ISSUES IN DISTRIBUTED CONTROL FOR ATM NETWORKS

Jonathan S. Turner
Washington University, St. Louis
jst@cs.WUSTL.edu

WUCS-95-12

May 17, 1995

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

Asynchronous Transfer Mode (ATM) network technology is expected to become a central part of the emerging global information infrastructure. ATM networks introduce a number of features that distinguish them from earlier technologies and introduce new issues in network control. This paper offers a framework for precisely defining and analyzing alternative approaches to the distributed control of ATM networks and explores some of the key design issues through a series of examples. It is hoped that it will provide a useful foundation for researchers in networking and distributed computing interested in exploring these issues further and developing more complete solutions.

[0]This work was supported by the ARPA Computing Systems Technology Office, Ascom Timeplex, Bay Networks, Bell Northern Research, NEC, NTT, Southwestern Bell and Tektronix.

ISSUES IN DISTRIBUTED CONTROL FOR ATM NETWORKS

Jonathan S. Turner
Washington University, St. Louis
jst@cs.WUSTL.edu

1. Introduction

Asynchronous Transfer Mode (ATM) network technology is expected to become a central component of the emerging information superhighway, both in the United States and around the world. The last decade has seen tremendous progress in developing the core ideas for ATM networks and in creating scalable, high performance switch architectures that are suitable for wide-scale deployment in both public and private networks. However, while the core switching problems are now fairly well understood, our understanding of the higher level control problems is incomplete and often confused. This is understandable, since these higher level problems are logically complex and difficult to separate from one another. These inherent difficulties are made worse by the lack of any common conceptual framework in the research literature, making it difficult to clearly define the key problems and evaluate possible solutions. In general, the literature on network control in ATM and its historical antecedents (e.g. the telephone network, virtual circuit data networks) focuses on specific protocols and the communication of signaling messages, ignoring the algorithmic questions of network control [1, 3, 6, 7, 8, 9, 11, 12, 13, 15, 17, 21]. Discussion of alternative design approaches and useful surveys are rare [20]. This paper is an attempt to formulate the network control problem for ATM and similar networks, in a way that clarifies the key issues and provides a framework for carefully addressing issues of correctness, computational efficiency and resource allocation. It does not offer any complete solutions to the problem but identifies some intersting points in the design space and develops some basic understanding of their merits. It is directed to researchers in both distributed systems and networking, since the creation of effective solutions for these problems requires the methods and perspectives of both research communities.

ATM technology has its roots in research projects in CNET [4, 5] and Bell Labs'[14, 16] in the early eighties. After more than a decade of research and development, it has emerged as a central technology component for the emerging information superhighway. ATM is a virtual circuit technology, meaning that information to be carried over the network is broken up into blocks (called *cells* in ATM) with an identifying label (called the *Virtual Circuit*

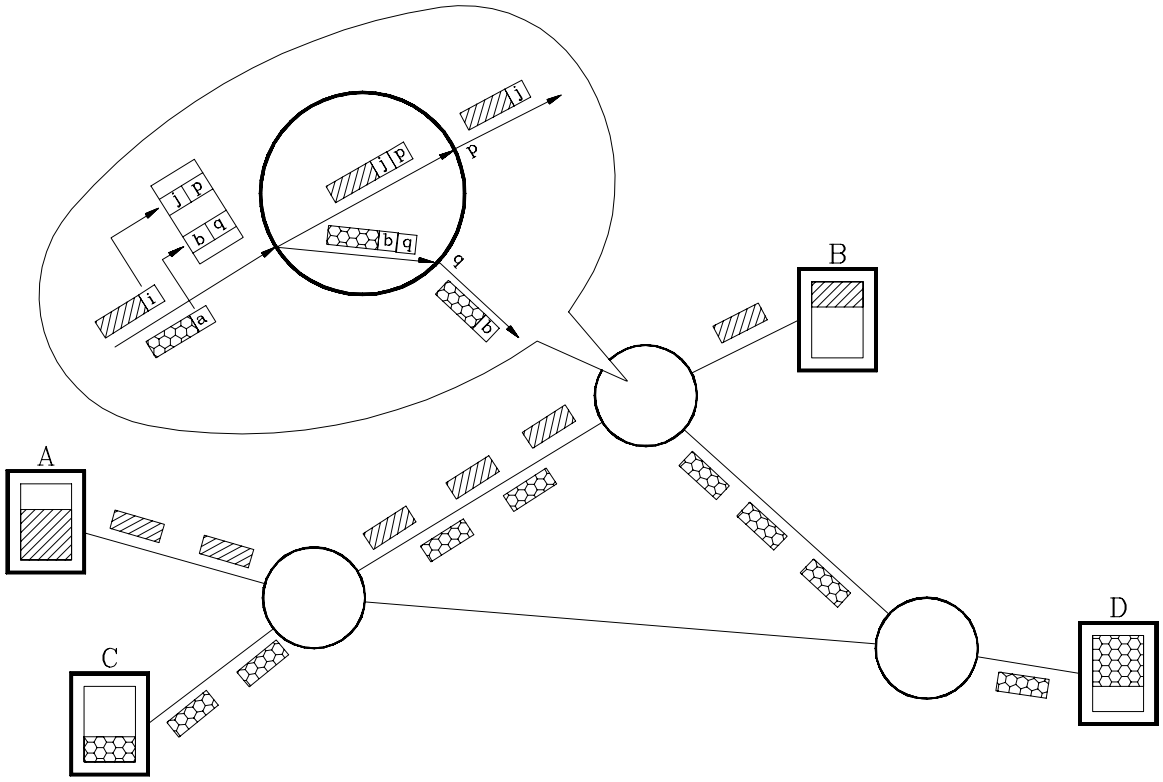


Figure 1: Virtual Circuit Switching

Identifier or VCI) attached to each block. The VCI is used to forward information along a fixed path in the network, defined by hardware tables in the switching systems along the path. This is illustrated in Figure 1 in which a cell arriving at the central switching system with a VCI of i is forwarded to output port p of the switch and assigned a new VCI of j. ATM networks can support virtual circuits that operate at any data rate, up to the capacity of the links that carry them. They can also vary the rate of transmission, allowing the use of statistical multiplexing to improve transmission efficiency. While they are not quite as flexible as datagram-oriented packet networks, they are fare more amenable to large-scale hardware implementations, leading to dramatic gains in cost-effectiveness when used for large networks.

Because one of the key intended applications for ATM is carrying real-time audio and video, ATM provides for reservation of link capacity for individual virtual circuits. In the simplest case, one can simply reserve a fixed fraction of the link capacity for different virtual circuit. In many cases the reserved capacity will correspond to the fixed transmission rate used by the terminals communicating over the virtual circuit. In cases where the terminals vary their transmission bandwidth, it may correspond to a long term average or some notion of effective transmission rate. We are not concerned with the issue of how these rates are obtained in this paper, and simply assume that each connection requires some fixed amount of capacity that can be allocated by the network control system. In the last section of the paper, we return to this issue.

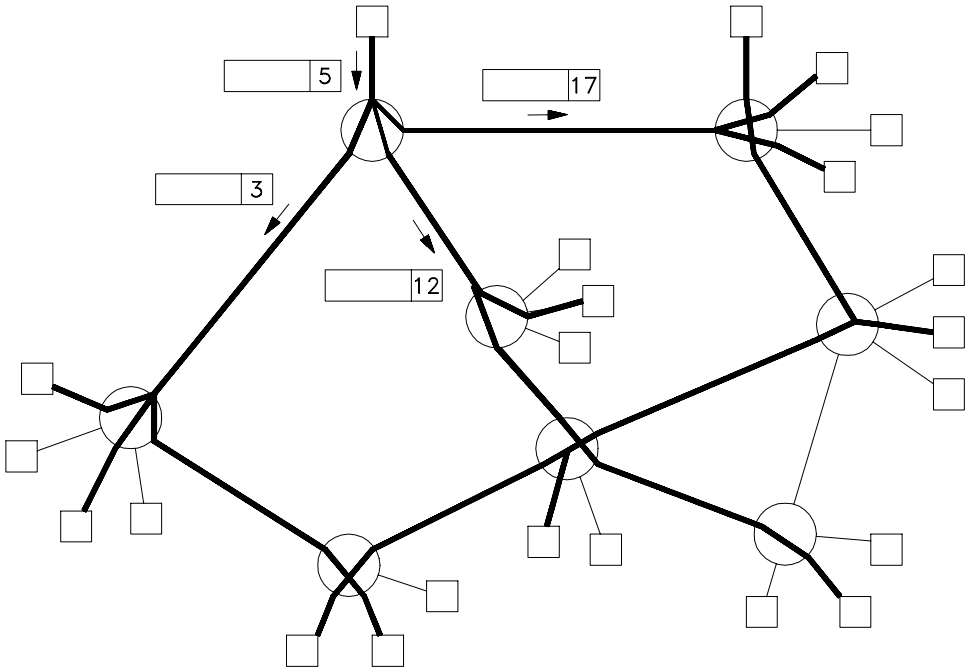


Figure 2: Multicast Virtual Circuits

One of the more interesting characteristics of ATM networks is their ability to support multicast virtual circuits, in which information from a single source is replicated and distributed to multiple destinations 2. The replication and forwarding is implemented in the switching system hardware. The management and control of such multicast virtual circuits is the main focus of this paper. In the simplest case, a multicast virtual circuit has a single sender and multiple receivers. However, it can also be useful to have virtual circuits in which each participant can both send and receive. We generalize this further by allowing virtual circuits in which some endpoints can be designated *senders*, some *receivers* and some can be both.

When considering design alternatives for network control systems, it's important to have a clear understanding of a variety of engineering issues. ATM technology is intended for networks ranging from campus networks with as few as one thousand users, to large public networks with hundreds of millions. Individual switches will support from 10 to 10^5 users and link speeds will range from 1 Mb/s to 10 Gb/s. Some connections may be highly dynamic. For example, if video programs are distributed through ATM networks with users able to switch among different connections by clicking a television's remote control, 'channel surfing' can create heavy loads on the control system. A less extreme, but still important case is the the use of browsing programs like Mosaic for retrieving information from distributed hypermedia information servers. Intrinsic network latencies can range widely (from less than a millisecond to hundreds of milliseconds), but will typically be dominated by simple propagation delays due to the finite speed of light. Because control latency can limit the speed with which connections can be modified, it is desirable to keep it as close as possible to the intrinsic network latency. Cost is determined primarily by the number,

capacity and length of the physical links, as well as the number and capacity of switches and the associated control system. There are major cost advantages to using the largest capacity links feasible when going over geographically significant distances. Hence, even if user access links are limited to moderate rates (typically 150 Mb/s), network backbone links will often by faster (600 Mb/s and 2.4 Gb/s today, 10 Gb/s by the end of the decade). Networks do exhibit natural clusters, reflecting organizational and social structures, as well as geography. These can often be used to make network control operations more efficient. The huge bandwidth of ATM networks makes it cost-effective to transport large quantities of data. For distributed control systems this means that large amounts of control information can be distributed, if it is done in large chunks, so that the overheads associated with message handling can be minimized.

We consider three types of network control elements that collectively implement the overall control system. *Terminal controllers* (TC) are associated with the individual user terminals and are the source of all requests for network services. Terminal controllers communicate their requests to *Network Controllers* (NC), which in turn carry out those requests by issuing commands to *Switch Controllers* (SC), which manage individual switches. Network controllers play the central role and are the focus of this paper. The NCs can communicate directly with one another, as well as with TCs and SCs. TCs and SCs communicate only with NCs.

An important component of the network control system is an underlying message transport mechanism that allows control messages to be reliably communicated among the various control elements. We assume such a component exists but don't discuss it in detail. We require that it be able to reliably deliver a message of arbitrary length to any specified destination in the network and return an acknowledgement indicating successful delivery. Failure to deliver a control message in a bounded time interval should be sufficiently rare that it is acceptable for such a failure to trigger failure of the entire operation associated with the message. We also require that the message delivery system deliver messages between two control elements in FIFO order.

We do not consider the question of network addressing in this paper, but do assume that every terminal is identified by an address which specifies a location (either logical or physical) in the network. Terminals may also have higher level names that can be converted to addresses by some separate process, but we don't consider those questions here.

In the remainder of this paper, we consider the design of control systems for ATM networks in some detail. Section 2 provides a formulation of the network control problem that is rigorous enough to allow one to make precise statements about the correctness, efficiency and use of resources by a network control system. This provides a framework for discussing network control algorithms and is perhaps the main contribution of the paper. Sections 3 through 5 discuss the issues associated with control systems for progressively larger-scale network configurations. In each section, we identify a particular point in the design space and explore the issues raised by the class of designs it represents. We do not provide complete solutions in any of these sections, but merely outline some of the possibilities and indicate how they might be fleshed out, and what the consequences would be. Section 6 summarizes the paper and discusses some of the issues that have been omitted from the main discussion.

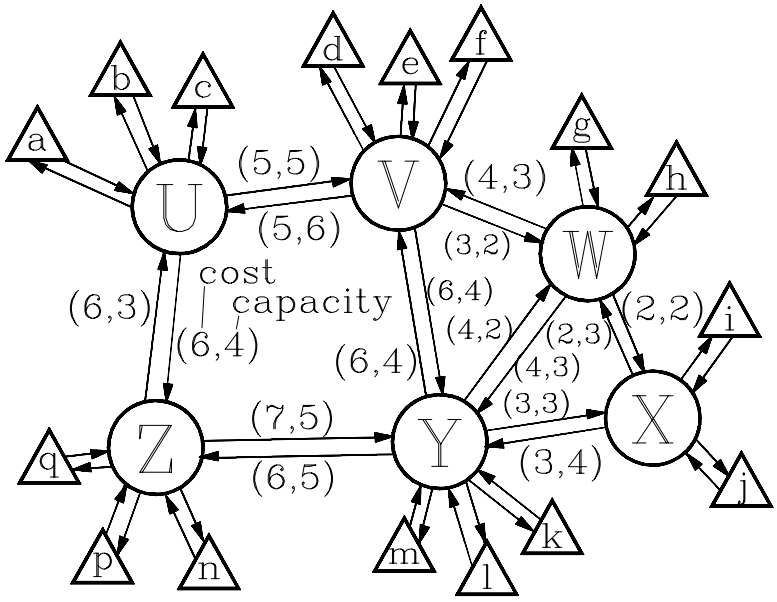


Figure 3: Example Network

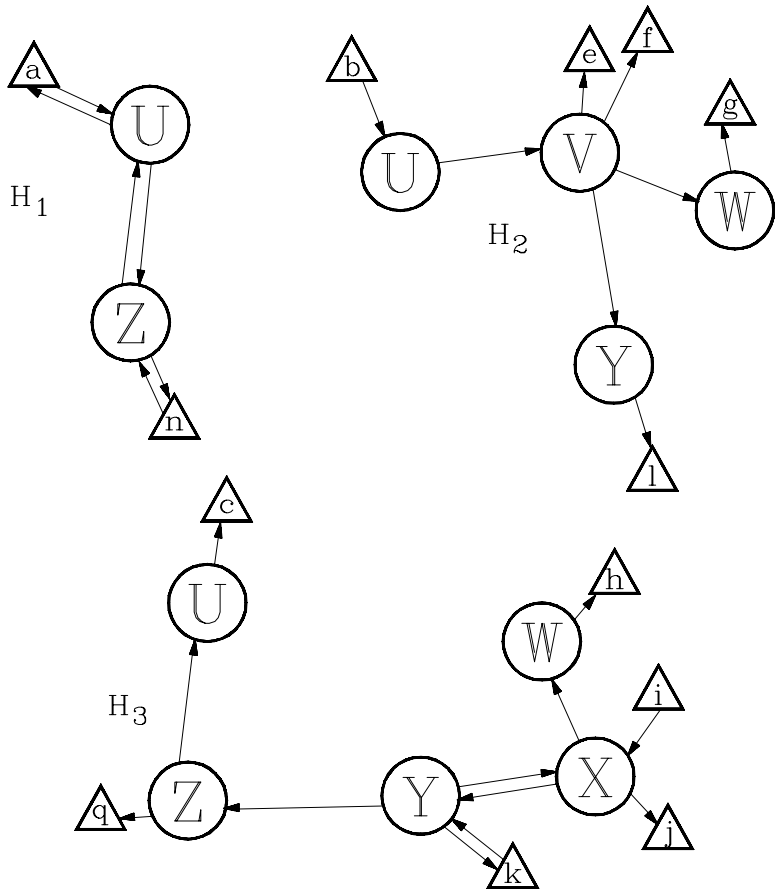
2. Formulating the Problem

In this section, we give an abstract model of the network control problem that will form the basis of our later discussion of alternative designs. The model suppresses a number of details that can be important in a practical system context, but retains enough of the real system issues to expose the essential complexities.

We model a network as a directed graph $G = (V, E)$ with integer edge costs $\gamma(u, v)$ and integer capacities $\kappa(u, v)$. Vertices which are incident to exactly one edge in the underlying undirected graph, are called *terminals*. All others are called *switches*. We let T denote the set of terminals and S the set of switches. These definitions are illustrated in Figure 3, where the terminals are shown as triangles and the switches as circles (edge costs and capacities are not shown for the edges incident to the terminals).

A *connection request* is a tuple $[c, \Sigma, \Delta, r, \omega]$, where c is a *connection identifier*, $\Sigma \subseteq T$ is a set of *sources*, $\Delta \subseteq T$ is a set of *destinations*, $r \in T$ is called the connection *owner* and ω is a non-negative integer called the connection *weight*.

Let $H = (W, F)$ be a subgraph of G whose underlying undirected graph is a tree, the leaves of which are in T. We say that a leaf u in H is a *source* if there is an edge leaving u. We say that a leaf u in H is a *destination* if there is an edge entering u. We say that H is a *connection graph* if for every pair u, v where u is a source and v is a destination, there is a directed path in H from u to v, and every edge in W is on at least one such source-destination path. A connection graph $H = (W, F)$ implements a request $[c, \Sigma, \Delta, r, \omega]$ if Σ is the set of sources in H and Δ is the set of destinations. A *connection descriptor* is a tuple $[c, S, T, r, \omega, H = (W, F)]$ where $[c, S, T, r, \omega]$ is a request and H is a connection graph that implements it. These definitions are illustrated in Figure 4.



Global Descriptors

[c1,{a,n},{a,n},a,1,H1]

[c2,{b},{e,f,g,l},b,3,H2]

[c3,{i,k},{c,h,j,k,q},k,2,H3]

Figure 4: Connection Descriptors

For a set of connection descriptors C and $(u, v) \in E$, define

$$\lambda_C(u,v) = \sum_{\substack{[c,\Sigma,\Delta,r,\omega,H=(W,F)] \in C \\ \text{such that } (u,v) \in F}} \omega$$

For example, if C is the set of connections in Figure 4, $\lambda(C(Z,U) = 3$. We say that C is *valid* if for all edges (u, v), $\lambda_C(u, v) \leq \kappa(u, v)$. A set of connection requests is *feasible* if there is a set of connection graphs for the set of requests that yields a valid set of descriptors. The cost of a set of descriptors C is defined to be

$$\sum_{(u,v)\in E} \lambda_C(u,v)\gamma(u,v)$$

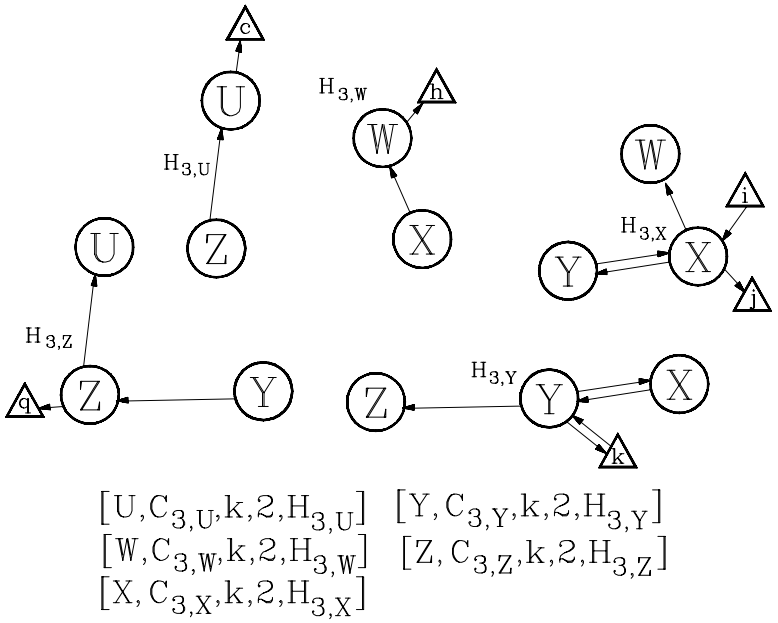


Figure 5: Local Connection Descriptors for Connection c_3

A set of connection descriptors is *optimal* if there is no other set of descriptors implementing the same set of requests that has lower cost. Note that the set of descriptors in Figure 4 is valid and optimal, but if we were to increase the weight of c_3 to 7, it would become both invalid and infeasible.

A *local connection descriptor at vertex* u is defined as a tuple $[u, c_u, r_u, \omega_u, H_u = (W_u, F_u)]$ where c_u is a connection identifier, $r_u \in T$, ω_u is a non-negative integer and H_u is a subgraph of G whose vertices are restricted to u and its neighbors in G. We say that a set of local descriptors implements a (global) connection descriptor $D = [c, \Sigma, \Delta, r, \omega, H = (W, F)]$, if it satisfies the following conditions

- for every $v \in W$, there is exactly one local descriptor $[u, c_u, r_u, \omega_u, H_u = (W_u, F_u)]$ for which $u = v$ and $c_u = c$,
- for every $v \in W$, the descriptor $[u, c_u, r_u, \omega_u, H_u = (W_u, F_u)]$ for which $u = v$ and $c_u = c$ also satisfies the conditions: $r_u = r$, $\omega_u = \omega$ and H_u is the subgraph of H induced by u and its neighbors in H.

Figure 5 shows the local descriptors for each switch involved in connection c_3 from the previous example. The connection descriptors for the terminals are not shown.

We define a *network control system* as an abstract data type that maintains a feasible set C of connection descriptors for a network G by operating on a set of local connection descriptors that collectively implement the set C. The abstract operations that are performed by the network control system are defined in terms of their effect on the global descriptors, but the system is required to realize these effects by operating on the local descriptors (this

is detailed further, below). Note that each of the operations listed below has an *invocation point* (denoted by the subscript x), which identifies the terminal controller which invoked the operation and which expects a response.

$create_x(u, \omega, typ)$ Adds a connection descriptor $[c, \Sigma, \Delta, x, \omega, H = (\{u\}, \{\})]$ to C, where c is an identifier not associated with any existing descriptor, ω is a non-negative integer, $typ \in \{src, dst, both\}$, $\Sigma = \{u\}$ if $typ \neq dst$ and empty otherwise and $\Delta = \{u\}$ if $typ \neq src$ and empty otherwise. Returns the value of c.

$destroy_x(c)$. Removes the connection with identifier c from C.

$invite_x(c, u)$ This operation simply results in a message being sent to $u \in T$, inviting u to join the existing connection c. An acknowledgement is returned to x when the message has been delivered.

$add_x(c, u, typ)$ where c is an identifier for some descriptor $D = [c, \Sigma, \Delta, r, \omega, H = (W, F)]$ in C, $u \in V - W$ and $typ \in \{src, dst, both\}$. This operation either does nothing and returns 0 or replaces D with a new connection descriptor $D' = [c, \Sigma', \Delta', r, \omega, H' = (W', F')]$, giving a new feasible set C' and returns 1. If the replacement is made

$$\Sigma' = \begin{cases} \Sigma & \text{if } typ = dst \\ \Sigma \cup \{u\} & \text{otherwise} \end{cases}$$

$$\Delta' = \begin{cases} \Delta & \text{if } typ = src \\ \Delta \cup \{u\} & \text{otherwise} \end{cases}$$

and H' is a connection graph that implements $[c, \Sigma', \Delta', r, \omega]$.

$remove_x(c, u)$ where c is the identifier of some descriptor $D = [c, \Sigma, \Delta, r, \omega, H = (W, F)]$ in C and $u \in \Sigma \cup \Delta$. Replaces D with a new connection $D' = [c, \Sigma - \{u\}, \Delta - \{u\}, r, \omega, H' = (W', F')]$, giving a new feasible set C', where H' implements $[c, \Sigma - \{u\}, \Delta - \{u\}, r, \omega]$.

$retype_x(c, u, typ)$ where c is the identifier of some descriptor $D = [c, \Sigma, \Delta, r, \omega, H = (W, F)]$ in C, $u \in \Sigma \cup \Delta$ and $typ \in \{src, dst, both\}$. This operation either does nothing and returns 0, or replaces D with a new connection descriptor $D' = [c, \Sigma', \Delta', r, \omega, H' = (W', F')]$, giving a new feasible set C' and returns 1. If the replacement is made

$$\Sigma' = \begin{cases} \Sigma - \{u\} & \text{if } typ = dst \\ \Sigma \cup \{u\} & \text{otherwise} \end{cases}$$

$$\Delta' = \begin{cases} \Delta - \{u\} & \text{if } typ = src \\ \Delta \cup \{u\} & \text{otherwise} \end{cases}$$

and H' is a connection graph that implements $[c, \Sigma', \Delta', r, \omega]$.

$reweight_x(c, \omega')$ where c is the identifier of some descriptor $D = [c, \Sigma, \Delta, r, \omega, H = (W, F)]$ in C and ω' is a non-negative integer. Either does nothing and returns 0 or replaces D with a new connection descriptor $D' =$

$[c, \Sigma, \Delta, r, \omega', H' = (W', F')]$, giving a new feasible set C', where H' implements $[c, \Sigma, \Delta, r, \omega']$.

Note that the owner of a connection is the terminal that invokes the initial *create* operation, but need not be an actual participant in the connection. This allows connections to be created on behalf of terminals that may be unable to invoke control operations independently. Ownership of a connection implies the right to control the connection configuration. A variety of policies can be defined relating to ownership. For example, one might restrict the *retype* and *reweight* operations so that they can be invoked only by the owner. One could include options for different connections so that an *add* operation is similarly restricted, allowed if the requester can supply a correct password or appears on an owner-defined list. Similarly, one could allow the owner to pre-specify restrictions on the *typ* of new participants in a connection. We do not consider these issues in detail here to avoid complicating the discussion. For the remainder of the paper, we simply assume no restrictions on which terminal controllers may invoke the various operations.

We now illustrate the operations with a few examples. Connection c_1 in Figure 4, would typically be constructed using the operation $create_a(a, 1, both)$, followed by $invite_a(c_1, n)$ and $add_n(c_1, n, both)$. Connection c_2 could be constructed using the operations $create_b(b, 3, src)$ followed by $add_g(c_2, g, dst)$, $add_e(c_2, e, dst)$, $add_l(c_2, l, dst)$ and $add_f(c_2, f, dst)$. The *add* operation can be extended to add a set of endpoints in one step, but we do not consider that option here.

The local connection descriptors are maintained by the switch controllers, which also make changes to the switches' hardware control tables necessary to provide the actual virtual circuits. The network control system implements changes to the local connection descriptors by sending messages to the switch controllers similar to those given above. We omit the definitions as they are fairly obvious analogs of the ones given above. We do note that a switch controller will carry out requests that, according to its local view, do not exceed any resource bounds (each switch controller maintains a record of the capacity of the edges incident to its switch). In any case, a switch controller returns a message indicating success or failure.

A network control system is called *incremental* if the connection graphs constructed by various operations are either supergraphs or subgraphs of the original connection graphs.

We say that a network control system is *sequentially correct* if given any set of local connection descriptors that implements a valid set of global descriptors, and any properly specified operation from the above set, it carries out the operation as defined above and returns. That is, at the time it returns, the set of local descriptors implements the new set of global descriptors required by the operation definitions.

Sequential correctness only means that a network control system will operate correctly when operations are presented one at a time (a new operation is requested only after a response is given to the previous operation). However, we are generally more interested in the case where requests are made concurrently at different vertices in the graph, and processing of different operations is carried out concurrently. We consider a network control system correct in this sense, if the set of local descriptors resulting from the concurrent execution of a set of operations is the same as would be obtained in some sequential execution.

Of course, the implementation of a network control system may also involve other data structures that allow various operations to be carried out more efficiently. Correctness of a particular system will also require that these data structures be consistent with the set of global descriptors. (For example, if there is a variable that records the link capacity in use between vertices u and v, its value should equal $\lambda_C(u, v)$, where C is the set of global descriptors.) However, since these data structures can generally be computed directly from the local descriptors, we won't address them separately.

Note that our definition of correctness defines a network control system to be correct, even if it never successfully completes any operation requiring allocation of resources. Such a system, while correct, is not particularly useful, so we generally require that in addition, a system be *responsive*. We say that a control system is *sequentially responsive*, if given any initial set of local connection descriptors that implements a feasible set of global descriptors, and an operation that requires the allocation of resources (*add, retype, reweight*), it successfully completes the operation if it is consistent with a stated *resource usage policy*. In the simplest case, we can make the resource usage policy simply: if there is a feasibly way to carry out the requested operation, then use whatever resources are needed. However, in practice it is often desirable to refuse an operation on policy grounds, if it requires excessive resources. For example, we might have a policy of refusing *add* requests if the only available path is so circuitous that it ties up excessive resources. Similarly, we might have a policy of carrying out a *reweight* or *retype* operation, only if they do not require changes in the vertex set of the connection graph. We say that a system is *responsive* if for every possible concurrent execution of a given set of operations, there is a sequential execution of the same set such that the subset of operations that completes successfully in the concurrent execution contains the subset that completes successfully in the sequential execution.

While one might argue that the separation of correctness and responsiveness is artificial, we find it a useful distinction to make. Correctness is a requirement for any network control system, while responsiveness is a matter of degree. So long as the governing policy is made explicit, it remains possible to evaluate and compare alternative network control systems, based on how responsive they are.

There are other criteria we're interested in when evaluating network control systems. One is run-time efficiency. There are different aspects to this. First, is the total amount of processing (including message transmission and computation) that is required to perform a given operation, and second is the response time for the operation (again, including the time associated with both computation and message transmission). In the case of response time, we are concerned with both the time required to carry out the operation in isolation and the time when the operation is executed concurrently with others. Both best-case and worst-case analyses are relevant here, since the best-case often reflects observed performance better than the worst-case. Another aspect of run-time efficiency is the effort that may be expended in background processing, not associated with any specific user operation. Such processing is generally required to speed up handling of user requests, but the resources devoted to such processing must be taken into account when comparing alternative approaches.

Another important criteria when evaluating a network control system is its effectiveness in managing the network's resources. For $\epsilon \geq 1$, we call a system *ϵ-conservative* if after completing any set of operations, the cost of the set of connection descriptors at that point

is no more than ϵ times the cost of the optimal set of connection descriptors. We say a system is *incrementally ϵ-conservative*, if the incremental cost of the resources allocated for any single operation are no more than ϵ times what would be allocated by an optimal incremental control system, starting from the same initial set of descriptors.

3. Centralized Control in a Campus Network

In this section we consider the design of a network control system for a campus network. This example illustrates one point in the design space and provides insight into some of the key design and performance issues. To make things concrete, assume that the network has 10,000 users who are served by 10 switches with 1,000 users each. Each user has an access link operating at a rate of 150 Mb/s and each pair of switches is connected by a link group comprising five links with a capacity of 600 Mb/s each. This gives each switch 27 Gb/s of capacity for communicating with other switches, or 27 Mb/s of non-local capacity for each user.

We start by exploring the simplest option for controlling such a system, which is to use a single network control processor, which can communicate directly with each terminal and switch controller. Let's first consider the essential processing requirements to get a rough understanding of the feasibility of a centralized design. To process a single operation, we'll typically need to do some resource allocation, send messages to one or more switch controllers and send a response to the user. Assume that an average operation requires that three messages be sent to switch controllers and that an average message has perhaps 200 bytes and requires 1000 instructions to do the low level message processing. This means that the network controller needs to execute 8,000 instructions to do the message processing for a single operation. If the resource allocation can be done in 2,000 instructions, then we have an estimate of 10,000 instructions to process an average operation. Assuming a processor with an effective instruction processing rate of 20 MIPS (about one fifth the peak rate of common processors today), this allows a single processor to handle about 2,000 operations per second or 12 operations per minute by each individual user. By contrast, telephone networks typically process fewer than one comparable operation every 10 minutes during the busiest part of the day, so the single central processor could handle more than 100 times the user request rate associated with users of telephone networks. While we expect users of multimedia ATM networks to generate more network control operations than users of telephone networks, it seems unlikely that it will be 100 times as many. This analysis suggests that if operating system overheads are kept to a reasonable level, that a single processor can effectively handle the control processing for a network this size and perhaps even larger networks. In general, if u is the number of users in a network, r is the average number of operations per minute by each user, P is the effective instruction processing rate of the control processor, M is the number of instructions that must be executed to process one message and A is the number of instructions that must be executed to implement the operation, then a single control processor can suffice if

$$\frac{P}{8M + A} \leq \frac{ur}{60}$$

So, for example, if $p = 50$ MIPS, $M = 500$, $A = 2,000$ and $r = 5$, we could conceivably support as many as 100,000 users with a single control processor.

Let's now take a more detailed look at the proposed system. With ten switches and five links joining each pair, we have a total of 225 inter-switch links. It's reasonable to expect that most point-to-point connections can be handled using a single inter-switch link, although we'll sometimes need to use a two link path. Using three links for a single connection seems wasteful, given the rich interconnection topology, so it may be a reasonable policy to allow point-to-point connections to pass through at most one intermediate switch. There are similar policy constraints that can be introduced for general multipoint connections. Define a connection to be type 1 if all the terminals connect to a single switch; type 2 if all the terminals connect to two switches; type 3 if the terminals connect to more than two switches and either all the source terminals are connected to one switch or all the destination terminals are connected to one switch; and type 4 otherwise. To avoid allocating an excessive amount of resources to any single connection, we require that type 1 connections use no inter-switch links, type 2 and 3 connections have no directed acyclic paths with more than two inter-switch links and type 4 connections have no directed acyclic paths with more than four inter-switch links. We call this the *bounded path length policy for small diameter networks*.

To implement the operation $create_x(r, \omega, typ)$, the network controller first checks to see if there is sufficient unused capacity on the edge incident to r (in the direction or directions required by typ), and if so, allocates the capacity needed, creates the required global connection descriptor and sends a message to the switch controller responsible for the switch connected to r, causing it to create a local descriptor. To implement $destroy_x(c)$, the network controller sends similar messages to all switch controllers that possess local descriptors for c, recovers the link capacity that has been allocated to the connection and destroys the global descriptor. To implement $add_x(c, u, typ)$, the network controller adds a branch between u and the connection using a path with the minimum possible number of hops, if there is such a path that is consistent with our bounded path length policy. This is done by searching its internal representation of the subgraph of the network that includes only the switches and the inter-switch links for the best path, allocating the required link resources, changing the global connection descriptor and sending *create* or *add* messages to the appropriate switch controllers. It can be shown that performing the routing in this way makes the system incrementally 1-conservative and 2-conservative if all link costs are equal. For a small network, like we're considering here (at least in terms of number of switches and inter-switch links), the identification of an appropriate route can be done very quickly. To implement $remove_x(c, u)$, the network controller prunes the portion of the connection tree used only by the endpoint u. This involves deallocating the resources that had been allocated to this branch of the tree, modifying the global connection descriptor and sending *remove* or *destroy* messages to the affected switch controllers.

To avoid disrupting the flow of user data, we're generally constrained to perform the various operations in an incremental way. For the *reweight* operation, this involves increasing or decreasing the amount of allocated capacity on all affected links, so long as all have sufficient capacity available. The operation will fail if any of the links lack sufficient capacity. There are similar constraints on the *retype* operation. Because these operations affect

more than a single path and don't allow selection of new routes, they can fail significantly more often than the *add* operation. This can justify some relaxation of the constraint that operations do not disrupt the flow of user data, but we do not consider that possibility in this paper.

Even though we only have a single network controller, there are scheduling and concurrency issues that can have an impact on the control capacity of the system. At one extreme, the network controller could do its processing in a strictly sequential fashion. That is, it would only process one operation at a time and always wait for a response from a switch controller before proceeding with its execution. While an implementation of this type may appear very constraining, it does have the merit of eliminating the complications of managing concurrent operations and the sometimes burdensome overhead that can be involved in switching among different contexts. If switch controllers can respond rapidly to connection requests (say in 100 μs or less) the amount of processing capacity lost due to idling while waiting for messages could be acceptable. In particular, if we consider our earlier example in which each operation requires an average of three messages to switch controllers, 1,000 instructions are executed for each message send and receive, and 2,000 instructions are required to implement the operation, we find that the number of operations that can be processed per second by a network controller with an effective instruction processing rate of 20 MIPS drops from 2,000 operations per second to 1,250. While this is a significant drop, it may still leave the controller with sufficient capacity to handle the given network configuration. Alternatively, the network controller might exploit some limited concurrency by sending control messages to switch controllers in parallel, rather than sequentially. This would increase the number of operations that could be done in a second in our example configuration, to 1,666.

The other extreme is to allow multiple operations to proceed concurrently, by creating a separate thread of execution for each operation. When there is a single central controller, we can make it easier to ensure correctness without sacrificing execution efficiency by constraining context switches among concurrent operations to occur only while a thread is waiting for messages. Since the network controller maintains complete information about the network state, it can ensure that messages to switch controllers will always succeed by first allocating resources in its local data structures and updating the proper global connection descriptor. This, together with our assumption of FIFO message delivery, is sufficient to ensure that concurrent operations execute as though they were sequential, even though they may be overlapped in time.

Our example also raises issues relating to how our abstract model reflects capacity allocation on links. In our model, we specify the capacity from one switch to another by a single number, but as our example configuration makes clear, this capacity may be provided by multiple physical links. This can lead to fragmentation in which there may be sufficient unused capacity on the set of links joining a pair of switches to accommodate a new connection request, but not enough capacity on any one physical link. In practice, individual connections must usually be assigned to a single physical link, and it is generally not practical to re-pack existing connections to reduce fragmentation (re-packing would disrupt the flow of data on the virtual circuits). The effect of this fragmentation is to reduce the effective capacity of a set of links. Since, in our example, the links connecting

to terminals have a capacity of 150 Mb/s each, a connection passing from switch to switch can be accommodated so long as at least one of the five 600 Mb/s links has 150 Mb/s of unused capacity. This implies that the effective capacity of the set of links is 2.4 Gb/s instead of 3 Gb/s. In general, if a single user request has a weight of at most B, then the *worst-case effective capacity* of a set of m physical links is equal to its raw capacity minus $(m-1)B$. This observation makes the simplification in our abstract model a reasonable one. However, the model can also be extended to explicitly model multiple links between every pair of switches. Since this does not lead to any fundamental changes, we retain the simplified model here.

4. Distributed Control with Global State

In this section, we consider a network which is too large to be controlled by single network control processor, but is still small enough to make it possible to distribute some global state information. Again, let's make the discussion concrete by assuming a specific configuration, with say 10 million terminals with 150 Mb/s links, supported by 1,000 switches with 10,000 terminals each. Assume that there are 100 additional *transit switches* for providing communication among the *access switches* and that each access switch has 100 links with a capacity of 2.4 Gb/s each connecting it to between one and five transit switches. Assume that each transit switch has 25 2.4 Gb/s links connecting it to each of about 40 other transit switches and has an average of 30 2.4 Gb/s links connecting it to each of about 30 access switches. With this configuration, each access switch has about 240 Gb/s of capacity for non-local traffic or about 24 Mb/s per terminal. Each transit switch has about 900 links connecting it to access switches and about 1,000 connecting it to other transit switches. Overall, there are about 3,000 link groups joining transit switches to access switches, and another 2,000 joining transit switches.

With this large a configuration, network control requires more processing capacity than a single processor can provide. Consider first, the case in which each switch has it's own dedicated network controller. To get a feeling for the processing capacity required, let's assume that users request an average of r operations per minute each, and that an average operation involves two access switches and two transit switches. With these assumptions, the network controller for each access switch must be able to process $333r$ operations per second and the network controller for each transit switch must be able to process $3,333r$ operations per second. For $r \leq 10$ a single processor can suffice for the network controllers associated with the access switches, if the amount of processing required per operation at each network controller is roughly the same as what we discussed earlier for the case of a single central controller. However, a single processor will clearly be insufficient for the transit switch controllers. On the other hand, ten processors can be sufficient, making it reasonable to control a transit switch using a single shared-memory multiprocessors. Memory requirements for the transit switch controllers could become a serious issue. If each connection requires M bytes of memory in each network controller involved in the connection, the average number of connections per terminal is t and the average number of transit switches involved in a connection is f, then the transit switches require an average

of $10^5 tfM$ bytes of memory. For $M = 1,000$, $t = f = 2$, this becomes 400 Mbytes, a large but manageable amount.

There are several ways we can approach the distribution of control. In this section we consider an approach that distributes the allocation of resources, but allows each individual connection to be controlled by a single controller. The idea is to have the access switch associated with the owner of a connection maintain a global picture of the connection, and all requests to operate on that connection are then handled by that access switch. To implement this, each operation request (except *create*) would be augmented with an additional parameter specifying the owner of the connection. A terminal, making an operation request, would send a message to the network controller for its access switch, and the controller would simply forward the message to the owner's access switch. The owner's access switch would implement the request, by sending messages to other switches, requesting the allocation or release of whatever resources are required.

In order for a network controller to allocate resources in response to an *add* request, it requires some knowledge of the state of the network. As discussed above, there are about 5,000 link groups in our sample configuration, a small enough number that we could reasonably maintain and distribute information about all 5,000 if the amount of information per link group is not excessive. Fortunately, very little information about a link group is really required to identify a path for an *add* operation. In particular, all we really need to know is if the link group has sufficient unused capacity to handle a new connection. If connections have a maximum weight of 150 Mb/s, a link group made up of 2.4 Gb/s links will be able to accommodate a new connection if less 93.7% of its capacity is allocated to other connections. Consequently, we would expect most link groups to be able to accommodate a new connection most of the time. Furthermore, once a link group becomes so busy that it can't accommodate a new connection, we would likely defer new connections until it has become at least a little less busy (say 85% of capacity allocated). These two considerations make it possible to use a single bit for the state of a link group, indicating it either is or is not accepting new connection requests. This state information will not change very rapidly, allowing it to distributed to all other switches in the network, either periodically or as changes occur.

Given a knowledge of network topology and the states of the various link groups, a switch can compute a path between any two points in the network. One way to perform the path computation is to carry out a standard shortest-path algorithm in response to each *add* request. However, the worst-case performance of typical algorithms make this expensive enough in a network with 5,000 link groups that alternatives need to be considered (even the most efficient implementations of Dijkstra's algorithm would require about 10–50 instructions per link group, or 1–5 ms on a processor with an effective processing rate of 50 MIPS). More efficient routing algorithms can be devised by taking advantage of the specific network structure. In particular, if we never use access switches as intermediate hops in longer connections and restrict ourselves to paths with at most links between transit switches (longer paths between transit switches can be ruled out for policy reasons, given the rich interconnection topology) we can speed things up considerably. For each pair of transit switches the number of other transit switches that they are both connected to will typically be less than 20, so it's reasonable to maintain a list of these intermediate switches

for each pair of transit switches, making it possible to check all two hop paths between a pair of transit switches in under 10 μs. Given this, we could check all possibilities between a pair of access switches that are each connected to three transit switches in under 100 μs.

From the above discussion, we can develop a picture of how each of the various network control operations might be implemented. First, each terminal requests that the network perform control operations on its behalf by sending messages to the network controller for the access switch it is connected to. The *create* operation is essentially the same as in the centralized control scenario, with the access switch connecting to the owner creating a global connection descriptor for the connection when it is created.

The *add* operation can be implemented by having the access switch first receiving the request, passing it on to the controller for the owner's access switch (call this the master controller). Since the master controller has a global view of the network and the connection, it can select an appropriate path, then send messages to the network controllers for the switches along the path, asking that they allocate the necessary link capacity and implement the connection in the switch hardware (the selection of the specific links and virtual circuits can be handled by the network controllers along the path). When they have either completed their individual operations or decided they cannot, the network controllers along the path will respond to the master controller, which can then either respond affirmatively to the requester or re-try the attempt with a different path. If the operation does not succeed after two or three attempts, the resources would be released and a failure indication returned to the terminal that made the request. The *remove*, *retype* and *reweight* operations can also be handled by the master controller in a straightforward way.

It is not hard to see how to implement the operations in a way that ensures that the local connection descriptors maintained by the various switch controllers remain consistent with the global descriptor maintained by each connection's master controller. However, the distribution of responsibility for resource management creates the possibility of resource management conflicts that can interfere with the system's responsiveness. Because resources required for an operation are released if not all the needed resources are immediately available, resource deadlocks are not possible. However *livelocks* are. In particular, concurrent *add* requests for different connections can interact in ways that prevent any of them from succeeding, even though the network may have the resources to handle some subset of them. Eliminating the possibility of livelock requires mechanisms for recognizing it and breaking the cycle of dependencies on which it depends. In this case, the likelihood of livelock is arguably small enough to make it reasonable to ignore the possibility altogether. We don't consider the subject further here, but note that it could conceivably be an important issue.

There is an obvious performance limitation of the network control system we have outlined. In the case of highly dynamic multipoint connections, the master controller could easily become a serious performance bottleneck. However, there is a simple extension that can drastically reduce the likelihood of this. The extension is to allow a network controller at an access switch to handle *add* and *remove* requests locally, without involving the master controller, whenever possible. Specifically, if a network controller at an access switch U receives a request to add one of its connected terminals to a connection that is already present at U, it could handle the request locally. This operation can be made transparent

to the connection's master controller, by omitting information about specific terminals in the connection from the master controller's global description. *Remove* operations could similarly be handled by the controller for the access switch associated with the target of the operation. The only time the master controller need be informed of an *add* or *remove* operation is when these operations cause the number of connected terminals at a given access switch to change from zero to one, or vice-versa.

Within the network controller for an access switch, we have the same issues with respect to scheduling and concurrency that we have in the case of centralized control. However, there are some new issues for the transit switches, since transit switches are controlled by a collection of shared memory multiprocessors. The network controllers for transit switches must maintain local descriptors for each connection passing through them, as well as detailed information about their local resources. The connection descriptors for individual connections can be assigned to specific multiprocessors based on the identity of the connection owner. Since transit switches get requests to operate on a connection through the network controller for the owner's access switch, this method of assignment also makes it easier to ensure FIFO message delivery. To manage resource information, simple locking mechanisms can be used to control access by different processors and ensure consistency.

5. Fully Distributed Control with Local State

The previous two sections considered networks that were small enough to allow at least some centralization of processing and some distribution of global network state. In this section, we focus on networks that are large enough that a more comprehensive distribution of processing and information is really required. Again, to make the discussion concrete, assume there are over 10^9 terminals, connected to access switches with an average of 10,000 terminals each. Each access switch connects to one or a few transit switches, using a total of 100 2.4 Gb/s access links, and the transit switches are each connected to about 50 others, with about 40 2.4 Gb/s links between each connected pair. With 10^9 terminals, there will be about 10^5 access switches and 10^4 transit switches. Assume the network topology is richly enough interconnected that we need never consider paths between transit switches with more than five links and that the average path between transit switches requires three links.

As in the last section, we assume a network controller for every switch, but given that the network is 100 times larger than in the previous case, it no longer seems reasonable to distribute any global network state that changes dynamically in response to individual user requests. So a key question becomes, how can we select paths to satisfy *add* requests, without global link status information, and without a complete picture of even a single connection? Let's first consider just the problem of selecting a path from a given point in the network to some destination. To allow this to be done quickly, we can provide the network controller for each switch with a table containing an entry for every possible destination switch in the network. Each entry consists of a list of neighboring switches through which the destination switch can be reached. Each list is ordered so that the shortest paths to

the destination through each successive neighbor in the list are increasing in length. To avoid routing loops, every neighbor in the list is closer to the destination than the network controller for which the table is configured. Given a set of tables like this, we can find a path to a given destination from some switch by consulting the switch's routing table, checking successive neighbors in the list for the destination and determining if there is sufficient available capacity along the connecting link group to accommodate the connection. If so, proceed to the neighboring switch and continue the search from there. In the event that none of the neighboring switches to a given destination has enough unused capacity on the connecting link group to accommodate the connection, one can either block the request or backtrack, with perhaps some limit on the total number of backtracking steps that would be allowed. While in general, it's difficult to say precisely how the paths produced by such an algorithm compare to the shortest path available, in a richly connected network, one would expect it to normally produce short paths, and to do so far more quickly than any method that guarantees shortest paths. In fact, one could probably design a network topology to ensure that paths produced by such an algorithm were never too much longer than the shortest path and that the failure of the algorithm to find a path meant that no 'short enough' path was available.

There is a natural generalization of the routing technique described above that could improve its performance. Instead of using a list of best neighboring switches to each destination, one could use a list of best switches within two hops of a given switch. If we also distributed link state information to switches within two hops, we could effectively look ahead one hop, before deciding which switch to go to next. The technique can clearly be extended to provide greater amounts of look-ahead, but the small additional improvement that is likely to be obtained may not compensate for the significant expansion in memory needed for the routing table and link state information.

The next issue to consider is the distribution of information about connections. With over 100,000 switches in the network, a single connection can be too large to reasonably manage from a central location. Hence, we also need to consider the question of distributing connection information. In this section, we take the extreme view of having each network controller maintain only the local connection description for each connection that passes through its switch. This immediately raises some questions with respect to routing, since no network controller has enough information to determine the best point to branch off of the connection in order to reach a new endpoint. The problem becomes simpler if we satisfy an *add* request by starting the path search from the new endpoint and go toward some known location in the connection, halting the search as soon as the connection is reached, even if we have not yet reached the target location. This implies that in order to process the *add* request, we need a target (or perhaps a selection of multiple targets) to aim for. We could require that the terminal making the request supply one or more targets, along with the connection identifier and weight. In common cases, the user may have this information anyway. For example, the user could have it as a result of receiving a prior *invite* message, or in the case of connections used for public information distribution, through secondary sources (e.g. T.V. Guide). In addition to these means, the network can provide information services that supply such auxiliary information, given a connection identifier. This is analogous to the use of directory assistance in the telephone network and

would be used only when needed to avoid making it a performance bottleneck. The load on the information service can be distributed over multiple servers by mapping some portion of the connection identifier to the address of a server responsible for maintaining information about that connection.

From the above discussion, we can develop a picture of how each of the various network control operations can be implemented. As in the previous section, each terminal requests that the network perform control operations on its behalf by sending messages to the network controller for the access switch it is connected to. The *create* operation is essentially the same as in the centralized control scenario. However, now the connection may have to be registered with an information server. The need for registration can be indicated by the owner, in an additional parameter in the *create* operation, or can be implemented as a separate operation, allowing the owner to decide to register or 'de-register' a connection at any time.

As discussed above, we assume that additional parameters are included in the *add* operation, specifying the weight of the connection and the identity of at least one target location in the connection. The access switch receiving the original operation request forwards the request to the access switch for the requested new endpoint (call it U). U first determines if the requested new endpoint has sufficient capacity on its access link, then selects one of the targets and starts hunting for a path to that target, as described above. Assuming the path hunt reaches the connection at some switch V, the path can be retraced with resources along the path being committed to the connection and the switch hardware configured to implement the new connection branch. At this point, an acknowledgement can be sent to the requesting terminal.

Remove operations are performed similarly to *add*, proceeding from the endpoint to be removed along the connection tree until a branch point is reached and terminating there. *Retype* can be done similarly, proceeding from the terminal whose type is to be changed, allocating new resources along the path if possible and deallocating resources no longer needed. *Reweight* proceeds through the entire tree, increasing or decreasing the allocated link capacity as appropriate, and failing if the required capacity is not available everywhere.

With the execution of connection operations being distributed across hundreds or even thousands of network controllers, there are clearly myriad opportunities for errors in algorithms to create inconsistencies in the local connection descriptors when operations are performed concurrently on a single connection. We could simplify the problem, as in the earlier sections, by forcing operations on a single connection to be done serially. This need not require centralized processing, but does at least require the acquisition of a lock from a central location before proceeding to modify the connection. Given that some connections may be very large and highly dynamic, it seems preferable to avoid such an extreme solution if possible. A natural possibility to consider is allowing operations to proceed concurrently in different parts of the connection tree and only force them into a sequential order when they affect information at a common set of switches. This can be implemented by creating a thread of control for each distinct operation some part of which, is being performed at a given network controller; but only allowing one active thread at a time, with respect to a single connection. This helps ensure consistency, but can introduce the possibility of deadlock, since threads for two or more operations can be queued waiting for one another

in a cyclic fashion. We can break the deadlock by imposing an arbitrary global ordering on the set of operations being performed on a connection and allowing operations that come first in the global ordering to proceed in advance of operations that come later in the global ordering. (One simple way to do this is to assign each operation a unique global identifier when it first starts.) Of course, this also means that we must be able to rollback any operation, before it has completed.

There is an intermediate possibility for controlling concurrency in a connection that can also be considered. First, one divides the network into regions of moderate size ($\leq$ 1000 access switches and $\leq$ 100 transit switches). Within each region, each connection is controlled by a single network controller, whose address can be determined from the connection identifier (possibly by hashing the connection identifier to produce an index into a table of network controllers for the region). Link state information could also be distributed throughout such a region. Operations that are local to a region can be carried out as described in the previous section. To route connections that leave the region, we can define for each destination in the network, a list of switches neighboring the region that are on short paths to the desired destination. The controller for a connection within a region could also keep track of which other regions the connection passes through. This can be done economically using a simple bit vector for networks with up to a few hundred regions. Even for highly dynamic connections, these region sets would generally change slowly enough that distributing them to all the region controllers need not be particularly expensive.

6. Closing Remarks

This paper has presented a framework for precisely describing and analyzing distributed algorithms for the control of ATM networks and explored several points in the space of possible designs. It is hoped that this will provide a useful introduction to the network control problem for researchers in distributed systems and a vehicle for more carefully defining and comparing alternative system designs.

In order to focus on the issues of distributed control, we have glossed over issues relating to resource management. In particular, we have assumed that the data rates of individual virtual circuits could be adequately represented by single numbers, and we have ignored the case of adaptive data traffic in which the application adapts its rate to match the availability of network resources. For some applications (e.g. coded video), there is enough variability in the traffic rate that simply allocating the peak rate leads to considerable inefficiency in the use of network resources. If the link rate is much larger than the peak rate of the virtual circuit (>50 times, say) then it's possible to allocate an amount that is slightly larger than the average data rate with high confidence that the aggregate traffic will not exceed the link capacity. However, often determining if a given virtual circuit will 'fit' on a given link requires a more complex computational procedure and more information about the traffic currently flowing on the link than a simple aggregate rate. When a network controller for a particular switch must decide if a given virtual circuit can share a link with other virtual circuits already using it, it must perform this more complex calculation. The question that

then arises is must we do the same calculation when trying to determine a path to reach a given destination, and if so, what link state information must be distributed. In situations where the capacity of link groups is much larger than the capacity of a single virtual circuit (which is inherently true in large-scale systems), a simple representation for the state of the link group can suffice, since a representation that leads to conservative decisions will have only a small impact on the efficiency with which network resources are utilized in this case. It may be desirable to use more than a single bit per link group, but it does not appear that complex representations are necessary or provide any significant advantages. Similar observations apply to adaptive data traffic, but it may be necessary to have separate state information for adaptive traffic than for reservation-oriented traffic.

Multicast routing is another issue that was touched on only lightly in the body of the paper. This is a major subject in its own right and has been discussed in a number of papers [18, 19]. For the kind of dynamically changing connections considered here, the only practical choices that have been identified are variations on a simple greedy algorithm that adds new endpoints using a shortest path from the endpoint to the connection, and deletes endpoints by pruning the branch needed only by the endpoint being dropped. This is the general type of algorithm considered here, although others are certainly possible within the context of the same general framework for distributed control. While the worst-case performance of these algorithms can be poor, simulations provide evidence that they should work well in practice [18, 19].

It's important to note that routing and network topology design are very inter-related. While it can be useful to consider them separately, one cannot really understand how routing algorithms are likely to perform without also understanding something about how networks are configured. ATM networks introduce some new issues for network design that have not been widely studied and are directly relevant to the question of routing in ATM networks. While such questions are beyond the scope of this paper, the interested reader will find a good introduction to these issues in [10].

There are other approaches for how to distribute control in a network that we haven't touched on. In general, all the control algorithms we have considered rely on *structural partitioning* to define regions of the network controlled by different entities (in some cases these regions are single switches). Another way to divide the responsibility for resource management is *layered partitioning* in which each of a number of network controllers is responsible for a 'layer' that spans the entire network. These could take the form of separate spanning trees or simply dividing each link group among different controllers. Layered partitioning has the advantage that it allows all the resource allocation for an operation to be handled by a single processor in most cases. Another option is to not to divide up the responsibility in any fixed way but allow different network controllers to directly allocate whatever their resources they need, using fine-grained locking mechanisms to prevent them from interfering with one another.

Yet another way to manage the distributed control in large networks is to organize the control in a hierarchical fashion. This is of course the classical approach used in telephone networks and has also been suggested in recent proposals for ATM network control [9, 21].

This paper does not consider the full range of operations that might be performed on

a connection. In particular, it does not allow multiple connections to be operated on as a group and possibly constrained to follow common paths in the network, a feature that is desirable when using several connections to support multimedia applications with separate connections for audio and video. While this does require some additional mechanisms, it does not change the problem at any fundamental level. We have also not considered variations of the *add* operation that allow sets of endpoints to be added at once, or operations that allow two connections to be combined to yield a larger connection. Nor have we considered how multicast connections can be automatically reconfigured when links fail. All of these would be worthwhile extensions, and the basic framework could certainly be augmented to handle them.

In summary then, this paper offers an approach to specifying and analyzing network control systems. While we strive for precision, we have avoided formal notations, preferring instead the level of description generally used in defining and analyzing algorithms in the technical literature. Within this framework, we have explored the central design issues for ATM network control systems through a series of examples. It is hoped that this will provide a useful foundation for other researchers in networking and distributed computing who are interested in understanding these issues and developing more complete solutions.

References

[1] Appenzeller, Hans. "Signaling System No. 7 ISDN User Part," *IEEE Journal on Selected Areas in Communications*, 5/86.

[2] Bettati, R., D. Ferrari, A. Gupta, W. Heffner, W. Howe, M. Moran, Q. Nguyen, R. Yavatkar. "Connection Establishment for Multi-Party Real-Time Communication," *Proceedings of NOSDAV*, 1994.

[3] Bubenik, Richard, John DeHart and Mike Gaddis, "Multipoint Connection Management in High Speed Networks," *Proceedings of Infocom*, 4/91.

[4] Coudreuse, J. P. and M. Servel. "Asynchronous Time-Division Techniques: An Experimental Packet Network Integrating Videocommunication," *Proceedings of the International Switching Symposium*, 1984.

[5] Coudreuse, J. P. and M. Servel. "Prelude: An Asynchronous Time-Division Switched Network," *International Communications Conference*, 1987.

[6] DeHart, John. "CMAP/CMIP Scenarios: A Tutorial," Washington University Computer Science Department, ARL-90-03, 1990.

[7] Dehart, John. "BPN Connection Management Access Protocol Specification," Washington University Computer Science Department, ARL-89-06, 1989.

[8] DeHart, John and Dakang Wu. "Connection Management Network Protocol (CMNP) Specification," Washington University, Computer Science Department, ARL Working Note, 11/93.

[9] Dykeman, Doug (editor). "PNNI Draft Specification," ATM Forum 94-0471R7, 1994.

[10] Fingerhut, J. A. "Approximation Algorithms for Configuring Nonblocking Communication Networks," Doctoral Dissertation, Washington University Computer Science Department, 5/94.

[11] Harman, Wendy and Cheryl F. Newman. "ISDN Protocols for Connection Control," *IEEE Journal on Selected Areas in Communications*, 9/89.

[12] Haserodt, Kurt and Jonathan Turner. "An Architecture for Connection Management in a Broadcast Packet Network," Washington University Computer Science Department, WUCS-87-3, 1987.

[13] Kearns, Timothy and Maureen C. Mellen. "The Role of ISDN Signaling in Global Networks," *IEEE Communications Magazine*, 7/90.

[14] Kulzer, John J. and Warren A. Montgomery. "Statistical Switching Architectures for Future Services," *Proceedings of the International Switching Symposium*, 5/84.

[15] Minzer, Steven and Dan Spears. "New Directions in Signaling for Broadband ISDN," *IEEE Communications Magazine*, 2/89.

[16] Turner, Jonathan and Leonard Wyatt. "A Packet Network Architecture for Integrated Services," *Proceedings of Globecom 83*, 11/83.

[17] Turner, Jonathan S. "The Challenge of Multipoint Communication," *Proceedings of the ITC Seminar on Traffic Engineering for ISDN Design and Planning*, 5/87.

[18] Waxman, Bernard. "Routing of Multipoint Connections," *IEEE Journal on Selected Areas of Communications*, 12/88.

[19] Waxman, Bernard. "Performance Evaluation of Multipoint Routing Algorithms," *Proceedings of Infocom*, 3/93.

[20] Wu, Dakang. "A Survey of Network Signaling," Washington University Computer Science Department, WUCS-95-08.

[21] Wu, Dakang. "An Efficient Signaling Structure for ATM Networks," Washington University Computer Science Department, WUCS-95-08. '