A SP comprises software running on a computer together with signaling links to its signaling parent and signaling children. Multiple SP processes can be running on the same physical computer so that a powerful computer can be efficiently used.

Each SP knows its *management domain*, the set of node addresses under its control. Each signaling point manages the resources for all the links that connect two of its child signaling domains. A level i domain is identified by the level i SP which manages it. A link that connects two level $i-1$ domains is called a *level i link*. Two level i SPs that are connected by at least one level $i+1$ link are called *neighbors*. Besides the parent signaling connection, each level 1 SP has a signaling connection with each of its neighboring nodes. At any SP, there defines a mapping $SP(u)$ which maps a level 1 node u to its next lower level SP when u is in the current SP's domain or a neighboring SP if u is in this neighbor's domain or nil otherwise. For example, in Figure **??** a level 2 signaling point S22 manages the bandwidth for level 2 links l_3, l_4 and l_7, but not the level 1 links internal to node S13, S14 and S16. S21 and S22 are neighbors since link l_2 and l_6 connect their management domains. $SP(S16)$ at S31 returns S22 since S22 is the next lower level SP that

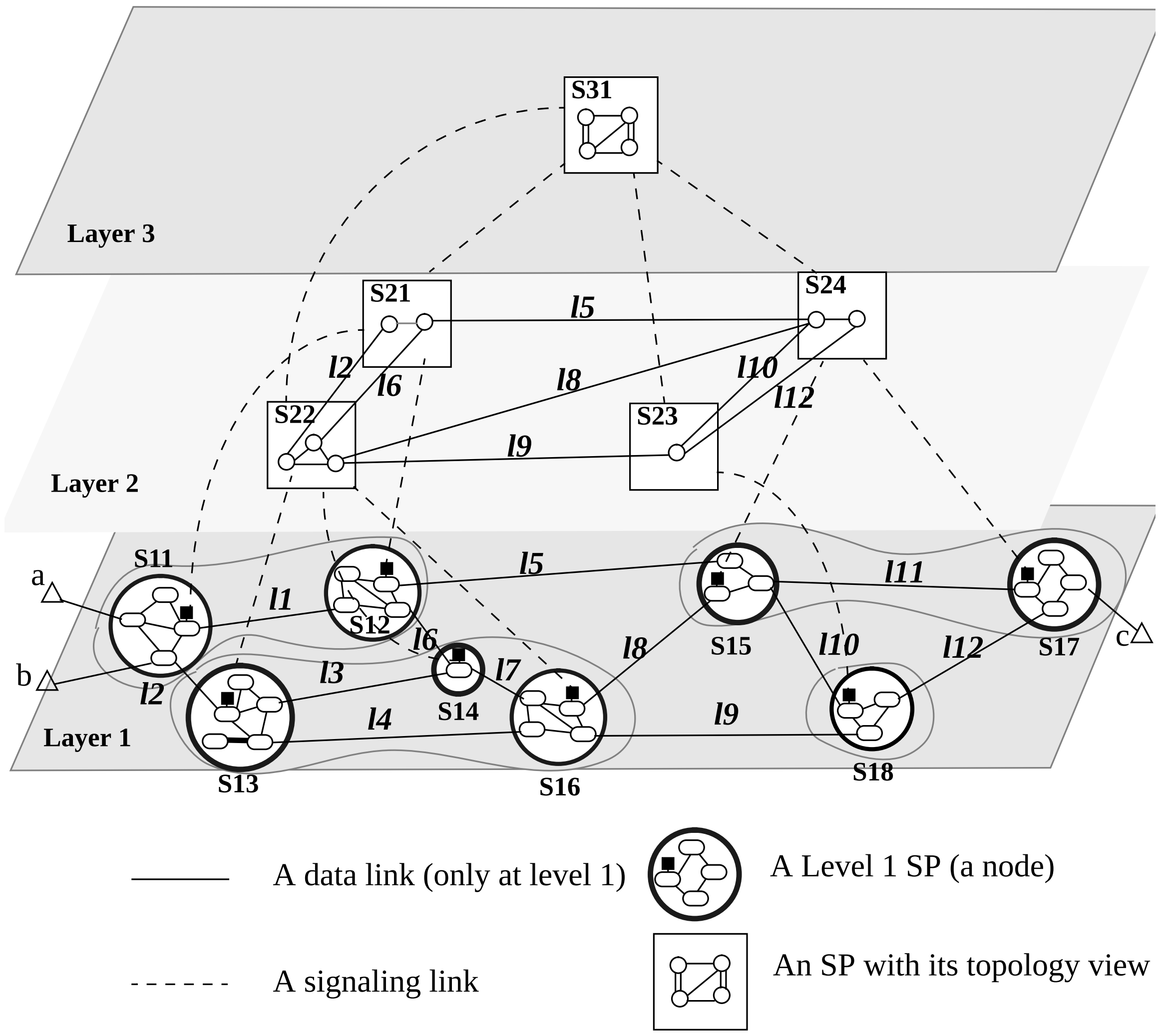


Figure 6: A Virtually Hierarchical Signaling System

contains node S16, and $SP(4)$ at S13 returns S11 since S11 is S13's neighbor that contains node 4. Only a level 1 SP really sets up switch tables that physically allocates bandwidth to a connection. The higher level SPs manage resources while making routing decisions.

The signaling algorithm is quite simple. Each SP maintains a connection data structure, like the one defined in section **??**, for every connection that has registered at that SP. When one SP can not resolve a *join* request, (the connection data structure does not exist at the SP and the root of the connection is not in the SP's management domain), the request is passed to its signaling parent. This upward message passing continues until either a connection data object for the connection is found or the root of the connection is in the SP's management domain. At that time a shortest path from the requesting point to the connection is calculated for the request. This shortest path only works at the current level. A *setup* request will be sent to all the signaling children along the path to request that they set up the connection. This *setup* message propagates downwards until it reaches the level 1 SPs. If all the level 1 SPs successfully allocate the bandwidth for the request, an ACK will propagate back to the higher level SP who sent the *setup* message. Otherwise a NACK will be passed to that SP. The result will pass back to the user down the hierarchy along the reverse path of the *join* request messages.

Before we define the details of the algorithm, let's look at some examples based on Figure **??**.

Example 1: User *b* tries to join a connection *c* which is owned by user *a* (S11 is the root). User *b* sends a *join* request to S11. Since the connection *c* exists at S11, S11 just allocates the resources to connect *b* to the connection.

Example 2: user *c* tries to join the same connection. When S17 receives the *join* request from user *c*, it can not resolve the problem since the connection does not exist at S17. It sends a *join* request to the higher level SP, S24. Since S24 can not resolve the request either, it forwards the request to S31. By checking the connection identifier, S31 knows that the connection should exist at S21. S31 picks a route from S24 to S21. In this case, it picks l_5. It then sends a *setup* message to both S24 and S21. When S21 receives the message, it tries to find a route to connect S11 with l_5. It picks l_1. It then sends a *setup* message to both S11 and S12. When S11 and S12 successfully find a route to connect l_1 with the existing connection and l_1 with l_5, they ACK their parent, S21. S24 does the similar thing. In this way, SPs work together to find a complete path to connect the user *c* with the connection.

3.2 VHSS Signaling Algorithms

Each operation is assigned a unique identifier when it starts. At each SP that participates in the operation there exists an object called an *operation handler*. An operation handler is created when a message related to that operation is delivered at the SP for the first time. It is the operation handler's responsibility to process all the messages related to that operation. An operation handler is destroyed when the operation completes. For any connection, at most one operation handler is assigned as the *current handler*. Only the current handler can process an incoming message. The operation corresponding to the current handler is said to be the *current operation*. A handler has a priority attribute which can be either HIGH or LOW. A handler created as the consequence of receiving a message from its signaling parent is assigned a HIGH priority. Otherwise it has LOW priority. A High priority handler will never change its priority. A LOW priority handler may change its priority to HIGH when a message from its signaling parent is processed or it has sent messages to its signaling children. A HIGH priority handler prevents the creation of other handlers for the same connection. Generally speaking, top-down operations have higher priority than the bottom-up operations.

We assume that there exists a *message delivery system*. The message delivery system guarantees the reliable delivery of messages to their proper operation handlers. When a message is received, the message delivery system will deliver the message to the current handler if this message is part of the current operation. If the handler for the message does not exist, the message delivery system may either create a new operation handler and deliver the message to the newly created handler or block the message depending on the priority of the current operation handler and the incoming direction of the message. The decision is made based on the algorithm shown in Figure **??**. When the current handler finishes, the message delivery system will scan through the blocked messages as if they were just received. Some of the message delivery system's functions defined here could be modeled in our signaling algorithm. We put them in the message delivery system only to simplify the presentation of the signaling algorithms. The priority assignment and FIFO property of the message delivery system are crucial to guarantee the correct behavior of the protocol. Figure **??** and Figure **??** show the signaling algorithms for each of the operation handlers.

We define some functions and mappings used in the algorithms. A mapping *root*(*cid*) maps the connection identifier *cid* to the address of the root node of the connection. A function *domain*() at a SP gives a set of addresses that are under the control of the current SP. $FindPath()$, $reserve()$, $allocate()$ and $release()$ are the resource manager's functions as defined in section 2. An SP-wide accessible routine $send(dest, message)$ sends the $message$ to the destination $dest$. A *connection* object has the structure as defined in section 2.1. The function $getConnection(cid)$ searches the system to find the connection with identifier *cid*. A connection object is returned if the search is successful. At each SP, a mapping $SP(u)$ or $SP(lk)$ maps a node u or a link lk to a child SP or a neighbor SP depending on the context. A level i path is a list of interleaved level $i-1$ SPs and level i links. A macro $L(p,r)$ returns the link which is on the left of SP r in the path p. Similarly $R(p,r)$ gives the right link of r. A macro $numberOfSP(p)$ gives the number of SPs in the path p. A macro *lastLink(p)* gives the last link in path p.

A *join* message is processed in four phases. When a *join* operation is initiated by a user, a *joinReq* message is generated and this *joinReq* message is passed up the signaling hierarchy until either a connection data object is found or the root of the connection is in the SP's domain. A multicast phase begins when the higher level SP chooses a path to route the connection request and multicast a *setup* message to all the child SPs along the path. A converge-cast phase followes in which every SP reports the operation status after it gets a response from its children that are involved in the operation. In the fourth phase, a *joinCommit* message propagates to all the SPs participated in the operation to inform the SPs to commit or abort the operation. At the same time a *joinResp* message is passed to the user to report the result of the operation.

There are five types of messages that are involved in the *join* operation. A *joinReq* message conveys the operation request going up the signaling hierarchy. It contains the connection identifier, the bandwidth requirement, and the source SP address as the information elements. A *joinResp* is the response message for a *joinReq*. It carries an operation status, either ACK or NACK, along with the connection identifier. A *setup* message is used by a higher level SP to request a set of signaling children to set up a path to route a connection request. It contains the connection identifier, the bandwidth requirement, and the sending SP as the information elements. A *setupResp* is a response message for a *setup* message. A *commit* message informs the lower level SPs to commit or abort a *join* operation. Besides the connection identifier, it carries a command information element. If the *command* is ACK, it directs the receiver to commit the operation. A NACK *command* directs the receiver to abort the operation.

A *create* operation handler is the simplest. A *create* handler only appears at the lowest level. When a *createReq* is received, the level 1 SP creates a connection data object and sends an ACK back. A *createReq* message carries the connection identifier, the bandwidth and the user sp identifier.

A *drop* operation is a little bit tricky. Since the higher level SP does not know the connection graph internal to a lower level signaling domain, the *drop* operation has to proceed hop by hop at the lowest level. Whenever a level i link is involved, a *dropReq* message is passed to the level i SP. The SP makes a *commit/abort* decision based on the connection graph at that level. Two message types are involved in a *drop* operation. A *dropReq* message carries a connection identifier, the sender SP, and the link identifier indicating the link the *dropReq* came from. A *dropReq* message serves to report to the signaling parent to start a *drop* operation as well as to be passed at the lowest level to request release of physical resources.

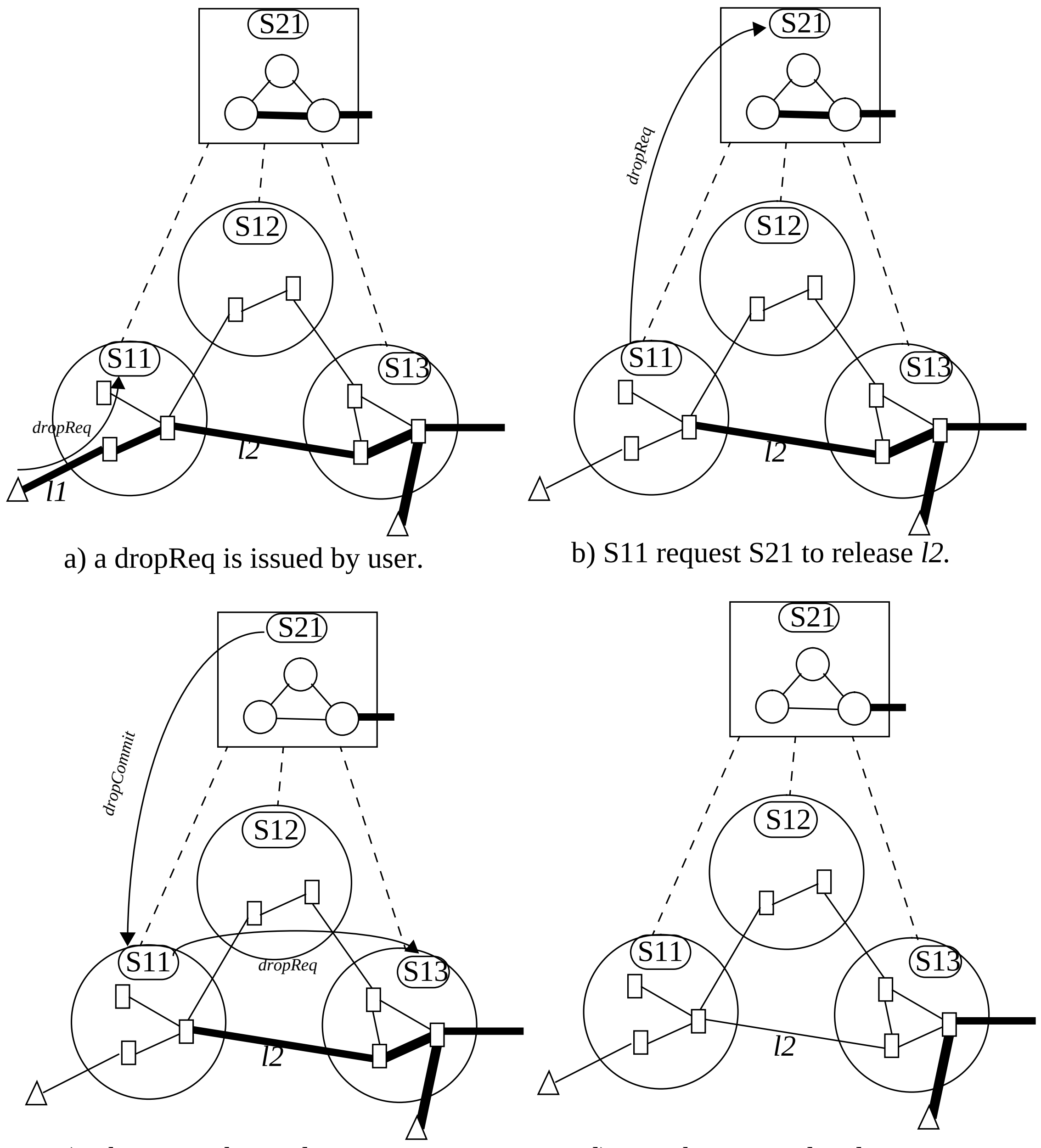
Figure **??** illustrates the drop operation. The figure inside a SP shows the SP's view of the connection which is shown as the thick line in the figure. When a *dropReq* reaches $S11$, a level 1 SP, S11 releases the resources up to link $l2$ which is a level 2 link. It requests the release of link $l2$ by sending a *dropReq* to its signaling parent, S21. Since $l2$ is the only link incident to S11, S21 agrees to release $l2$ by sending a positive *dropCommit* message to S11. Receiving the *dropCommit*, S11 releases the resources on $l2$, and sends a *dropReq* to its neighbor S13. S13 releases the resources

```
Join Handler
/* Handler-wide accessible variables
and initial values */
  Link          srcLK := nil;
  Path          p := nil;
  integer       receivedResp := 0;
  OpResult      opResult := ACK;
  integer       i := Current SP's level;

• joinReq(cid,bw,lk)
/* comes from a child SP */
  srcLK := lk;
  con := getConnection(cid);
  if ((con ≠ nil) OR (root(cid) ∈ domain()))
    if (con = nil)
      con := new connection(cid, bw);
    p := findPath(bw,SP(lk),con.graph(V));
    if (p = nil)
      send(SP(lk),joinResp(cid,NACK));
    else
      reserve(bw,p);
      if (i = 1)
      /* level 1 SP really allocate bandwidth */
        allocate(bw,p);
        send(SP(lk),joinResp(cid,ACK));
      else
        for (r ∈ p)
          send(r,setup(cid,bw,L(p,r),R(p,r)));
        end for
      endif
    endif
  else
    send(parentSP, joinReq(cid,bw,lk));
  endif

• joinResp(cid,result)
  send(SP(srcLK), joinResp(cid,result));

• joinCommit(cid,command)
  if (command = ACK)
    if (i = 1)
      allocate(bw,p);
    else
      for (r ∈ p)
        send(r,joinCommit(cid,command));
      end for;
    endif
    con.updateGraph(ADD, p);
  else
    go back to the state before the operation
  endif
  delete myself;

• setup(cid,bw,lk1,lk2)
/* come from the signaling parent */
  if (lk1 = nil)
    lk1 = srcLK;
  endif
  if (lk2 ≠ nil)
    con := new connection(cid, bw);
    p := findPath(bw,SP(lk1),{SP(lk2)});
  else
    con := getConnection(cid);
    if (con = nil)
      con := new connection(cid, bw);
    endif
    p := findPath(bw,SP(lk1),con.graph(V));
  endif
  if (p = nil)
    send(parentSP,setupResp(cid,NACK));
  else
    reserve(bw,p);
    if (i = 1)
      send(parentSP,setupResp(cid,ACK));
    else
      for (r ∈ p)
        send(r,setup(cid,bw,L(p,r),R(p,r)));
      end for
    endif
  endif

• setupResp(cid,result)
  receivedResp := receivedResp + 1;
  if ((result = NACK) AND (opResult = ACK))
    opResult := NACK;
  endif
  if (receivedResp = numberOfSP(p))
    if (parentSP ≠ nil)
      send(parentSP, setupResp(cid,opResult));
    else
      send(SP(srcLK), joinResp(cid, result));
      for (r ∈ p)
        send(r,joinCommit(cid,opResult));
      end for
      if (opResult = ACK)
        con.updateGraph(ADD,p);
      else
        go back to the state before the operation
      endif
    delete myself;
    endif
  endif
```

Figure 7: Signaling Algorithms for VHSS (*join* Operation Handler)

```
Create Handler
• create(cid,bw,sp)
  con := new connection(cid,bw);
  send(sp,createResp(cid,ACK));
  delete myself;

Drop Handler
  SignalingPoint srcSP := nil;
  Path p := nil;
  Link l;
  integer i := Current SP's level;

• dropReq(cid,sp,l)
/* a dropReq may come from a signaling child
  or a level 1 signaling neighbor */
  srcSP := sp;
  con := getConnection(cid);
  if (i = 1)
    p := con.findPathToBranch(sp);
    l := lastLink(p);
    if (l is a higher level link)
      release(bw,p-l);
      con.updateGraph(con.bw,REMOVE,p-l);
      send(parent,dropReq(cid,currentSP,l));
    else
      release(bw,p);
      con.updateGraph(con.bw,REMOVE,p);
      delete myself;
    endif
  else
    if (l is a higher level link)
      send(parent,dropReq(cid,currentSP,l));
    else
      if (l is the only level i link in con for sp)
        release(con.bw, {l});
        con.updateGraph(cid,REMOVE,{l});
        send(sp,dropCommit(cid,ACK));
      else
        send(sp,dropCommit(cid,NACK));
      endif
      delete myself;
    endif
  endif

• dropCommit(cid,command)
  con := getConnection(cid);
  if (i = 1)
    if (command = ACK)
      release(bw,l);
      con.updateGraph(cid,REMOVE,{l});
      send(SP(l),dropReq(cid,currentSP,l));
    endif
  else
    if (command = ACK)
      delete con;
    endif
    send(srcSP,dropCommit(cid,command));
  endif
  delete myself;

Destroy Handler
• destroyReq(cid)
  con := getConnection(cid);
  if (exist higher level links in con)
    send(parent,destroyReq(cid));
  else
    release(con.bw, con.graph);
    for (r ∈ con.graph(V))
      send(r,destroyCommit(cid));
    end for
    delete myself;
  endif
  delete con;

• destroyCommit(cid)
  con := getConnection(cid);
  release(con.bw,con.graph);
  if (i ≠ 1)
    for (r ∈ con.graph(V))
      send(r,destroyCommit(cid));
    end for
  endif
  delete con;
  delete myself;
```

Figure 8: Algorithms for VHSS (*create*, *drop*, *destroy* Handlers)



a) a dropReq is issued by user.

b) S11 request S21 to release *l2*.

c) when S21 changed its view,
S21 confirms drop *l2*.
S11 sends dropReq to S13.

d) S13 changes its local view.
No need to report to its parent.

Figure 9: An Example of Drop Operation

```
if (the message is for the current handler)
  deliver the message to the current handler
else
  if (the current handler has LOW priority and
      the message comes from the parent)
    create a new operation handler for the
    message, then deliver the message to that
    handler and assign it as the current handler
  else
    block the message
  endif
endif
```

Figure 10: Algorithm for the Message Delivery System

to complete the drop operation. S13 does not need to send a request to its signaling parent since no more higher level links are involved in the drop operation.

A *destroy* operation is started by the owner of the connection by sending a *destroyReq*. This *destroyReq* is passed up the hierarchy until the highest level SP that is involved in the connection is reached. Then a *destroyCommit* message is multicast to all the SPs that are involved in the connection. The connection data object is deleted when the *destroyCommit* is received. At the bottom level, all the resources allocated for the connection are released.

3.3 Correctness and Performance Evaluation

In this section, we will prove the correctness of VHSS and evaluate its performance.

In a centralized signaling system, the correctness can be easily guaranteed since the central CP keeps track of all the network status and maintains the entire connection graph in its connection data structures. Furthermore, all the operation requests are processed sequentially. In this case, it can guarantee that a *join* request will always go through if the network has enough bandwidth to support it. In a distributed signaling system, the network status and the connection graph information is distributed in the network. To prove the correctness of the distributed algorithms for a signaling system, we need to extend our problem definition in a distributed way.

A *local connection descriptor* at a SP s is defined as a tuple $[s, c_s, r_s, w_s, H_s = (W_s, F_s)]$ where s is the SP's identifier, c_s is a connection identifier, r_s is the root of the connection, w_s is the bandwidth requirement and: if s is a level 1 SP, H_s is a subgraph of G whose vertices are restricted to s and/or its neighbors in G; if s is a higher level SP, H_s is a graph whose vertices are its child SPs and its signaling neighbor SPs, and whose edges are the links of G that connect two signaling domains of its vertices.

Let GD denote a set of (global) connection descriptors. Let GD(c) be a connection descriptor with connection identifier c. Let LD be a set of local connection descriptors. Let LD(c) be the subset of LD with connection identifier c, and $\mathrm{LD}_s(c) = [s, c, r_s, w_s, H_s = (W_s, F_s)]$ denote the elements of LD(c) whose first element is s.

We say that LD(c) implements $\mathrm{GD}(c) = ([c, T, r, w], H = (W, F))$ if for all $\mathrm{LD}_s(c) \in \mathrm{LD}(c)$, $r = r_s$, $w = w_s$, and

(1) if s is a level 1 SP

$W = \bigcup_{LD_s(c) \in LD(c)} W_s$, and $F = \bigcup_{LD_s(c) \in LD(c)} F_s$ and
for all $u, v \in V$, $(u,v) \in F_u \Leftrightarrow (u,v) \in F_v$

(2) if s is not a level 1 SP
for all $(u,v) \in F_s \Leftrightarrow (u,v) \in F_{SP(u)}, (u,v) \in F_{SP(v)}$ and
$(u,v) \in F_s \Rightarrow SP(u), SP(v) \in W_s$

We say that LD implements GD if LD implements every descriptor in GD and no others. This defines the safety conditions for the VHSS in a distributed way. The condition (1) is general enough to give the correctness conditions for any distributed signaling system. The condition (2) only applies to VHSS and is included to ensure that VHSS fulfils condition (1). The local connection manager algorithm, running at each SP of a signaling network, maintains a set of local connection descriptors. Together, they implement a set of global connection descriptors. Notice that our connection data structure is consistent with a local connection descriptor. We define a *good state* to be a state where both condition (1) and (2) are satisfied.

Before we start to prove the correctness of the algorithms, let's introduce some notations. We define an event $e(s, c, op, typ, [status])$ to be one call of any handler's event processing routine, where s is a SP's identifier indicating the place where the event occurs, c is the connection identifier, op is the operation identifier, typ is the message type, and the optional $status \in \{$ *ack, nack* $\}$ gives the result of the event. We say $\mathcal{E}$ is an execution of the system iff $\mathcal{E}$ is a sequence of events that satisfies the natural order of events, such as the event for sending a message always appears before the event for receiving the same message. Let $\mathcal{E}(c)$ be the subsequence of $\mathcal{E}$ with all events related to connection c. A *stable state* is a state in which no event will occur if there are no new user requests. We define a *complete execution* to be an execution with all the operations are completed. We say that a system is correct if starting from any good stable state, it is guaranteed to reach a good stable state after any complete execution.

We prove the correctness by induction. Initially the system starts in a good stable state since there is no connection at all. We prove the correctness in three steps. First we prove that the system behaves correctly if there is only one operation in the execution. Then we prove that the system behaves correctly if the execution contains multiple operations for a single connection. Finally we prove the correctness for any complete execution.

Lemma 1 *If the system starts in any good stable state and starts a complete execution $\mathcal{E}$ in which all events relate to a specific operation of connection c, then the system will stop at a good stable state where the new LD(c) implements the new GD(c).*

proof : The lemma can be proved by carefully checking the algorithms for each operation separately.

- **case 1: op = create**

  When the level 1 SP, the CP, receives a $create(cid, bw, sp)$, it creates a connection data object with itself as the only vertex in the connection graph. It sends a response immediately and no follow up event will occur. Obviously, the new LD(c) implements the new GD(c). The execution $\mathcal{E}$ contains only one event.

- **case 2: op = join**

  The *joinReq* message propagates up the hierarchy. Since the highest level SP can see the whole network, this upward propagation has to stop at some SP. Then the downward propagation begins. During the downward propagation of the *setup*, each level i SP chooses a level i

path to complete a section of a level $i + 1$ path. When the *setup* reaches the level 1 SPs, a whole path from the requesting user to the node where the connection exists is found. When successful, this path is added into all participating SPs after a positive *joinCommit*. The resulting connection data structure at all the SPs implements the new GD(c). If one of the SPs fails to find a path, the negative *joinCommit* erases all the modifications for the operation, and the new LD(c) at the end of the operation is same as the old LD(c) that implements the old GD(c). Since the request is rejected, the new GD(c) is equivalent to the old GD(c).

- **case 3: op = drop**

  From the algorithm, we can see that a level 1 node releases the bandwidth on a level i link only when it is approved by the level i SP who manages that level i link. When the *drop-Commit* that approves the release of bandwidth propagates downwards, all the intermediate SPs modifies their connection data structures. Since no other operations are performed in the execution, the *dropReq* will propagate to the place where the connection branches. At the end of the operation, the new LD(c) implements the new GD(c).

- **case 4: op = destroy**

  A *destroy* operation releases all the bandwidth reserved for the connection and deletes all the connection data structures at all the SPs. So at the end of the operation, the new LD(c) implements the new GD(c).

#

A *drop* operation propagates and gets committed hop by hop. So we can think an external *drop* operation as composed of multiple internal *drop* operations, each of which starts when a level 1 SP receives a *dropReq* and ends when a *dropCommit* has been processed at the same SP. When we refer to an operation in the following lemmas, we refer to this kind of internal operation.

Lemma 2 *If the system starts in any good stable state and starts a complete execution $\mathcal{E}$ in which all operations relate to a single connection c, then there exists another execution $\mathcal{E}$' which is a permutation of $\mathcal{E}$ such that in $\mathcal{E}$' all the events relating to a single operation are consecutive and both $\mathcal{E}$ and $\mathcal{E}$' leave the system in the same final state.*

proof : The SPs that are involved in an operation form a tree. We call it an *operation tree*. If two operation trees are disjoint, we can reorder the events so that all events for one operation appear before all events for the other. Since events for different operations modify different connection objects at different SPs, the reordering does not affect the final state.

Notice that the propagation of *joinReq*, *joinResp*, and *destroyReq* do not change connection graphs at any SP, we can rearrange all *joinReq* for one *join* operation just before the root of the *join* operation tree starts changing *joinReq* to *setup*. This reordering does not affect the end state of an execution. The same arguments can be applied to *joinResp* and *destroyReq*. So from now on, we can view a *join* operation as starting at the root of the operation tree when the root sends *setup* messages to its children and ends when the *joinCommit* messages have propagated to the whole operation tree. Notice that a *join* or *destroy* operation starts at the root of the operation tree, and an internal *drop* operation starts at one leaf of the operation tree, a single branch tree. We call a *join* or *destroy* operation a *top-down* operation, and a *drop* operation a *bottom-up* operation.

Let's consider a *join* operation interleaved with a *drop* operation. If the *dropReq* reaches its root of the operation tree before the *setup* reaches the same SP. Due to the FIFO characteristic of the signaling channel, the *drop* operation will be complete at any SP before the starting of the *join* operation at the same SP. In this case, rearranging the *drop* operation before the *join* operation does not change the final state. If a SP receives a *setup* message while waiting for the *dropCommit*, the *dropReq* and the *setup* must have crossed on their way. Since the downward *setup* will generate a HIGH priority *join* handler, no *dropCommit* will be received by any SP before the completion of the *join* operation. In this case, we can rearrange the whole *join* operation ahead of the entire *drop* operation without changing the final state of the execution.

The result of interleaved *join* and *drop* operations can be generalized. Any top-down operation has higher priority than any bottom-up operation. Any blocked bottom-up operation can be rearranged totally after the top-down operation that blocks the bottom-up operation. A bottom-up operation that reaches its operation root before a top-down operation can be rearranged totally before the top-down operation due to the FIFO characteristic of the message delivery system. Two top-down operations will be serialized when they propagate to a common SP which must be the root of one of the operation trees. As a result, any execution for one connection can be serialized. #

Corollary 3.1 *If the system starts in any good stable state and starts a complete execution $\mathcal{E}$ in which all operations relate to a single connection c, it will end in a good stable state.*

proof : Based on **??**, operations can be serialized. According to **??**, after each operation, the system goes to a good state. #

The operations for different connections only interact through the resource manager. We call the points when a SP executes a *reserve*, an *allocate*, or a *release* the *resource access point.* If we can rearrange the execution sequence so that the operations for each connection are serialized and the order of the resource access points are unchanged at each SP, then the new execution will lead to the same state as the original execution. This argument leads to the following lemma.

Lemma 3 *If the system starts in any good stable state and starts a complete execution $\mathcal{E}$, then there exists another execution $\mathcal{E}'$ which is a permutation of $\mathcal{E}$ such that $\forall c \in C$, $\mathcal{E}'(c)$ is serialized and for all SPs, the order of the resource access points are the same as the order in $\mathcal{E}$.*

proof : From the proof of the **??**, we can see that the order of the resource access points are preserved at each SP when we serialize the operations for one connection. We can serialize each $\mathcal{E}(c)$ individually and then mix the $\mathcal{E}(c)$s by preserving the order of the resource access points for each SP. #

Lemma 4 *The capacity constraint is always strictly observed.*

proof : In VHSS, links are partitioned by their management SPs. The bandwidth of a level i link is solely managed by a level i SP. Any allocation or release of the bandwidth on that link is initiated by that SP. At a level i SP, to route a *join* request it calls a global resource manager function *findPath* to find a level i path with required bandwidth. Before sending a *setup* message to all signaling children on the path, the SP reserves the bandwidth on all the level i links in the path. The bandwidth reserved will not been seen by the later call of *findPath* until it is released. A level 1 node will only allocate the bandwidth for a level i link when it has been reserved by the level i SP. #

Theorem 3.1 (Safety) *The VHSS is a safe system.*

proof: By lemma **??**, any execution is equivalent to another execution in which operations for every connection are serialized. By lemma **??**, every individual operation maintains a global connection descriptor. Different connections only interact through the resource manager when they request or release resources. Lemma **??** proves that the resource constraint is always preserved. Altogether, VHSS is safe. #

Now we will prove the low level liveness of our system, that is that every operation will complete.

Theorem 3.2 (Low Level Liveness) *If at some point in an execution, no new operation requests are made, after a sufficiently long time period all operations finish.*

proof sketch: Operations for different connections are processed independently. They never block each other. We only need to prove that a set of operations for the same connection will finish. We prove the lemma by induction. When there is only one operation in the system, we can prove the lemma in the same way as we prove lemma **??**. Assume that any N operations for the same connection can finish. Now we consider the case that we have $N+1$ operations. Notice that when one HIGH priority operation, say op_1, becomes the current operation at a node SP, it will remain being current for its life time. The HIGH priority operation will propagate to all its children in the operation tree. This propagation will only be blocked at some SP where a HIGH priority operation, say op_2 is current. Notice that the root of op_2's operation tree is on the lower level than the root of op_1's. As the consequence, the HIGH priority operation that has the lowest root can always go through. If all the operations are with LOW priority, then at least one of them can reach its root. After that it becomes the HIGH priority operation. By the discussion, at least one operation can go through. By the induction hypothesis, the lemma is true. #

Lemma 5 *For every node u, the CP of u has deg(u)+1+(number of local users) signaling connections. For every other SP, there are O(B) signaling connections where B is the maximum branching factor of the SP. The single request process times for different requests are given in table 2 where n is the network size, b is the minimal branching factor, and d is the diameter of the network.*

System Type	Single Request Processing Time	Single Request Message Complexity	Busiest CP Message Complexity
create	$O(1)$	$O(1)$	$O(1)$
join	$O(B * log_b(n))$	$O(d)$	$O(B)$
drop	$O(d)$	$O(d)$	$O(B)$
destroy	$O(log_b(n))$	$O(d)$	$O(B)$

Figure 11: Single Request Processing Complexity

proof : The number of signaling connections for all SPs are obvious. A *join* operation has four phases. Phase 1 takes $O(log_b(n))$ time units. Phase 2 and 3 together takes $O(B * log_b(n))$ time units. B and b are usually constants decided by the network designer. A *destroy* operation does not need response. So it take $O(log_b(n))$ for the request to propagate to all the level 1 nodes. The *drop* messages are passed hop by hop, so that the execution time is $O(d)$. Since the user's satisfaction depends mostly on the response time to *join* request, the system is scalable.#

3.4 Live Lock Problem

Live lock is an interesting phenomenon in a distributed system. In our context, the live-lock problem can be stated as follows: There is a set of *join* operation requests, each of which can be successfully routed by the *findPath* functions at different levels if the operations come individually. Since the events of the operations are interleaved in some order, none of them can be satisfied. With the configuration of the system unchanged, the same sequence of events can occur again and again. No progress is made even though all the SPs work properly. It is very similar to the classical deadlock problem. Each request holds some resources and requests others. The wait-for relationship forms a loop. In a distributed system, live-lock is an annoying problem. In the following lemma, we prove that VHSS is live-lock free.

Lemma 6 *Live-lock will never occur in VHSS.*

proof sketch: We prove the lemma by contradiction. Assume that $\mathcal{E}$ is an execution with a set of *join* operations, $\{j_1, j_2, \ldots, j_m\}$, that leads to a live-lock. We assume, w.l.o.g, that $\{j_1, j_2, \ldots, j_m\}$ is ordered by the time when the requests reached the roots of their corresponding operation trees. Let $\{t_1, t_2, \ldots, t_m\}$ and $\{r_1, r_2, \ldots, r_m\}$ be the operation trees and the roots of the trees respectively. Assume r_1 is a level i SP. Since j_1 is the first *join* request that reaches r_1 and by the live-lock assumption, r_1 can chose a level i path to route j_1. Since j_1 can not successfully complete, there must be another *join* operation request that reaches one of r_1's descendents, say s, before j_1 and has taken the bandwidth that j_1 requires to complete j_1. Assume, w.l.o.g., j_2 is the earliest that reaches s. By the FIFO assumption of the message delivery system, s must be the root of the j_2's operation tree. Now we reorder the sequence of the operations as $\{j_2, j_1, \ldots, j_m\}$ where j_2 is the earliest to reach s and s is on the lower level than r_1 is.

Apply the same argument repeatedly. Eventually we can find a sequence of operations $\{j_1', j_2', \ldots, j_m'\}$ which is a permutation of $\{j_1, j_2, \ldots, j_m\}$. In this sequence, j_1' is the first in $\mathcal{E}$ to reach its root and when the *setup*s for j_1' propagate to the connection tree of j_1', no other operation has taken any bandwidth required by j_1'. According to the live-lock assumption, j_1' can go through. This contradicts the assumption. #

4 Conclusion and Future Work

VHSS takes advantage of the flexibility that an ATM network provides. It provides all the benefit of a common channel signaling system like SS7. The same ATM network works both for data communications and signaling processing in an integrated way. The signaling protocol is simple to implement. It scales well which is extremely important for a public telecommunication network.

In this paper, we assume all the system components are reliable which is not the case in a real system. By combining VHSS with the layered approach, especially at the higher level, it can be made reliable and cost efficient.

In this paper we omit one important measure for a signaling system: how efficient is the signaling system in terms of managing the resources. Some measures such as the throughput, profit, or competitive ratio can be used to evaluate the efficiency of using resources. To evaluate these measures, we may have to expand our model to include link cost, per call profit or similar attributes for networks and connections. Awerbuch [**?**] proposed a point-to-point routing algorithm for a central controlled system. Their algorithm is optimal in terms of competitive ratio, which is defined as the ratio of the throughout of the algorithm to be evaluated versus the throughput of an optimal off-line algorithm. We have proved the similar result for a centralized multipoint communication network. We need to expand our model to do some resource efficiency analysis.

Besides theoretical analysis, we are planning to do some simulations to compare the performance of a VHSS with other signaling systems. The proofs of the algorithms presented in this paper are not complete. The detailed formal analysis will appear in follow up papers.

5 Acknowledgements

In completing this work, I have had the benefit of many fruitful (and often spirited) discussions with professors and colleagues at Washington University. Dr. Jonathan Turner spent a lot of his time with me to discuss the ideas and detailed system design. Andy Fingerhut, John DeHart, and Ken Goldman all provided very useful input.

References

[1] Baruch Awerbuch, Yossi Azar, and Serge Plotkin. Throughput-competitive on-line routing. In *Proceedings of 34th FOCS93*, 1993.

[2] CCITT. Specification of signaling system no. 7. *CCITT Blue Book*, 1988.

[3] M. Imase and B. M. Waxman. Dynamic steiner tree problem. *SIAM J. on Disc. Math.*, pages 369–384, 3 1991.

[4] R. Rey. *Engineering and Operations in the Bell System.* Marray Hill, 1983.

[5] W. Stallings. *ISDN.* MaCMillan, 1989.

[6] J. S. Turner. An optimal non-blocking multicast virtual circuit switch. *IEEE INFOCOM*, pages 298–305, 1994.

[7] B. M. Waxman. Routing of multipoint connections. *IEEE J. Select. Areas Comm.*, pages 1617–1622, 6 1988.