The Design and Evaluation of Routing Algorithms for Real-time Channels

Ron Widyono
widyono@cs.Berkeley.EDU
Tenet Group
University of California at Berkeley, &
International Computer Science Institute
TR-94-024
June 1994

Abstract

The Tenet Scheme specifies a real-time communication service that guarantees performance through network connections with reserved resources, admission control, and rate control. Within this framework, we develop and evaluate algorithms that find routes for these multicast connections. The main goals are to maximize the probability of successful establishment of the routed connection, to maximize the useful utilization of the network, and to be timely. The primary problem to be solved is finding a minimum cost tree where each source to destination path is constrained by a delay bound. This problem is NP-complete, so heuristics based mainly on minimum incremental cost are developed. Algorithms we develop use those heuristics to calculate minimum cost paths that are then merged into a tree. We evaluate our design decisions through simulation, measuring success through the number of successfully established connections. These experiments demonstrate that our evaluation methodology is a useful tool for the development of robust routing algorithms for real-time channels.

1.0 Introduction

The growing speed of wide-area networks has spawned much interest in new kinds of communication-based applications, which require more predictable performance from the network. The Tenet Group at the University of California has been addressing the problem of providing a communication service with real-time guarantees [FeBaZh92]. The initial design, known as Scheme 1, provides a communication abstraction called a *real-time channel*, which is a simplex unicast end-to-end connection with performance guarantees and restrictions on traffic. Currently under development is Scheme 2, the next generation of algorithms which will add support for multi-party communication and for a more flexible client-service interface [Hef94]. To this end, the real-time channel abstraction has been redefined as a *multicast* (one-to-many) connection.

A basic requirement for the establishment of real-time channels is a route between the source and destination(s). An algorithm that calculates such a route needs to consider that a client will desire a high probability of success in channel establishment, that the route will make efficient use of network resources, and that the routing calculation should be done in a timely manner. This paper describes the development of such algorithms, concentrating on the motivations and evaluation of our design choices. The primary goals of the project are to provide algorithms for a subset of the requirements of Scheme 2[1], and more importantly, to provide a framework for the development of more robust algorithms.

This paper is organized as follows: Section 2 describes the details of the Tenet Schemes; Section 3 explores the design of the desired routing algorithm; Section 4 develops various routing algorithms that satisfy our requirements; Section 5 describes our evaluation methodology, which includes an analysis of simulation experiments; and Section 6 concludes the paper.

2.0 The Tenet Scheme

[FerVer90] shows that performance can be guaranteed in a connection-oriented network by reserving adequate resources and performing admission control at channel establishment time, and by enforcing rate control during data delivery. The Tenet Scheme (set of algorithms) uses these concepts to provide a network level real-time communication service to its clients (entities at the transport level and above), and its main features are described in this section. The routing algorithms in this paper will be based on a subset of this framework.

2.1 Service Definition

The service definition is the interface that the network offers to its clients. A client may request the establishment of a simplex multicast channel, which has a single source and an arbitrary number of destinations. The client must characterize the data traffic that will flow through this channel throughout its lifetime using the following parameters [Fer90]:

- X_{min} - the minimum packet inter-arrival time
- X_{ave} - the worst-case average packet inter-arrival time

1. The design of Scheme 2 is work-in-progress, so we did not address features that are not yet mature. Furthermore, to simply the analysis, we did not address some of the performance requirements.

- I - the averaging interval
- S_{max} - the maximum packet size

These parameters are a measure of the peak and average load as well as an indication of their burstiness. The client should also specify the performance requirements for each destination using the following parameters:

- D - the maximum end-to-end delay permissible
- J - the maximum delay jitter (or variation in delay)
- Z - the minimum probability that the delay of a packet is smaller than the delay bound, D
- W - the minimum probability of no buffer overflow along the route of the channel

When the requested channel is established, the network guarantees that it will be able to handle the throughput specified in the traffic parameters and to deliver the data to each destination within its performance requirements. This guarantee is binding only so long as the actual offered load stays within the bounds of the traffic specification.

2.2 Real-time Guarantees

The network delivers its guarantees through resource reservation, admission control, and rate control. Given the traffic specification, the performance requirements, a route, and a scheduling discipline (see Section 2.3), the network can calculate how much of each resource needs to be reserved at each node along the route so that the performance requirements can be met. During the admission control phase of channel establishment, each node accepts or rejects the channel depending on whether or not it has the resources required of it. If every participating node accepts the channel, the channel is considered established, and the resources required are reserved. As data flows through the channel, the scheduling discipline at each node ensures the timely processing and dispatching of packets, and it protects the data stream against best-effort traffic. Rate control is required at each node to protect against clients who either unintentionally or maliciously exceed their traffic specifications, a condition the scheduling discipline is not usually equipped to handle gracefully.

2.3 Scheduling Discipline

The scheduling discipline used in this paper is EDD (Earliest Due Date) [FerVer90], although any starvation-free discipline could be used with the Tenet Scheme [ParFer93]. Identifying the scheduling discipline is important because it affects the way network resources are characterized and reserved, which in turn determines the admission control tests.

2.4 Network Resources

For the EDD scheduling discipline, the following are the network resources (modeled on a per-network-link basis):

- Processing Power—The network hardware's ability to process and dispatch packets before the next packet from this channel arrives (in our case, the throughput of the network interface). *Node Saturation* occurs when processing power is insufficient to service the worst-case traffic from all established channels on a link.

Admission control prevents the establishment of a new channel when doing so results in node saturation. As an example, assume that a link is already processing packets arriving once every millisecond and that the processing time for such packets is half a millisecond. Then the link would reject a new channel delivering packets once a millisecond if each packet requires more than half a millisecond of processing time.

- Delay Bound—The longest delay at the current link that a packet can tolerate without missing its end-to-end deadline. *Scheduler Saturation* occurs when the link is unable to provide delay guarantees for all established channels. For example, if packets from two channels arrive simultaneously, if both packets must be transmitted within 80 microseconds, and if it takes 40 microseconds to process one of the packets and 50 microseconds to process the other, then it is not possible to meet both deadlines.

  Even though a test for scheduler saturation exists, no admission control test is performed in each intermediate node. Rather, a minimum delay bound is calculated such that scheduler saturation will be impossible. This approach is possible because delay is theoretically an infinite resource. Practically, however, large minimum delay bounds in individual intermediate nodes will lead to a failure of the end-to-end delay bound test.

- Buffer Space—The buffer space required to hold the packet on the channel, for the duration of their local delay bound. *Buffer Saturation* occurs when the node runs out of available buffer space.

2.5 Channel Establishment

The establishment protocol consists of a single round-trip pass from the source to all the destinations[2]. During the forward pass, each node will perform its admission control tests, and, if they succeed, resources are reserved, and the request is forwarded on to the next node(s). When the request arrives at a destination, tests are performed to ensure that the end-to-end performance requirements can be met. As an example, the cumulative delay bounds between the source and a destination are compared with the maximum delay bound the destination can tolerate (as specified by the client).

After a successful forward pass, complete knowledge has been gathered about the availability of resources along the source-destination path. Therefore, on the reverse pass, intermediate nodes are allowed to relax their resource reservations to a level that is better for network efficiency, yet does not allow the client performance bounds to be violated. Once the source receives notification of success or failure of the establishment, it can begin sending data on the channel. Note that this scheme allows for partial failures, in which some destinations will receive data while others will not.

2.6 Simplifications

To reduce the number of factors in our evaluation, we have chosen to concentrate on and show our motivations for only the bandwidth requirements and the delay bound performance guarantee D, which we expect to be the most critical performance requirement. Designing for the remaining parameters will follow similar lines of thinking.

2. This is true in Scheme 1 and for the subset of Scheme 2 with which we are concerned. Join/Leave operations and third-party initiated channel establishments may require more than two one-way transmissions.

3.0 Design Issues

In this section, we explore some of the motivations for this work.

3.1 Routing Algorithm Goals

The Tenet Scheme provides a real-time communication service to a community of users that will over time request a large number of real-time channels. From a client's perspective, the service is effective if it not only does its best to accommodate each individual request, but also to maximize the useful utilization of the network.[3] Since the success of the service relies mainly on the ability to allocate network resources, one of the Scheme's primary tasks is to efficiently manage limited network resources. With a connection-oriented approach, and especially with the Tenet approach, the onus is mainly on the channel establishment process to allocate resources wisely and fairly; once a channel is successfully established, and barring a catastrophic failure, the client can be assured that its performance requirements will be met.

When a channel establishment fails, it means the network's best attempt failed, and the client is denied service. It would be undesirable then to declare a failure if a path does exist somewhere in the network. On the other hand, if the client's request truly cannot be accommodated given the current state of the network, the establishment system is not necessarily blameless. The unavailability of resources is largely due to previously established channels, and inefficient use of resources by those channels would be directly attributable to their establishment.[4]

The channel establishment system can affect the efficiency of a channel during both the forward pass and the relaxation phase on the reverse pass (see Section 2.5); however, the routing algorithm has a major impact on resource management as well. It is the route that largely determines the success or failure of the establishment as well as the level of resource consumption. With this in mind, we have identified the following goals for the routing algorithm.

3.1.1 Maximize the probability of a successful establishment

We would like to ensure that the current channel being routed has the best chance of being established. Under the environment described in Section 2.0, it is clear that it is the admission control process that determines whether or not an establishment succeeds. A rejection by one of the intermediate node tests implies that some of the required resources were not available; therefore, the routing algorithm needs to be able to locate available resources within the network. A rejection by one of the destination tests implies that the route chosen for that destination is too long in terms of delay[5]—that is, the sum of the delay bounds calculated at the intermediate nodes (the end-to-end delay) exceeds the client specified limit. For this case, the routing algorithm should be able to avoid such routes, ideally by accurately estimating their end-to-end delay.

3. A client may look at maximum utilization of the network as a noble ulterior motive, but, more likely than not, there will be a cost, usually financial, associated with the resources. It is sometimes only this cost that prevents malicious clients from reserving the entire network to achieve their real-time goals.

4. Being inefficient means providing the requested throughput and performance guarantees using more resources than necessary. It is always possible for a client itself to use a channel inefficiently.

5. In this paper, end-to-end delay is the only requirement considered in the destination tests. The full Scheme includes end-to-end delay jitter, end-to-end delay bound violation probability, and end-to-end buffer non-overflow probability.

3.1.2 Maximize the probability of future establishments

For successful establishment of the current channel, the routing algorithm has no special requirements of the resources it needs to locate, other than its ability to meet the reservation requirements of the channel. However, as explained above, a channel should be as efficient as it can in its use of resources, so as to maximize the probability that a future channel will be successfully established given an optimal route. In our environment, being inefficient means using more bandwidth (links) than necessary or contributing to a hot-spot.

Because we have to view the network as a finite pool of resources, using more links than necessary reduces the overall availability of resources in the network and thus reduces the overall channel throughput that can be supported. Since each link of a channel consumes an equal amount of bandwidth, we simply would like to minimize is the total number of hops in the channel's route.

A second efficiency consideration is the localized limitation on the alternative paths a route can take. These roadblocks can be topological (for example, a sending node must use one of the generally small number of links emanating from that node), or they can be due to resource saturation from previously established channels. Unfortunately, identifying and prioritizing such limitations is likely an intractable proposition. Even if identifying potential hot-spots were feasible, determining their likelihood of becoming a bottleneck would require the prediction of future channel requests. At the very least, the route calculation should assume a uniform distribution of channel requests and should distribute its consumption of resources evenly among the alternative paths (by choosing links furthest from saturation). A possible improvement is to use recent usage patterns as a predictor of usage in the near future, since a channel will affect only future channel requests made during its lifetime.

3.1.3 Route in real-time

The routing algorithm should perform its computation in "real-time." Here the notion of timeliness is not in the same scale as that for the real-time data transfer itself, but it may nonetheless be within a few orders of magnitude. A client requiring a real-time channel will presumably be executing in real-time and will not tolerate large delays in setting up its communications. Timeliness is even more critical for a client that has tight time constraints between the time it identifies what kind of channel it needs (destinations, QoS requirements, etc.) and the time its data should be delivered. We will keep an eye on the running times of the algorithms we develop.

3.2 Source (Centralized) Routing

The routing algorithm will be designed for source (or centralized) routing. Note that, in a connection-oriented environment, source routing refers to the routing of the connection establishment rather than of the data, and it contrasts with distributed route computation during connection establishment. The computation is centralized at the source of the channel, at one of the destinations, or even a third-party node. There are several advantages to source routing [Moran93]:

1. It is easier to control construction of the tree. One of the primary functions of the routing algorithm is to constrain the end-to-end delay of the paths. A constrained shortest-path algorithm will inherently do a lot of retracing and revising of paths, which is much more easily done on a centralized map than through some distributed process.

2. It is easier to avoid routing loops. Regardless of how stale the view of the network is, a central decision maker can easily avoid creating a routing loop. No matter how perfect a decision a node makes in a distributed computation, the information on which it based its decision may be stale by the time the next decision is made by another node.
3. It is easier to manage policy decisions. The Tenet Scheme is intended for use in internetworks, where routing will inevitably have to deal with administrative domains that want to restrict traffic through them.
4. It is easier to handle sharing. The routing algorithms will inevitably have to handle sharing relationships between channels (e.g., if two or more sources agree to alternate use of the network, such as in a multi-participant conference, any links their channels have in common only need to reserve the resources for the more demanding link), and centralized management of such policies is easier.

There certainly are drawbacks to source routing; the primary one—maintenance of network state knowledge—is discussed next.

3.3 Network State Knowledge

It is clear that the success of the routing algorithm is predicated on having an accurate view of the network's state, including its topology and the availability of resources at every node (but not the instantaneous load on the network). With source routing, the staleness and granularity of this view become an important issue.

A distributed routing computation during channel establishment would presumably occur on the nodes with the resources, and its decisions would become effective immediately on those nodes. With centralized computation, however, network state from every node needs to be distributed to all nodes that might potentially calculate a route. Even with gigabit speed networks, this dissemination will take time, and, during that time, the state of the network will invariably change. Secondly, the amount of network bandwidth required to distribute a perfect view of the network would be enormous, and so it is necessary to summarize some of the state information. Finally, a centralized route computation's decision affecting a node would not be seen by the node until the propagating channel establishment arrives. These aspects of a centralized computation result in routing decisions based on stale and summarized information; furthermore, even an accurate route can grow stale as a channel establishment progresses.

3.4 Scalability

Several of our design decisions have scalability implications. The most important one is the need to distribute network state. If every node in the network could potentially compute a route, or even if a subset proportional to the size of the network had that responsibility, the bandwidth required to disseminate network state from every node would grow exponentially with the size of the network. In addition, the disseminated information has to be stored somewhere, so storage requirements grow as the network grows. Since source routing requires that an entire route be transmitted with the establishment request packet, the size of that packet could grow unacceptably large with the size of the network. Finally, we will see that the routing algorithms (and their timeliness) are sensitive to the size of the network and to the number of destinations in a channel.

The long-term plan to deal with scalability is to limit routing tasks to routing server nodes within a hierarchical routing scheme. For this project, we do not explore any aspects of hierarchical routing.

4.0 Algorithm Development

In this section, we describe the various issues visited during the development of our algorithms.

4.1 Assumptions

We made the following assumptions about the environment in which we are designing our algorithms:

1. The network can be modeled as a graph of nodes connected by point-to-point links. Each link has a bandwidth value and a propagation delay, and these values are known to the routing algorithm. There can be no more than one direct link between two nodes.
2. Each node can be modeled as having an input and output queue on each network link. Each link has its own processor that can handle the full bandwidth of the link. Each link also has buffer space dedicated to it. The processing of packets in one link never affects the processing in another link. The CPU (if there is one) is never a bottleneck.
3. At end-nodes, we only model the path between the physical layer and the network layer, as guarantees at and above the transport layer are affected by end-system considerations that have nothing to do with the routing algorithm.
4. Some view of the state of the network's resources (with some level of staleness) is provided to the routing algorithm. These resources are on a per-node/link basis, so their values are best stored in a directed graph. The following are the resources parameters:
    - β = Residual Bandwidth
    - δ = Estimated Delay Bound
    - ψ = Residual Buffer Space

   The Residual Bandwidth represents the unused processing power on that link; the Residual Buffer Space represents the unused buffer space; and the Estimated Delay Bound is a very rough estimate of the minimum delay bound that the next channel through this link will experience. Note that this delay bound value varies largely with the channel's maximum packet size, which affects its service time.
5. The output of the routing algorithm should be a proper tree. Nothing precludes the data forwarding mechanism from differentiating between the branches of a channel and allowing them to cross. However, this is not generally done, and so our algorithms must do what they can to return a tree.

4.2 Multicast Trees

Multicast communication can be done to N destinations by using N independently constructed unicast routes carrying N duplicate streams. It can also be accomplished by broadcasting to every node in the network, although this technique would not make sense in a connection-oriented environment. Both these approaches represent the extreme in routing inefficiency. In the N unicast route case, the N duplicate streams on a link is clearly N-1 streams too many, since the data packets may be duplicated at the point the streams fork. So it almost goes without saying that an integrated tree is the proper way to multicast data. The principle behind this statement is of course the *sharing* of resources, and a higher level of sharing generally means a more efficient use of resources.

How should a tree be constructed to connect the source with all destinations in a graph? In the absence of any constraints on our decisions, a breadth first search would do as well as a depth first search as would a random walk. But of course we need to minimize resources, to stay within a delay constraint, and to be timely. The network parameters associated with the first two requirements can be represented by weights for the directed edges in the graph: a cost function **C** to represent the resources to be minimized, and a delay function **D** to represent queuing delays, transmission delays, and propagation delays. Our problem now is to find a tree of minimum cost, but subject to constraints on the delay function.

4.3 The Constrained Steiner Tree

The problem of finding a minimum cost tree (known as the Steiner Tree problem) is known in graph theory to be NP-complete and thus intractable. Kompella [Komp93] has defined the problem of finding a minimum cost tree under the end-to-end delay constraints the Constrained Steiner Tree (CST) problem, and he has proven it to be NP-complete as well. More specifically, the CST problem is defined as follows:

> **A Constrained Steiner Tree is a tree covering a given subset S of nodes in a graph, such that the sum of costs on the edges in the tree is a minimum, while the delay on the path from source to each destination is bounded above by some given delay tolerance** Δ.

The only practical solutions to NP-complete problems are ones that use heuristics, and so we will search for one or more heuristics that will help us construct a CST. The details of the heuristics can be found in Section 4.6. We will first examine the cost function and the delay function.

4.4 Cost Function

Given our goals in Section 3.1, the cost function needs to reflect both the availability of resources and the number of hops. For the first, the relationship is inverse—the lower the cost, the more resources are available. The following cost function **C** is a function of the Residual Bandwidth β, the Residual Buffer Space

$$\boldsymbol{C}(\beta, \delta, \psi) = \delta + \frac{K_1}{\beta} + K_2 \cdot e^{\frac{K_3}{\psi}} \qquad \text{(EQ 1)}$$

ψ, and the estimated delay bound δ.

We can first consider each component separately. When choosing between two links with equal residual bandwidth and equal residual buffer space, the link with a lower delay bound δ will have a lower cost. There are two reasons why we may want to include delay in the cost: although we consider delay to be mainly a constraint, low delay routes are often desired by clients; secondly, a higher delay bound is generally an indication of higher load on a link. Of course, the relative magnitudes of the propagation, transmission, and queuing delays may give one factor significantly less weight than the other.

Among links with identical delays and residual buffer space, a link with a higher residual bandwidth β will have a lower cost. This tends to encourage parity within the network with respect to residual bandwidth—links with excess bandwidth are favored and thus are more heavily used. After the links eventually attain a more uniform level of residual bandwidths, network load is evenly distributed among them. Another feature of the residual bandwidth expression is that as a link approaches saturation, it becomes "exponentially" more costly, and it quickly becomes undesirable. The residual buffer space ψ expression has an effect on cost similar to that of the residual bandwidth component.

The residual bandwidth component's exponential form compared with the delay component's linear form reflects an emphasis on direct measurements of resource exhaustion as the primary basis for cost. The scaling factor K_1 allows us to modulate this relationship even further, although it is still unclear exactly how bandwidth units should be compared with delay units. The residual buffer space component's unusual form is an attempt to minimize this parameter's relative effect on the cost until available buffer space is critically low.

4.5 Delay Function

In designing the delay function, we need to find a middle ground between overestimating and underestimating the delay bound on links. If we underestimate the delays, we will start constructing routes that are destined to fail the end-to-end delay bound test. By missing the opportunity to reroute over paths that have an appropriately lower delay, we reduce the number of successful establishments. On the other hand, if we overestimate the delays, we run the danger of artificially restricting our choice of routes. Eventually, paths that would have otherwise succeeded are summarily rejected by the routing algorithm in favor of a more conservative path (in terms of delay). Since the network is bounded by delay (i.e. there are a finite number of paths that meet the delay constraint), routes will be restricted to a smaller set of paths. This in turn speeds up the resource consumption on the links of those paths, leading to earlier saturation of resources.

The ideal strategy would be to predict the exact delay bounds that would be calculated for the channel at each node, and this could be done by performing the actual delay bound calculation as explained in Section 2.2. This calculation essentially tries to fit the candidate channel's packet service time into a slot where it will not affect the schedulability of previously established channels. We decided to estimate the outcome of this calculation using the most recently calculated delay value on this link. This scheme, while simple, has two major pitfalls: channels may be torn down, in which case a lower delay slot may open up; secondly, if the previous channel had a different maximum packet size, the difference in delay bounds could be quite significant.

4.6 Heuristics

In this section, we present the heuristics on which we will base our algorithms. Design parameters may be identified, but the actual choices will be left for the algorithms section.

4.6.1 Independent Paths

The Independent Paths heuristic attempts to approximate a Steiner Tree by merging independently calculated minimum cost unicast routes between the source and each destination. This is a slight step up from the naive unicast approach to multicasting described in Section 4.2 in that packets will not be duplicated over sub-paths that (accidentally) share common links. The details of the merge step vary depending on whether or not the minimum cost route is constrained (see Section 4.6.5), so they will be discussed with the algorithms.

4.6.2 Minimum Incremental Cost

The Minimum Incremental Cost heuristic takes advantage of the fact that the cost of a shared path is charged exactly once, regardless of the number of destinations sharing that path. In other words, once a link has been chosen to be part of the routing tree, its cost should be considered zero for the purposes of subsequent minimum cost calculations *for this channel.* That is, the cost of a path to any node in the existing subtree can be considered the cost of that path all the way to the source. So now to minimize the cost impact of an unconnected destination to the tree, the routing algorithm should find the minimum incremental cost path to the tree. This is done by choosing the lowest cost path from among the paths between the destination and every node in the existing tree.

The Minimum Incremental Cost heuristic is not concerned with how the next destination to be added is chosen, nor with how the existing tree came to be in the first place. The first point is addressed in the next section, and the latter is addressed in Section 4.6.4.

4.6.3 Arbitrary vs. Adaptive Ordering

The Minimum Incremental Cost heuristic ensures that, no matter what destination is chosen to be added to the existing tree next, the cost impact on the tree by that destination will be minimized. However, it is possible for the order in which destinations are added to have a negative impact on the low cost potential of the tree.

In Figure 1(a), destinations ***D1***, ***D2***, and ***D3*** are to be connected to the tree (shaded nodes and links). If they

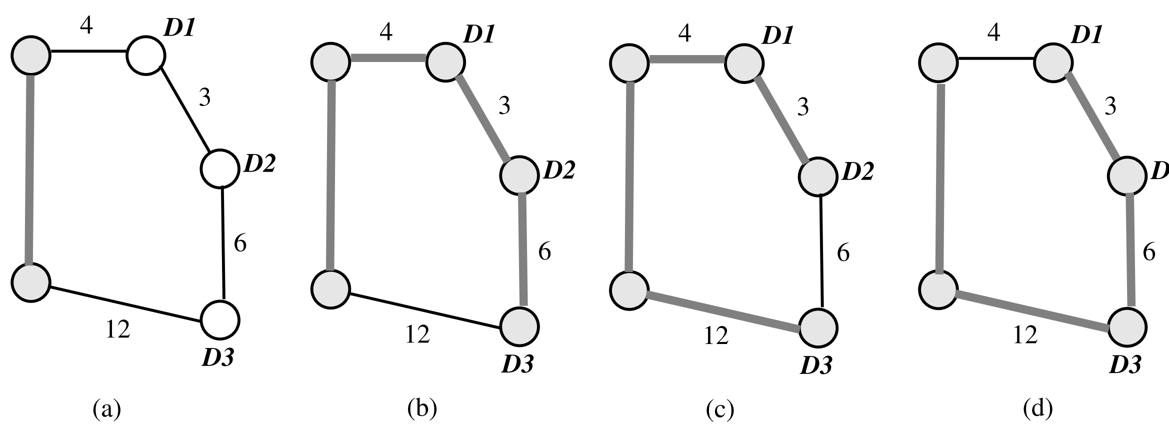


Figure 1.

are connected in that order using the Minimum Incremental Cost heuristic, we get the tree in Figure 1(b) with a low total cost of 13. This is also the case for the orders (***D1***, ***D3***, ***D2***), (***D2***, ***D1***, ***D3***), and (***D2***, ***D3***, ***D1***). But when ***D3*** is the first node to be routed, it turns out that the direct link of cost 12 is the minimum incremental cost, but that decision results in higher cost trees: 19 for the order (***D3***, ***D1***, ***D2***) in Figure 1(c), and 21 for the order (***D3***, ***D2***, ***D1***) in Figure 1(d).

To remedy these anomalous situations, we can try a greedy approach to ordering. With the Adaptive Ordering heuristic [Moran93], we select the next destination based on the absolute minimum incremental cost we can achieve. That is, we calculate the minimum incremental cost of all remaining destinations, and then select the destination with the lowest such cost. This is very similar to the greedy strategy of Prim's Minimum Spanning Tree algorithm [CoLeRi90], except that intermediate costs to non-destination nodes are not considered.

4.6.4 Hierarchical Ordering—Clustering

The most straightforward algorithm based on the previous heuristics would be to start with a tree containing only the source, and then to let the tree grow by sequentially applying the heuristic(s) to the list of destinations. Such algorithms produce "natural" trees, even if they are single branches snaking through the network. An alternate approach would be to encourage the growth of the tree in areas deemed important first.

As stated earlier, our primary motivation for building multicast trees is to encourage sharing of links among the destinations. Destinations that are close to each other should try to feed from a common backbone that carries a single stream as close to them as possible. With this in mind, our next heuristic features the clustering of destinations followed by hierarchical growth of the tree—a backbone is created by routing to a node in or near each cluster first, and then the cluster members are attached to the tree [Gupta93]. It should be apparent that there are many design parameters here, from the clustering strategy, to the order in which clusters are added, to the strictness of the hierarchy (e.g., should cluster members strictly attach to a central node, or should they attach to whatever node some other heuristic chooses for them?)

4.6.5 Delay Constraints

Up till this point, the heuristics have been concerning themselves strictly with minimizing cost and have ignored the delay along the paths. It is then likely that an algorithm using just those heuristics will construct a routing tree that violates the requirements of a CST. To avoid this problem, any "Shortest Path" algorithm employed should at least enforce a delay constraint. It should also be noted that the constraint is truly end-to-end—we cannot temporarily zero delay as we did with cost, nor can we blindly attach a branch to a tree without checking the remaining delay back to the source. Incorporating delay constraints into the heuristics only ensures that the resulting trees are constrained trees; it neither improves nor compromises the ability to approximate a Steiner Tree.

5.0 Algorithms

In this section, we present eight algorithms incorporating various combinations of the heuristics described in Section 4.6. They are further subdivided into four unconstrained algorithms and four constrained counterparts to the first four.

5.1 Unconstrained Algorithms

The following four algorithms do not consider the delay in the paths they construct, and as such cannot be seriously considered as solutions to the Constrained Steiner Tree problem. The Floyd-Warshall All-Pairs Shortest Path algorithm [CoLeRi90] is used to compute the minimum cost paths.

5.1.1 Unconstrained Independent Path

The Unconstrained Independent Paths algorithm (UIP) uses the Independent Paths heuristic exclusively. For this algorithm, a Single Source Shortest Paths algorithm such as Dijkstra [CoLeRi90] would actually have sufficed and would have less computational complexity than that of Floyd-Warshall.

Merging the paths into a proper tree requires nothing more than arbitration over equivalent cost paths. Figure 2 shows two minimum cost paths between source ***S*** and destinations ***D1*** and ***D2.*** Their paths may

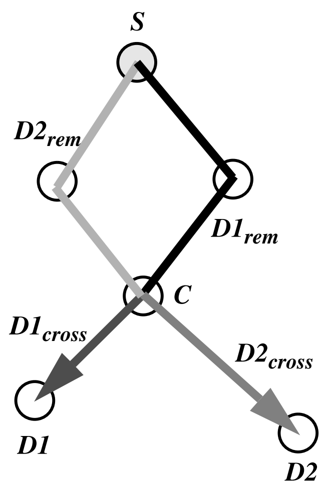


Figure 2.

cross at zero or more points, and may even share common links along the route. Consider the first common node ***C*** when tracing both reverse paths from the destinations to the source (this common node may be one of the destinations itself, or it could be the source). The cost of the path between ***D1*** and ***C*** is $\boldsymbol{D1}_{\text{cross}}$, and the remainder of that path to the source has cost $\boldsymbol{D1}_{\text{rem}}$. Similarly, the path between ***D2*** and ***S*** has segments of cost $\boldsymbol{D2}_{\text{cross}}$ and $\boldsymbol{D2}_{\text{rem}}$. Because these are minimum cost paths, it must be the case that

$$\boldsymbol{D1}_{rem} = \boldsymbol{D2}_{rem}$$

If this were not the case—say if $\boldsymbol{D1}_{\text{rem}} < \boldsymbol{D2}_{\text{rem}}$—then we could construct a new route for ***D2***, using the remainder of ***D1***'s path from ***C,*** with cost $\boldsymbol{D2}_{\text{cross}} + \boldsymbol{D1}_{\text{rem}}$. The cost would be less than ***D2***'s original cost $\boldsymbol{D2}_{\text{cross}} + \boldsymbol{D2}_{\text{rem}}$, contradicting the minimum cost assertion.[6] Since the remaining costs are equal, we can arbitrarily pick one of the two remaining paths as the common path for both destinations, resulting in a tree that preserves the minimum cost properties for both destinations. This argument can apply recursively to a merge between a merged tree and a minimum cost path.

The algorithm itself grows a tree by successively merging paths to the existing tree in an arbitrary order:

```
Unconstrained_Independent_Paths(source, destinations[])
begin
     Floyd-Warshall()
     Add source to the Set TreeSet
```

6. The reverse case uses a similar argument.

```
        for each destination do
            Retrace shortest path route from destination to source until
                a node in Set TreeSet is visited.
            Make retraced path part of the tree; add all nodes in path to
                TreeSet
        end
    end
```

The algorithm is dominated by the Floyd-Warshall computation, which takes time $O(V^3)$, where V is the number of nodes in the network. If we had used Dijkstra, the running time would be $O(V^2)$.

5.1.2 Unconstrained Minimum Incremental Cost

The Unconstrained Minimum Incremental Cost (UMIC) algorithm employs the Minimum Incremental Cost heuristic in a non-hierarchical and arbitrary order. With the Floyd-Warshall algorithm, which is run once per channel establishment request, we obtain shortest paths from every node to every other node. As each destination is considered, we scan the entire existing tree to find the smallest minimum cost path from the tree to the destination. That path then becomes a branch in the tree.

How can we be sure that the chosen minimum cost path P does not cross the tree again to create a cycle? Well, if it did, and if C is the first common node with the tree when tracing back from the destination to the source, then the path segment between C and the destination would have a cost less than that of P (a link cannot have a zero or negative cost). This contradicts the assertion that no path between the tree and the destination with a cost smaller than P can exist.

The running time of this algorithm is once again dominated by Floyd-Warshall's $O(V^3)$.

5.1.3 Unconstrained Adaptive Ordering

The Unconstrained Adaptive Ordering algorithm (UAO) applies the Adaptive Ordering heuristic to the UMIC algorithm. On each step, the minimum incremental cost path to each remaining destination is computed, and the destination with the smallest of those costs is attached to the tree next. The running time continues to be that of the Floyd-Warshall algorithm, $O(V^3)$.

5.1.4 Unconstrained Hierarchical Adaptive Ordering

The Unconstrained Hierarchical Adaptive Ordering algorithm (UHAO) uses clustering to create a hierarchy in the ordering of the UAO algorithm. There are many clustering algorithms in the literature, such as the Minimum Spanning Tree method [Jain91], although we designed our own with similar characteristics [Gupta93]:

1. Run Floyd-Warshall to obtain all-pairs shortest path costs.
2. For each destination, add it to a cluster if it is within distance *internode_distance* of another cluster member and if it doesn't make the cluster width larger than *cluster_width*. Create a new cluster for the destination otherwise.

3. Locate the center node in each cluster by calculating, for each cluster member, the maximum distance between that member and any other member, and then choosing the member with the smallest such value. Calculate the center node among all cluster centers using the same algorithm.

Next we run the UAO algorithm (excluding the Floyd-Warshall run) in three phases with the following sets of destinations: the center of cluster centers, the cluster centers, and finally the remaining cluster members (without cluster boundaries). At all times, the destinations are allowed to attach to whatever part of the existing tree the Adaptive Ordering Incremental Cost heuristic chooses.[7] The Floyd-Warshall algorithm takes $O(V^3)$ as usual, while the remainder of the clustering algorithm has running time $O(V^2)$. The UAO runs take $O(V^2)$, so the total time is $O(V^3)$.

5.2 Constrained Algorithms

The four algorithms presented in this section are constrained counterparts to the first four. Because of differences in their merge philosophies—to be discussed in Section 5.2.2—we cannot call them equivalents. In addition, the unconstrained algorithms rely on the well known Floyd-Warshall "Shortest Paths" algorithm. We knew of no constrained single source shortest-path algorithms, so we developed one based on the well known Bellman-Ford algorithm.[CoLeRi90]

5.2.1 Constrained Bellman-Ford

The Constrained Bellman-Ford (CBF) [Moran93] algorithm finds independent minimum cost paths between a source and a set of destination nodes subject to each destination's delay constraint. That is, the path will have the lowest cost possible without violating the delay constraint. The algorithm performs a breadth-first search, discovering paths of monotonically increasing delay while recording and updating lowest cost paths to each node it visits. The algorithm stops when the longest constraint is exceeded. CBF's running time has not been analyzed conclusively—we have constructed scenarios in which CBF's growth is exponential, although we have found its performance to be reasonable in practice (see Section 6.4.3 on page 37).

```
CBF(source, destinations[], constraints[])
begin
// minCostPath[] contains a list of paths for each node, sorted in order
// of increasing cost (and decreasing delay); to find a constrained path,
// take the first path in the list that meets the constraint
     max_delay = MAX(constraints[])     // Find maximum constraint
     cumm_delay = 0
     for each edge from source do       // Priority Queue key is delay
          edge.setcost(Cost(edge))      // Cost function
          edge.setkey(Delay(edge))      // Delay function
          priority_queue.insert(edge)
     end
     while (cumm_delay <= max_delay) and (not priority_queue.empty()) do
```

7. We could also attach cluster members directly to the cluster center. However, our chosen approach allows for a more uniform comparison with the constrained algorithms, with which delay constrains prevent them from arbitrarily attaching cluster members to cluster centers (see Section 4.6.5 on page 14).

```
            edge = priority_queue.dequeue()
            if relaxed(minCostPath, edge.sink(), edge.source(),
                         edge.cost(), edge.key()) then
                 for each newedge from edge.sink() do
                      newedge.setcost(edge.cost() + Cost(newedge))
                      newedge.setkey(edge.key() + Delay(newedge))
                      priority_queue.insert(newedge)
                 end
            end
            cumm_delay = edge.key()
      end
end

boolean relaxed(minCostPath, currnode, predecessor, newCost, newKey)
begin
// Add a new path to the list of paths for currnode if it has a lower cost
// (and a higher delay) than that of the path at the head of the list
      currPath = minCostPath->getList(currnode)->head()
      if (currPath != NIL) then
            if (currPath->cost() <= newCost) then
                 return false        // cost is greater, no need to relax
            else
                 if (currPath->key() == newKey) then
                 // Remove the path at head of list because it can be
                 // replaced by a path with identical delay but lower cost
                      minCostPath->getList(currnode)->removeHead()
                 end
            end
      end
      newPath->setpred(predecessor) // new way back
      newPath->setcost(newCost)
      newPath->setkey(newKey)
      minCostPath->getList(currnode)->prepend(newPath)
      return true
end
```

5.2.2 Merge Algorithm

Merging constrained paths is not as simple as it is with unconstrained paths. The problem is illustrated in Figure 3, where the edges are labeled with a cost value followed by a delay value. In this example, two destinations ***D1*** and ***D2*** have identical end-to-end delay constraints Δ of 3. ***D1*** has two alternatives for routes meeting its constraint: ***S–> N2–> N1–>D1*** with a total cost of 3; and ***S–>N1–>D1*** with a total cost of 4. Minimum cost routing obligates us to choose the first route. ***D2*** on the other hand has no choice but to use the route ***S–>N1–>D2*** with a total cost of 4, for the other alternative ***S–>N2–>N1–>D2*** has a total delay of 4 and would violate the destination's constraint. There is a similar scenario in which two destinations have identical choices among routes as far as cost and total delay are concerned, but their end-to-end constraints differ, once again forcing one destination to use a lower delay but higher cost route.

The example in Figure 3 shows how, even in the simplest of networks, the union of two constrained paths

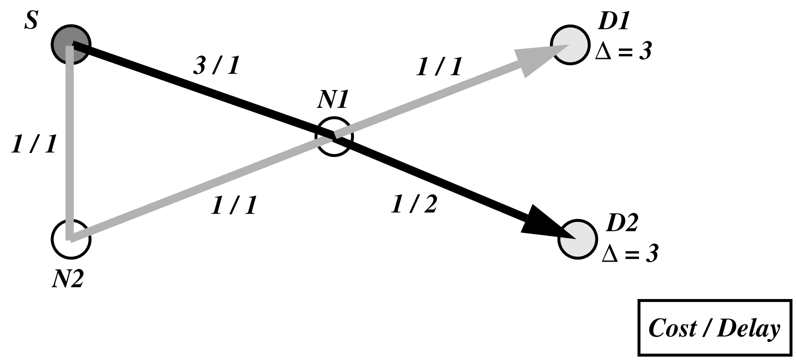


Figure 3.

can result in an improper tree. One solution would be to run a minimum delay spanning tree algorithm on the union of the complete set of paths [Komp93], which would have running times of around O(E lg E) or O(E lg V) [CoLeRi90]. Our approach is to make locally optimal decisions that keep costs down, but with the added expense of detecting and breaking deadlocks.

To resolve a conflict at the node, the minimum remaining delay path (the delay from that node back to the source) of all the conflicting paths is chosen as the one and only path back for all the paths. In the previous example, the segment ***S*–>*N1*** would be used for both destinations. This is in fact the only choice available.

Lemma

Let ω and δ be the cost and delay of the minimum remaining delay path ρ (between the source and the conflicting node ***N***), let ***D*** be the destination associated with that path, and let Δ be the maximum delay that ***D*** can tolerate along ρ (see Figure 4). Also, let ω', δ', Δ', and ρ' be the cost, delay, maximum delay, and alternate path from node ***N*** of a conflicting destination ***D'***. A path is an *acceptable alternative* for a destination if it does not cause that destination's end-to-end delay constraint to be violated. We assert that no path other than ρ is an acceptable alternative for every other destination.

Proof

Since ρ has minimum delay, we have $\delta \le \delta'$. By definition, $\delta' \le \Delta'$, and thus $\delta \le \Delta'$. This says that path ρ is an acceptable alternative from node ***N*** for every other destination.

Suppose ρ' is an acceptable alternative for ***D***, then we have $\delta \le \delta' \le \Delta$. But we could not have $\omega' < \omega$, for CBF would have chosen ρ' over ρ for destination ***D*** because of its lower cost. We could also not have $\omega' = \omega$, for CBF would have chosen ρ over ρ' for destination ***D'*** because of its lower delay (we do not have to consider the case where $\delta = \delta'$, because CBF does not record more than one path with a particular cost and delay). Finally, we could not have $\omega' > \omega$, for CBF would have chosen ρ over ρ' for destination ***D'*** because of its lower cost. Thus by contradiction, we can only have $\delta' > \Delta$, which says that ρ' is not an acceptable alternative for ***D***.

We can then say that the merge algorithm cannot affect the efficiency of the resulting tree.

The merge algorithm is based on applying the minimum remaining delay path decision at all conflicting nodes, although some nodes may not be able to make a decision until a conflict further away from the source is resolved. In Figure 5, the paths for destinations ***D1, D2,*** and ***D3*** intersect at nodes ***N1*** and ***N2***. There can be no resolution at ***N*2** until ***N1*** makes its decision, because only one of the two paths returning from ***N1*** will be an actual tree branch. In this example, ***N1*** chooses ***D1***'s path over ***D2***'s first; only then can

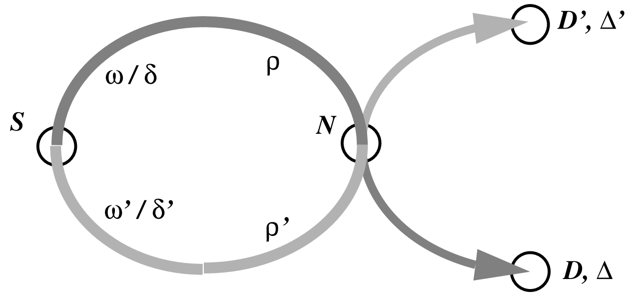


Figure 4.

N2 know that it should only choose between ***D1***'s path and ***D3***'s path, and it chooses ***D1***'s.

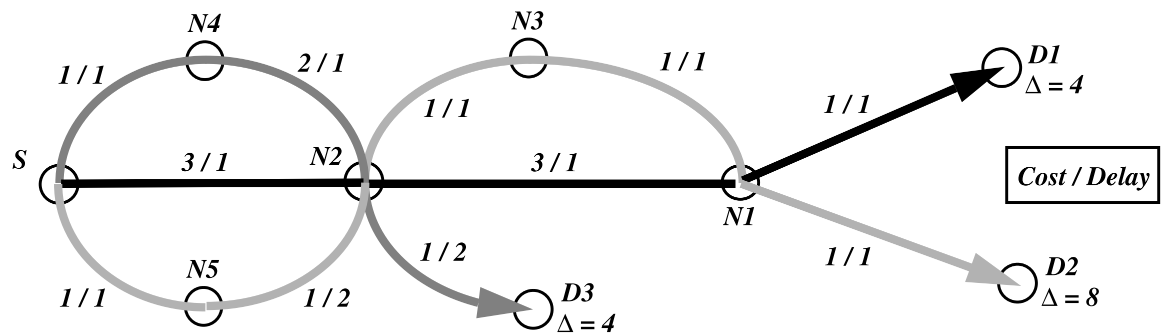


Figure 5.

The following is an outline of the algorithm:

1. By walking each path calculated by CBF for the destinations from end to end, we count, at each node, the number of paths traversing that node.
2. Starting with a forest of single node trees (the destinations), we grow the tree toward the source. Growth can proceed unimpeded and in parallel until a node with a count greater than 1 is reached.
3. At such nodes, until all potential paths are accounted for, no final decision can be made. Accounting for a path occurs when either the branch leading to this node actually becomes part of the tree, or, due to a decision further down, it is determined that this path will not be used. The count is decremented for each path accounted for.
4. When the last path to be accounted for arrives at that node, the minimum delay path is chosen as the continuation of this tree. All other paths back to the source are declared to be unused, and nodes along those paths are notified.
5. A condition that leads to a deadlock is when two paths traverse a link in opposite directions. We chose deadlock detection over deadlock avoidance because we have found deadlocks to be an infrequent occurrence.

   In Figure 6(a), each node waits for the other to resolve its conflict (the values on the tails represent the remaining cost/delay to the source).To break the deadlock, we make note of each head unresolved link (a head unresolved link is the first unresolved link encountered by tracing from a destination to the source), and then traverse each unresolved path from the destination towards the source until a "reverse match" is found. In the example, we note that links ***N1–>N2*** and ***N2–***

>***N1*** are links at the head of unresolved paths; then, if we start traversing at ***D1***, we will match regular unresolved link ***N1–>N2*** with head unresolved link ***N2–>N1***.[8] If the head link's remaining delay (10 + 50 = 60) is shorter than the tail of the other path (70), then the head link has the right-of-way (since it represents a shorter path from the source to ***N2***), and we can forcibly remove this link from the deadlock.

This test can still result in stalemate, as can be seen in Figure 6(b). Here, the head link's remaining delay of 60 is greater than the tail link (55). However, we cannot just choose the tail link at ***N2*** as the resolved tree branch, because we have no idea whether that path will actually become part of the tree. This example is even worse, because we run into the same situation from ***N1*** as well. Our algorithm tries to solve this by looking ahead a few links, but a more sophisticated algorithm would be needed to handle all deadlocks.

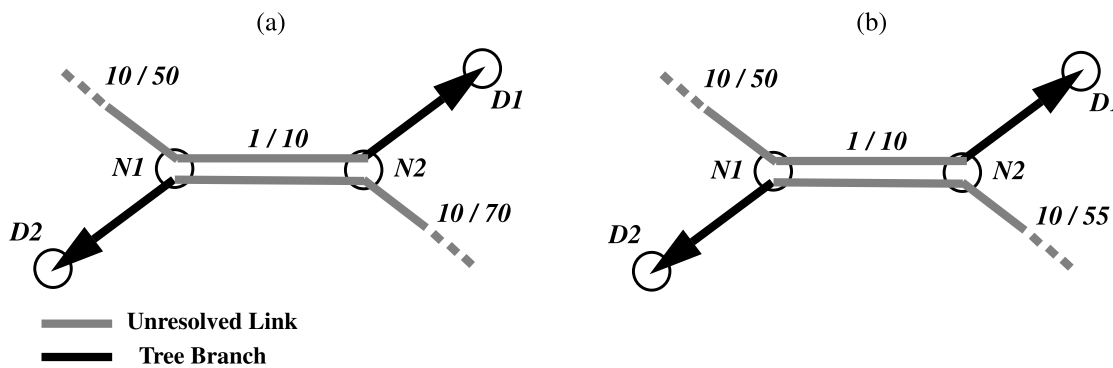


Figure 6.

The merge algorithm runs in time $O(V_d{*}V)$, where V_d is the number of destination nodes (bounded by V), since it traverses each path twice per destination, and any path has at most V edges.

5.2.3 Constrained Independent Paths

The Constrained Independent Paths algorithm (CIP) uses CBF once to obtain independent constrained minimum cost paths to every destination, and then runs the merge algorithm over them to get a tree. This algorithm runs in time O('CBF') due to the CBF run, which dominates.

5.2.4 Constrained Minimum Incremental Cost

The Constrained Minimum Incremental Cost algorithm (CMIC) uses the zero cost strategy of the Minimum Incremental Cost heuristic. CBF is run for each destination in arbitrary order. After each run, the cost of the resulting path is set to zero for the remainder of the destinations. Because of delay constraints, destinations that attach to an existing zero cost paths may not necessarily stick with that path to the end, or they might stay away from the existing tree altogether. The merge algorithm is run on the collection of paths as a final step. This algorithm runs in time $O(V_d{*}O(\text{'CBF'}))$, as the CBF algorithm is run for every destination.

8. This example just happens to have two head links conflicting. A head link may conflict with an intermediate link of another path, if that path's progress is deadlocked elsewhere.

5.2.5 Constrained Adaptive Ordering

The Constrained Adaptive Ordering algorithm (CAO) applies the Adaptive Ordering heuristic to the CMIC algorithm. CBF is run at each step, after which we find the shortest of the paths between the source and the remaining destinations. That path and destination are added to the collection of zero cost paths (they don't necessarily form a tree yet). Again, the final step is to run the merge algorithm. This algorithm runs in time $O(V_d*[O(\text{'CBF'})+V_d]) = O(V_d*O(\text{'CBF'}))$.

5.2.6 Constrained Hierarchical Adaptive Ordering

The Constrained Hierarchical Adaptive Ordering algorithm (CHAO) uses the clustering algorithm in Section 5.1.4 on page 17 to impose a hierarchy on the ordering of CAO. Analogous to the strategy employed by UHAO, CAO is executed in phases: on the center of cluster centers, on the cluster centers, and finally on the remaining cluster members. The merge algorithm is the final step. This algorithm runs in time $O(V^3)$, the running time of the clustering algorithm.

6.0 Evaluation

In this section, we evaluate our design decisions through simulation.

6.1 Evaluation Metrics

A common metric for evaluating heuristic algorithms is a comparison of the results with the theoretical solution. In our case, we could compare the cost of the trees our algorithms produce with the cost of a constrained minimum cost Steiner Tree for the same problem.[9] However, the CST is only a means to an end, the end being the goals in Section 3.1. Even a perfect CST has diminished usefulness if the cost or delay function is inaccurate, and with it we really can't measure the routing algorithm's success with respect to channel establishment. In addition, the timeliness of the algorithms cannot be measured using such an analytical approach, nor can we use it to investigate issues such as how network knowledge staleness affects our routing effectiveness.

A more useful metric is some measure of the number of successful channel establishments achieved. Such a metric must of course be qualified to provide a context, and it is important that the context allow for "interesting" results—a network with exactly one route between any two nodes wouldn't be useful, and the same goes for clients requiring no performance guarantees or clients that underwhelm the network with slow traffic (i.e., long packet interarrival times). If we are measuring the probability of successful current establishment, we might count how many random establishments are successful if each one is initiated in a network of identical load. Similarly, for the probability of future establishments, we could repeatedly interrupt a set sequence of establishments with a single random establishment, and then measure how long it takes to reach saturation [Moran93].

One semantic clarification we must make is with the meaning of "successful establishment." The Tenet Scheme allows for partial success in channel establishment, meaning that not necessarily all destinations

9. The CST problem is intractable but not impossible. The solution requires a systematic enumeration of spanning trees [Komp93], giving it an exponential running time; nonetheless, it certainly is solvable.

have to be included in the 1 by N multicast channel for the channel to be considered active. Applications may accept partial success (e.g. a video lecture to thousands of participants) or may consider it a failure of the channel (e.g. a three-way conference call). The concept of partial success raises the possibility that a client has prioritized the destinations, giving the success of some destinations more weight over the success of others. This definition refers to the actual establishment of the channel—if one or more destinations "fail" during routing and are omitted from the route, we consider this a failure of the channel, regardless of how many destinations actually do get established. This is to catch algorithms that do a bad job of estimating delays, thus rejecting destinations when they could actually have succeeded.

As an initial step and for the purposes of this paper, we used for a metric the total number of successful establishments possible given an initially unloaded network. That is, we would repeatedly establish channels in some contrived or random pattern (depending on what we want to investigate) until the network is saturated, where saturation is defined below. We will measure both the total number of successful destinations as well as the total number of completely successful channels, and, to avoid an overly complex metric, we will consider all destinations to be equally important. When the pattern is a repetition of a single channel request, saturation occurs on the first complete failure, since the state of the network does not change from that point on. With random requests, saturation is harder to detect. Consider a network that is fully loaded except for one link which can accept one more channel, and then consider how many random establishments may be needed before that last link is used successfully. For our evaluation, we will determine a threshold frequency of establishment successes, below which we will consider the network saturated.

This aggregate measurement reflects a routing algorithm's effect on both the probability of successful establishment as well as the probability of future establishment, with no clear way to isolate each contribution. Nevertheless, this is enough feedback to help us refine the majority of the parameters of the algorithm. We would eventually like to have secondary metrics that provide a sense of the effectiveness of an algorithm after only looking at one or a few routes. A well tuned cost function would serve such a purpose—if minimizing the cost really does meet our goals, then the lower the cost of the tree, the better the algorithm is for our needs. There are many other potential secondary metrics [Widyon93], although they are beyond the scope of this paper.

6.2 Simulation

Metrics can be measured through analytical modeling, through simulation, or taken directly from the system [Jain91]. The system in our case is the universe of all possible networks (topologies, node capacities, and link capacities) together with a routing algorithm, and the service is the Tenet real-time channel. The number of permutations make any evaluation technique intractable, and so we worked with a few selected representative network configurations. The inaccuracy of the analytic modeling approach favors simulation, especially when the simulation of routing and channel establishment is not expected to be time consuming (considering that timeliness is a goal). Simulation of pure computational algorithms (routing and admission control) can behave very much like the real system[10], and it will have the flexibility needed to perform the experiments we desire. An implementation of Tenet Scheme 2 does not currently exist, and, even if it did, finding many interesting real-life network topologies on which to run experiments would be impractical within the time frame of this project. For these reasons, we have chosen simulation as our evaluation method.

10. In the Tenet case, the implementation will likely adapt much of the algorithmic code from the simulations.

6.2.1 Simulator

To evaluate our routing algorithm design choices, we needed to simulate not only the routing algorithm but also the complete channel establishment protocol. We developed GalileoMC[11] [HeOgWi93], a simulation of Scheme 2 protocols and algorithms, on top of the Ptolemy object-oriented simulation tool under its Discrete Event domain [Alm92][LeeMes93].

Under Ptolemy, the applications execute as objects (*Blocks*) managed by the Ptolemy kernel, which provides scheduling and inter-object communication among its services. The various application-specific functional components are elemental Blocks called *Stars*. Stars can be organized into composite Blocks called *Galaxies*, which can themselves be organized into other Galaxies. Blocks communicate among themselves by sending *Particles* (messages) through their output ports, which have been logically connected to some other Block's input port. A closed system (one with no unconnected ports) of interconnected Stars and Galaxies makes up a complete application, called a *Universe*.

A Universe or a subset of one will execute under a *Domain*, which determines the semantics of the progression of the application. Under the Discrete Event domain, individual stars will execute for arbitrary wall clock durations, but it is artificially time-stamped messages sent between Stars that matter to the Ptolemy scheduler. The time stamps come from a virtual time scale and are intended to represent actual relative or absolute times in the system being simulated. The messages constitute events to be processed by the receiving Stars, and they are globally queued with temporal priority. The kernel will continue to schedule Stars for execution until either the queue is empty or a virtual time limit has been exceeded.

In GalileoMC, our network nodes are Galaxies comprising the various functional components that implement the channel establishment and data delivery phases of a Tenet channel. Of interest to us is the Real-Time Channel Administration Protocol (RCAP) module, which, as a distributed entity present on all nodes, accepts and executes channel establishment requests from Host (client) modules. The first step for each request is to obtain a route from the routing module. Then channel establishment packets propagate, duplicate, and coalesce from the source to the destinations and back, with admission control being performed at the intermediate RCAP modules. When a link is accepted into a channel, the virtual resources it would consume are recorded, affecting future admission control tests. In this way, the simulation very closely models what would transpire in an actual implementation.

Some simplifications have been built into the simulator:

1. We did not simulate the propagation of network state throughout the network, and, instead, the routing algorithm receives instantaneous notification of changing resource availability. When a channel segment is established on a particular node and link, that node immediately updates the cost of that link in the data structure representing the network to the routing algorithm. The routing algorithm uses the same data structure for all nodes, thus it is behaving like a route server.
2. All nodes were given identical processing times for the channel establishment, and we did not attempt to associate this virtual time with any real processing time. The timeliness of the channel establishment phase is important to us (especially its scalability), but it is beyond the scope of this study.

11. GalileoMC is an extension of Galileo, a simulation tool for Scheme 1 [KniVen93].

6.3 Experimental Methodology

We ran several experiments to verify that our metric does help differentiate between different design choices, to illustrate some alternatives we considered, and hopefully to set us on the right course towards choosing an algorithm for Scheme 2. Clearly, these are not intended to be the final experiments in this area—our goal was to provide a framework for the development of routing algorithms.

The experiments would ideally be performed on a large number of networks and with a wide variety of workloads, to ensure the robustness of the algorithms. However, the sheer number of potential factors forced us to fix quite a few of the network parameters and to acknowledge that the results can not be readily generalized. Indeed, several of our observed results are artifacts of having some parameters fixed.

6.3.1 Networks

We used an eight by eight grid for our primary test bed (Figure 7). This provided a rich set of alternative paths between any two nodes, and it would also let us dynamically generate interesting random networks. Every link was identical, with a bandwidth of 1.544 Mbps (T1) and a propagation delay of 80 µsec (the

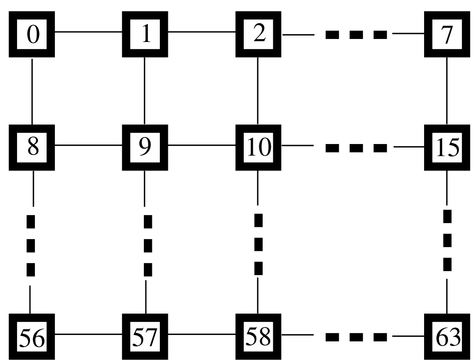


Figure 7. 8x8 Grid Network

approximate propagation through 10 miles of fiber-optic cable).[12] Every node was identical, with 64MB of buffer space (memory) each.

6.3.2 Workloads

As stated earlier, our ideal workload would include all possible patterns of real-time channel requests, with varying source and destinations, traffic, and performance requirements. Since this is not practical, a reasonable alternative is to use a trace consisting of a mix of channel establishment requests. Unfortunately, since connection-oriented multicast communication has only recently emerged, there are few such examples; those that do exist most likely are not characterized using the parameters in Section 2.1. For this study, we turned to two types of synthetic workloads: for most of the detailed comparisons, we continuously repeated a contrived channel establishment request until saturation; and for a final comprehensive comparison, we generated a random trace, varying only those parameters that were not fixed in the earlier comparisons.

All our establishment requests had a fixed traffic type, with the parameters in Table 1. The 53 byte maxi-

12. These values are arbitrary, and were chosen to allow for saturation within a reasonable amount of time.

Parameter	Value
x_{min} - Minimum Packet Interarrival Time	6.25 msec
x_{ave} - Average Packet Interarrival Time	12.50 msec
I - Averaging Interval	100 msec
S_{max} - Maximum Packet Size	53 bytes

Table 1. Traffic Parameters

mum packet size corresponds to that of an ATM cell, and such cells arriving at the maximum rate every 6.25 milliseconds result in a bandwidth requirement of

$$53\,(\text{bytes} / \text{packet}) \times 8\,(\text{bits} / \text{byte}) \times \frac{1\,(\text{sec})}{0.00625\,(\text{sec} / \text{packet})} = 67.8\,(\text{Kbps})$$

With a 1.544 Mbps bandwidth, each link will reach node saturation after approximately $1.544\,(\text{Mbps}) / 67.8\,(\text{Kbps}) \cong 22$ channels of this type have been established on this link. All requests were spaced apart in virtual time such that no channel establishment will overlap any other one, and we also did not tear down any channels for the duration of the simulation.

The channel that was repeated originates at node 3 (see Figure 7) and has 8 destinations: 23, 30, 48, 49, 57, 61, 62, and 63 In Figure 8, the source is the solid black node on the top row, and the destinations are the eight nodes highlightedwith various shades. The groups represented by the shades are the clusters that will be formed by the algorithms using Hierarchical Ordering. In order to obtain “interesting” results, the clustering algorithm’s parameters were chosen specifically so that some clusters would be formed. Finally, this channel’s performance guarantee requirement was either varied between 5 milliseconds and 1 second, or it was fixed at some value of interest, as explained in the next section.

For the random synthetic trace, we created 1000 random channel establishments. We picked the source and between 1 and 10 destinations randomly and uniformly from the 64 nodes. We also varied the maximum delay bound D within ranges that will be discussed with the experiments..

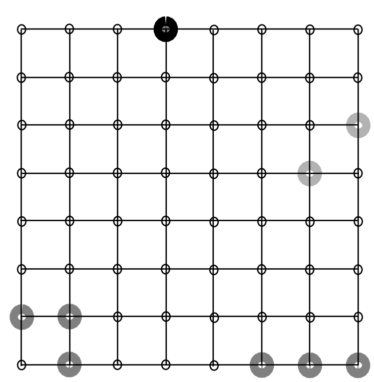


Figure 8. Source and Destinations

6.4 Experiments

In this section, we describe and present the results of experiments conducted either to assist us in choosing among design alternatives or to verify some basic assumptions or conclusions we made earlier. Care must be taken not to over-generalize the results, as they have been obtained from an environment with many fixed parameters.

6.4.1 Cost Function

The cost function in Equation 1 on page 10 has two significant components, the residual bandwidth β and the delay δ. Because buffer space is not expected to be a scarce resource in most cases, we elected to defer investigation of the residual buffer space component ψ to a future study. We wanted to determine whether each component contributes positively to the route and how much each component should contribute. To do so, we ran several repetitive trace simulations of the CAO algorithm while varying the cost function configuration. For each configuration, we made a series of runs with maximum delay bounds ranging between 20 milliseconds and 200 milliseconds, and we counted the total number of destinations successfully established.[13]

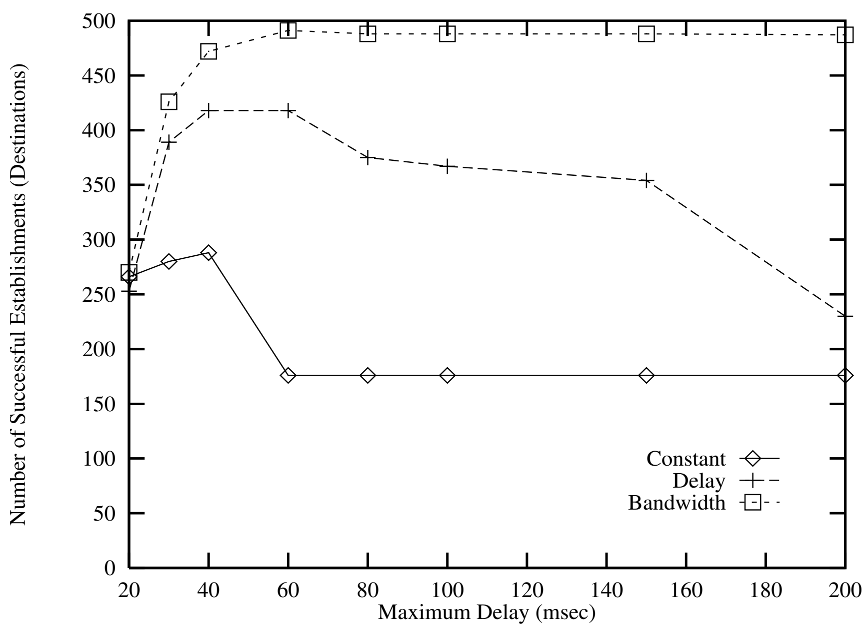


Figure 9.

In Figure 9, we compare the results of using a constant function ($C(\beta, \delta) = 1.0$), the delay component isolated, and the residual bandwidth component isolated. The constant function essentially provides a hop-count for a tree, and it clearly performs poorly compared with the "intelligent" functions. As stated earlier, delay bound only has a loose correlation with the load on a link, and the results show that it does indeed steer routes towards less heavily loaded links. But as expected, the bandwidth function outperforms the other two functions since it deals directly with the node saturation.

The bandwidth function tends towards infinity as a link approaches saturation, sending a clear signal to the routing algorithm to avoid this link at all costs. The delay function, on the other hand, will be unaware of node saturation on a link, and when other links eventually earn higher delays, the routing algorithm will continue to use saturated links. This effect is minimized when the maximum delay bound is low (between 20 msec and 60 msec) as constraint violations dominate here, protecting routes from saturated links. This

13. The choice of algorithm, delay bound range, and metric are due to feedback from experiments presented later.

protection wears away with higher bounds, and so we see a dramatic decrease in performance of the delay function. Finally, it is precisely the constraint violations that force all functions to accept fewer destinations on the low end (towards 20 msec).

We can also verify that a link reaches saturation after establishing 22 channels. With the constant function, we expect the routing algorithm to produce the same tree every time, as long as delay constraints are not violated. This means that all links will progress evenly toward node saturation until 22 channels or 176 destinations have been established. For maximum delay bounds of 60 or greater, this is exactly the case. On the other hand, with lower maximum delay bounds (below 60), the routing algorithm is eventually forced to consider alternate routes, as the delays along the original routes start violating the constraints. The use of alternate links in turn allows more destinations to be established, since more links need to reach node saturation before the entire network is saturated.

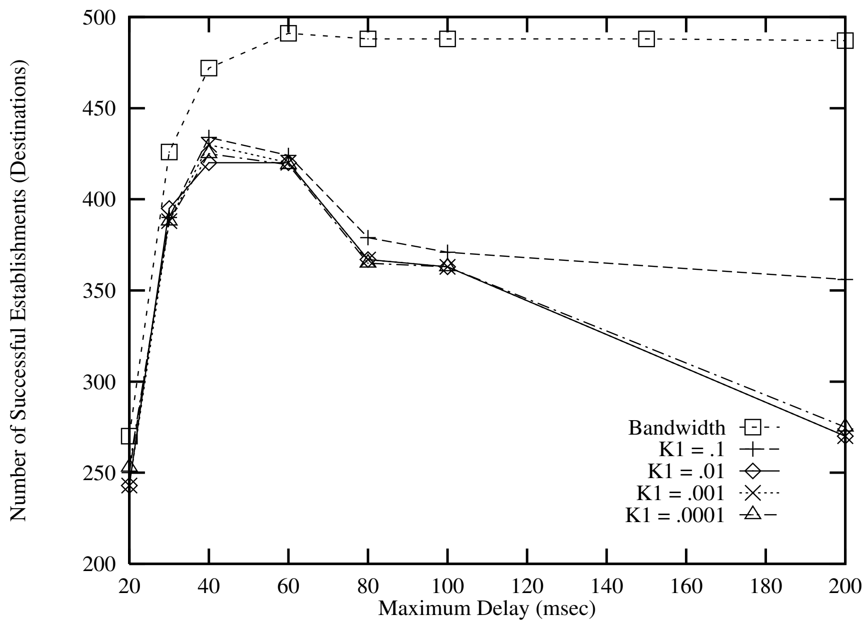


Figure 10.

The next question is whether some combination of delay and residual bandwidth might perform even better, and if so, what combination? The unmodified cost function in Equation 1 (excluding the residual buffer portion) makes such a union, with a scaling factor K_1 equal to 1.0. In Figure 10, we compare our current front-runner, the bandwidth only function, with versions of the cost function where the residual bandwidth is scaled down ($K_1 < 1.0$). It is clear we don't want the delay portion to dominate, so we next look at values of K_1 greater than or equal to 1.0.

Figure 11 suggests that we very likely cannot do better. It seems that the bandwidth function behaves as the $K_1 = \infty$ limiting case of the cost function.We thus conclude that the delay portion of the cost function does not make a significant positive contribution, and so it should be omitted.

6.4.2 Algorithms

In our next set of experiments, we compared the eight routing algorithms by running them against both the repetitive trace and the random traces. For the repetitive trace, we varied the maximum delay bound

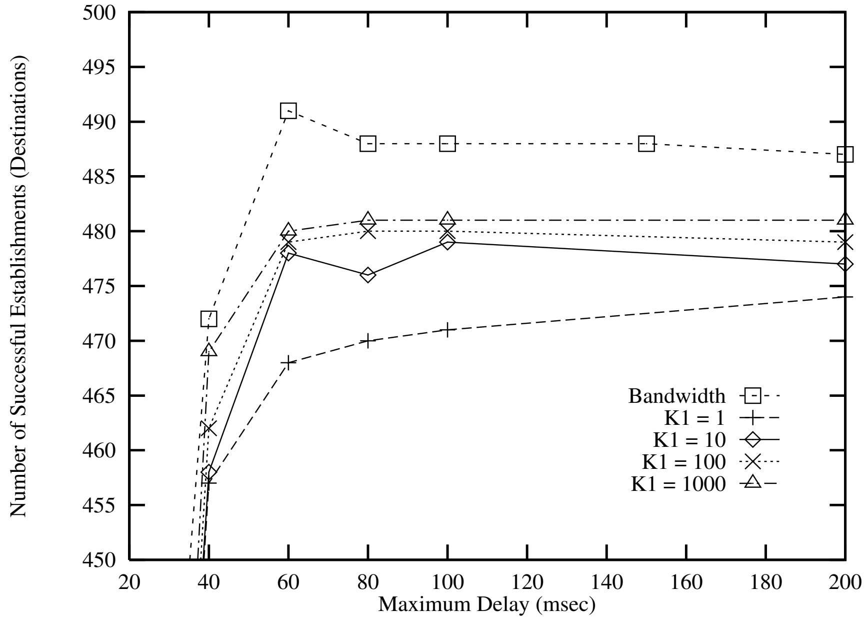


Figure 11.

between 5 and 1000 milliseconds, and we recorded both the total number of destinations established and the total number of channels fully established. The cost function was fixed with K_1=1.0, an artifact of the order in which we ran the experiments.

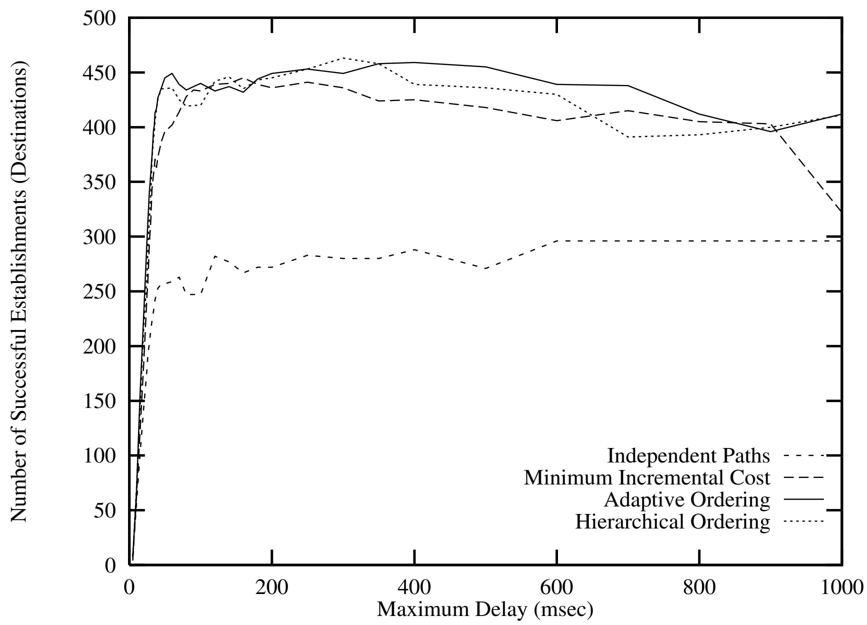


Figure 12.

Figures 12 and 13 show the total established destinations results of the unconstrained algorithms separated from the results of the constrained algorithms. The 176 destinations from a constant cost function can be considered a worst case baselevel. By using minimum cost unicast paths, we obtain a modest improvement to around the 280 mark as a reward for being somewhat efficient with resources. However, clearly the most significant incremental improvement is due to the application of Minimum Incremental Cost heuristics, which indicates the importance of bandwidth sharing.

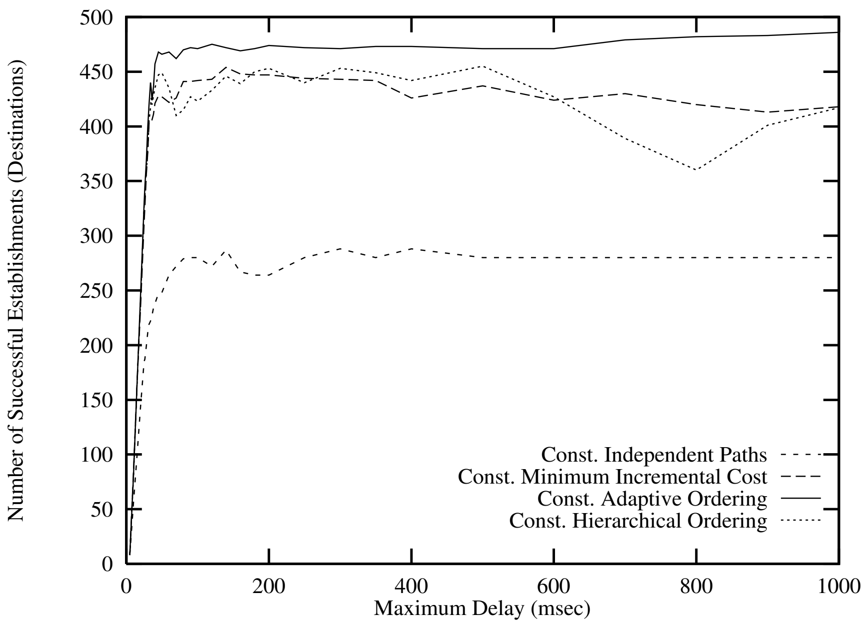


Figure 13.

As for the six variants of Minimum Incremental Cost algorithms, there is an interesting difference between the three unconstrained and the three constrained algorithms. Among the unconstrained algorithms, there never is a clear winner, as the curves intersect at numerous points. This demonstrates the significant incoherence introduced by ignoring a primary cause of failures, namely delay constraints. The variance in results due to constraint violations masks any fundamental cost advantages one algorithm may have over another. We should also remember that this is the result from a single (repeated) channel request—a different request results in a different, unpredictable pattern.

With the constrained algorithms we have removed a major source of noise, revealing more consistent patterns. We can now verify that intelligent ordering (CAO) is better than random ordering (CMIC), and that there are a significant number of anomalies that the Adaptive Ordering heuristic deals with. The erratic curve for the CHAO algorithm seems to be due to the contrived nature of the Hierarchical Ordering heuristic (clustering and forcing a balanced tree), as compared with the other "natural" heuristics. In addition, the position of the curve below CAO tells us that we did not derive any usefulness out of this heuristic, at least for this type of channel request. So overall, the Constrained Adaptive Ordering algorithm seems to do the best job.

When comparing constrained algorithms with their unconstrained counterparts, we find that the constrained ones generally do better, but not by much. Figure 14 shows the comparison for the Adaptive Ordering algorithms, where we see a clear advantage for taking constraints into account.[14] Looking across the horizontal axis, we can see some commonality in how the algorithms deal with different delay constraints. When the bounds are tight (below 35 milliseconds), constraint failures completely dominate and limit the number of establishments possible. In fact, other statistics we gathered show that not only is the number of failures by the constrained algorithm similar to that of the unconstrained algorithm, but it is also the case that all the unconstrained algorithm's establishment failures were "predicted" by the constrained algorithm, validating our delay function model (for this experiment). When bounds are not tight, node saturation becomes the

14. Comparisons for the other algorithms can be found in Appendix A.

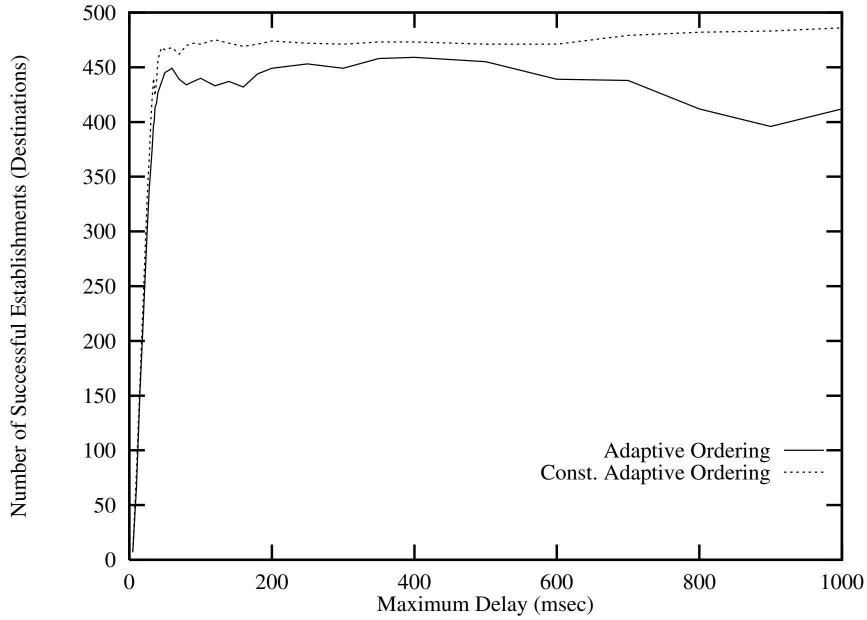


Figure 14.

primary cause of failure, but an unconstrained algorithm will continue to inject constraint failures.

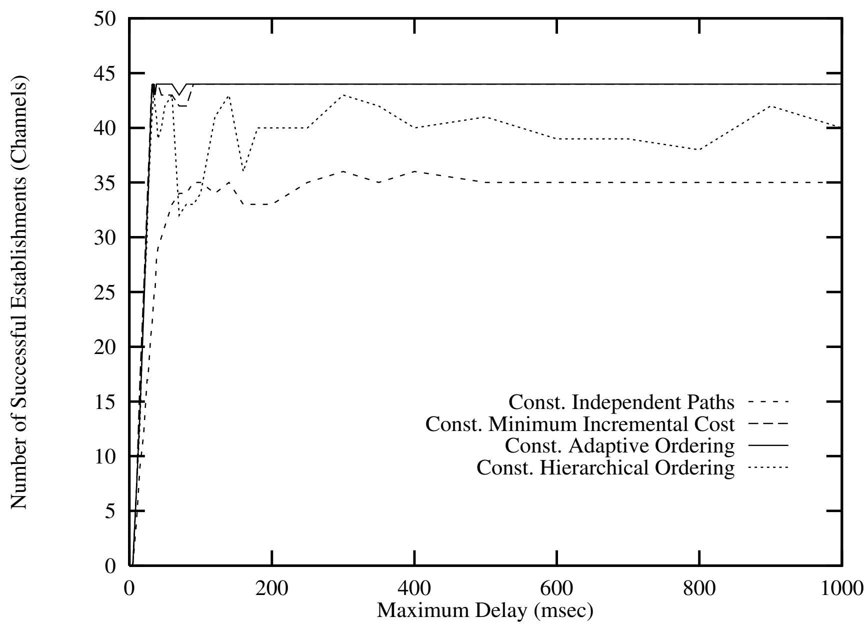


Figure 15.

We also counted the number of fully established channels, and the numbers seem to parallel the destinations numbers, but within a compressed range. This limited range also seemed to limit the usefulness of these statistics. Figure 15 shows the comparison of constrained algorithms (the remainder of the graphs are in Appendix A), where we can't really differentiate between CMIC and CAO. We must keep in mind that this data should only be interpreted as performance within a mixed environment, where partial failures are allowed. In a network where only complete channels may be established, the results may be quite different.

So far, it seems that the CAO algorithm has outperformed the others, but this is for that one channel that

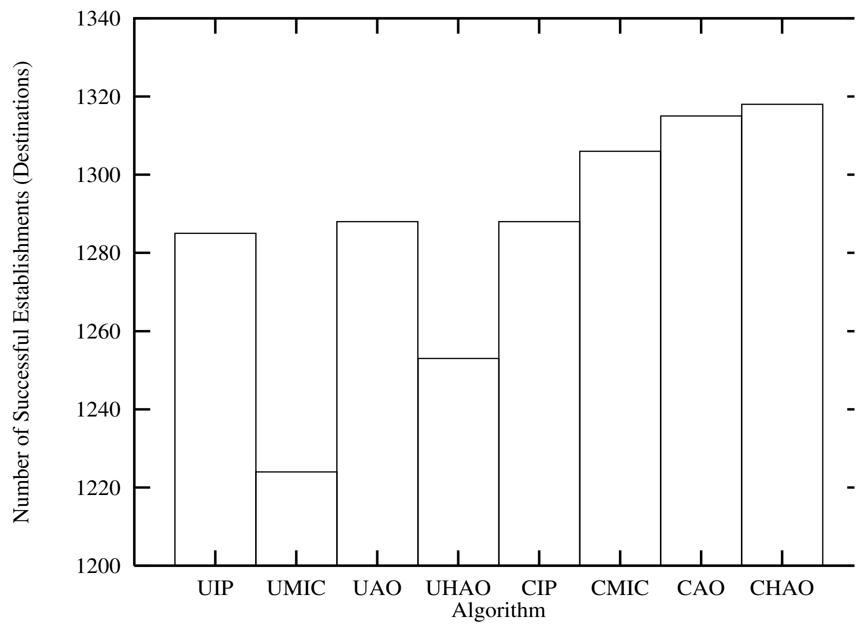


Figure 16.

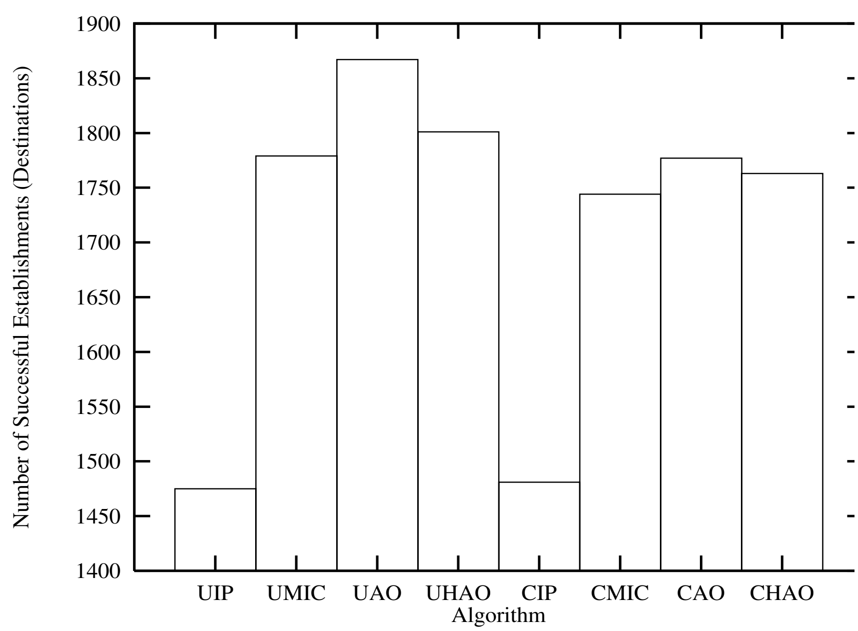


Figure 17.

was repeated. We also ran the random trace of 1000 establishments over all the algorithms, as described in Section 6.3.2. In one run, we varied the delay bound between 5 milliseconds and 20 milliseconds to ensure that constraint violations will be prevalent. In the second run, we varied between 20 milliseconds and 50 milliseconds. The results of the first run in Figure 16 were consistent with our earlier observations in that constrained is better than non-constrained under tight bounds, and that the three constrained Minimum Incremental Cost algorithms are close to each other. The second run (Figure 17) brought much more of a surprise: the unconstrained algorithms for the most part outperformed the constrained algorithms. One explanation is that our delay function model is not as good after all, and it is causing the algorithms to be overly conservative with delay bounds, i.e., routes are being rejected prematurely. A second explanation is that the constrained algorithms are not doing that good a job minimizing aggregate cost.

6.4.3 Timeliness

Timeliness is difficult to measure outside of the system itself. When measuring something as compute intensive as a routing algorithm, however, the running time of a simulation should correlate closely with that of the implementation. Our simulations ran on a DECsystem 5500 with 128MB of memory (23 SPECmark rating), and we timed the execution of all our algorithms. Figures 18 and 19 show the average execution times of the eight algorithms, after running them on networks of size 8 through 64. As expected, the unconstrained algorithms (Figure 18) have very similar running times, since they are all based on the Floyd-Warshall algorithm. The constrained algorithms (Figure 19), on the other hand, apply CBF differently. CIP executes CBF once per channel, and thus has a slow growing curve. CMIC and CAO both run CBF once per destination, and they have steeper curves. CHAO also runs CBF as often as CAO or CMIC, but it also needs Floyd-Warshall to do its clustering algorithm. The combination leads to some very big and disturbing numbers. 600 milliseconds for 64 nodes is not a problem; the question is whether the algorithm can handle networks with thousands or perhaps millions of nodes in a reasonable amount of time. We

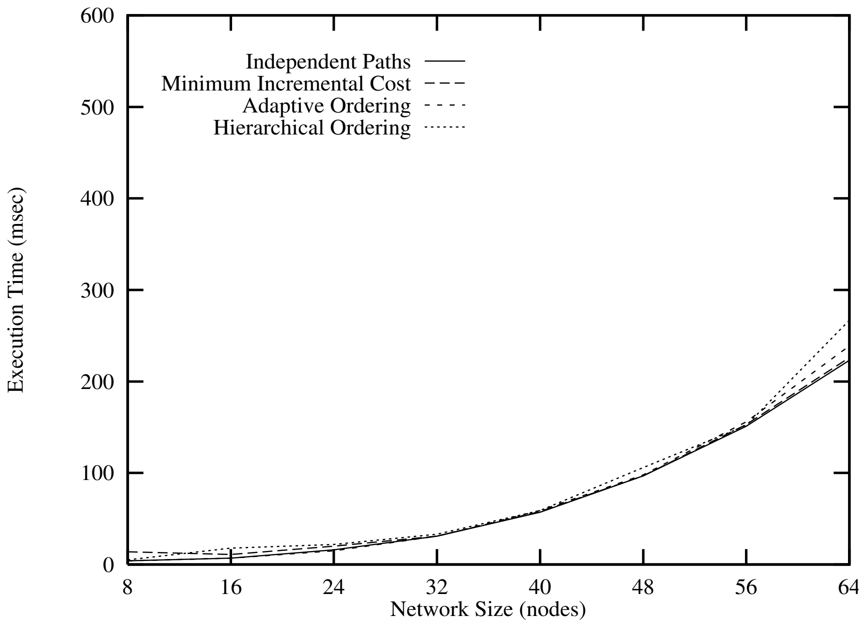


Figure 18.

expect the planned research in hierarchical routing to deal with this issue.

7.0 Related Work

Very little work has been done in the area of multicast routing for connection-oriented networks with performance constraints. The most relevant work to date is by Kompella [Komp93], whose CMCT algorithm treats delay strictly as a constraint, a position we agree with. The algorithm creates a closure graph of collapsed minimum cost constrained routes between the source and every destination, on which it constructs a minimum spanning tree (again subject to constraints). The collapsed links in the spanning tree are unraveled, and finally loops are broken by running a minimum delay spanning tree algorithm. This algorithm is based on the Kou-Markowsy-Berman (KMB) algorithm, which uses the MST over a closure graph approach to approximate a regular minimum Steiner Tree. Our primary objection with CMCT is that the closure graph prevents us from meeting certain advanced routing goals, such as maximizing link sharing

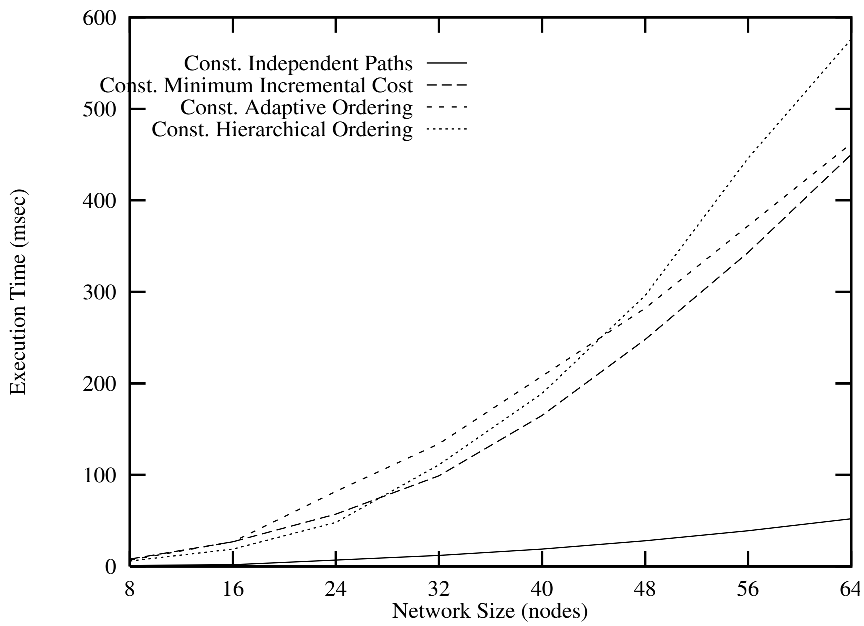


Figure 19.

among several channels.

The running time of CMCT is $O(V^3\Delta)$, where V is the number of nodes in the network, and Δ is the maximum delay bound. Because Δ must be represented with $O(\log \Delta)$ bits, the solution is exponential. Kompella's claim is that, by reducing the granularity of Δ and thus restricting the set of values it can take to a small range, the solution is practical. We have found, through simulation, that this strategy compromises the accuracy of the algorithm when the size of the network is large and there are several orders of magnitude between the smallest and the largest delay bounds. Rounding a 10 μsec propagation delay up to 1 msec is not a problem with one or two hops, but, with 50 hops, a 500 μsec actual end-to-end propagation delay is modeled as a 50 msec delay, which increases the chance for the path to be artificially rejected.

[DoaLes93] has looked into strategies for adding new members to heuristically derived (unconstrained) Steiner Trees. Using the computational complexity and the inefficiency (the ratio of the cost of the resulting spanning tree to the cost of the minimal Steiner tree) as their performance metric, they found that the naive strategy identical to our Independent Paths heuristic does not perform much worse than a complex heuristic, yet membership addition is as easy as computing another independent minimum cost path.

In [Waxman93], the Weighted Greedy Algorithm (WGA) is compared with other dynamic Steiner tree algorithms. The WGA operates in a bandwidth reservation connection-oriented environment, and it favors minimizing network resource usage over minimizing delay (not constraining delay) as a criterion for route selection. In addition, it is designed to handle destinations dynamically joining and leaving a connection. It is also "practical" in that it is an extension of point-to-point routing and can be implemented in a distributed fashion. Using the ratio of carried load to offered load as a metric, WGA was found to perform well against a Minimum Spanning Tree algorithm, a theoretically "good" algorithm for the minimum Steiner Tree problem.

The DCM Routing Algorithm [ParFer93] is a unicast algorithm for the Tenet Scheme 1. It solves the problem of finding constrained routes using a constrained version of the Bellman-Ford shortest path algorithm.

Because the algorithm is based on the Rate-Controlled Static Priority scheduling discipline [ZhaFer92], and because it only calculates minimum hop paths, it cannot be extended for our purposes.

Other protocols for network QoS guarantees either do not address routing, or they make a simple attempt at routing. The ST-II protocol builds a multicast distribution tree from unicast routing tables, while RSVP relies on the underlying network for routes [MiESZ93]. The Session Reservation Protocol (SRP) [DiGen93] makes resource reservation on the Internet, but relies on IP routing to provide its route.

8.0 Conclusions

In this project, we set out to learn how to develop a routing algorithm in a real-time multicast connection-oriented environment, specifically the Tenet Scheme 2 from the University of California at Berkeley. With few precedents to guide us, we needed to specify a methodology for the evaluation of such algorithms. The first order of business was to clearly define the goals of the routing algorithm: to maximize the probability that a channel being routed will be successfully established, to maximize the useful utilization of the network, and to be timely. From this, we were able to identify resource efficiency and resource (delay) constraints as primary components of the algorithm and of the evaluation metric.

To obtain resource efficiency, we needed to solve the Constrained Steiner Tree problem, where the cost of a tree is minimized with constraints on the maximum delay length of any source to destination path. We formulated a cost function, estimated delay, and, because the CST problem is NP-complete, developed several heuristics for the construction of the multicast tree. These heuristics were then used to develop eight algorithms with various combinations.

We defined the total number of successful establishments possible from an empty network as our evaluation metric, and built a simulator to measure that value. We ran some experiments that verified some of our assumptions, but they also revealed many new issues to be investigated in future studies. These experiments attest to the viability of our tool and methodology for the development of a truly robust algorithm.

Among the questions and issues remaining to be addressed are:

- We made many assumption and restrictions, giving our results limited validity within a true network. We need to test more types and ranges of performance bounds, test many more traffic types, test many different individual routes, and test many different sizes and configurations of networks. We should allow establishments to overlap with other establishments as well as teardowns. We need to simulate the staleness of network state knowledge.
- We can also improve our cost function and delay estimate, and we can develop even better heuristics based on what we've learned. Specifically, we can already imagine a heuristic that favors destinations with tighter delay bounds first, as they have a more limited choice of paths.
- We need to compare with the work of Kompella and others, and we need to test the theoretical CSTs in our simulation as well.
- We will need to make compromises to reduce the running time of the algorithms in the face of the growing scale of network communication.
- We need to support upcoming Scheme 2 features such as sharing relationships between channels and routing within a partition of resources. [FerGup93]

- We need to look at real-life network configurations to ensure that we are not over-engineering a solution. Simple networks may only require simple solutions. Real-life networks do not all have point-to-point links, and using such a model may restrict our effectiveness.

9.0 Acknowledgments

The author wishes to acknowledge Mark Moran as the progenitor of many of the ideas in this paper. He also wishes to acknowledge the contributions of Prof. Domenico Ferrari, Amit Gupta, Wendy Heffner, Kim Keeton, Ed Knightly, Ginger Ogle, and Prof. Abhiram Ranade to this project. Funding for this project was provided by Digital Equipment Corporation's Graduate Engineering Education Program.

10.0 References

[Alm92] "The Almagest," Manual for Ptolemy, Version 0.4. University of California at Berkeley, 1992.

[CoLeRi90] Cormen, T., et al., Introduction to Algorithms, C. 1990, The MIT Press, Cambridge, Massachusetts.

[DiGen93] DiGenova, P., "Comunicazione Real-Time su Reti di Calcolatori: Introduzione ed Analisi dell'Approccio Tenet di Berkeley," M.S. Report, University of Bologna, Italy, December 1993.

[DoaLes93] Doar, M., and Leslie, I., "How bad is Naive Multicast Routing?" Proceedings of the I.E.E.E. INFOCOM Conference, San Francisco, 1993.

[FeBaZh92] Ferrari, D., Banerjea, A., and Zhang, H. "Network Support for Multimedia - A Discussion of the Tenet Approach," Tech. Rept. TR-92-072, International Computer Science Institute, Berkeley, CA, October 1992.

[Fer90] Ferrari, D., "Client Requirements for Real-Time Communication Services," Proc. of the International Conf. on Information Tech.," I.E.E.E. Communications Magazine, vol. 28, n. 11, pp. 65-72, November 1990.

[FerGup93] Ferrari, D., Gupta, A., "Resource Partitioning for Real-time Communication," submitted for publication, Tenet Group, University of California, Berkeley, June 1993.

[FerVer90] Ferrari, D., and Verma, D., "A Scheme for Real-Time Channel Establishment in Wide-Area Networks," I.E.E.E. Journal on Selected Areas in Comm., vol. 8, n. 3, April 1990, pp. 368-379.

[Gupta93] Gupta, A., personal communication, February 1993.

[Hef94] Heffner, W., et al. "Scheme2 Design Document," unpublished, Tenet Group, University of California at Berkeley, 1994.

[HeOgWi93] Heffner, W., Ogle, G., and Widyono, R., "Real-Time Channel Establishment for Multicast Traffic," CS268 class project, University of California, Berkeley, May 1993.

[Jain91] Jain, R., The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, C. 1991, John Wiley & Sons, Inc., New York.

[KniVen93] Knightly, E., and Ventre, G. "Galileo: a Tool for Simulation and Analysis of Real-time Networks," Tech. Rept. TR-93-008, International Computer Science Institute, Berkeley, CA, March 1993.

[Komp93] Kompella, V., "Multicast Routing Algorithms for Multimedia Traffic," PhD Dissertation, University of California, San Diego, 1993.

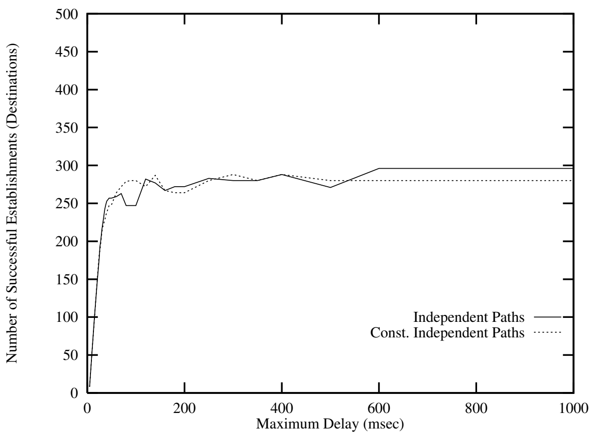
[LeeMes93] Lee, E., and Messerschmitt, D. "An Overview of the Ptolemy Project," Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1993.

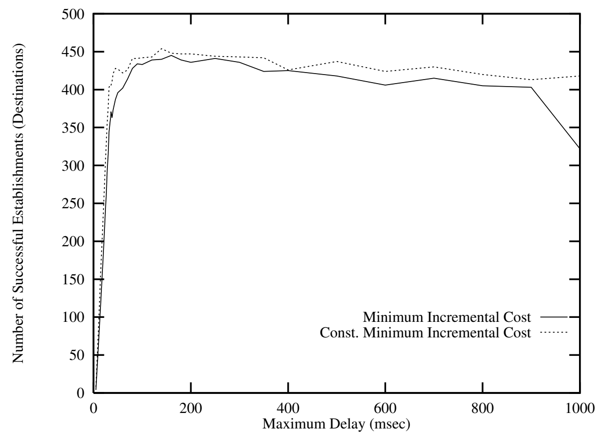
[LiuL73] Liu, C. L., and Layland, J.W., "Scheduling algorithms for multiprogramming in a hard real time environment," J. ACM, Vol. 20, n. 1, pp. 46-61, January 1973.

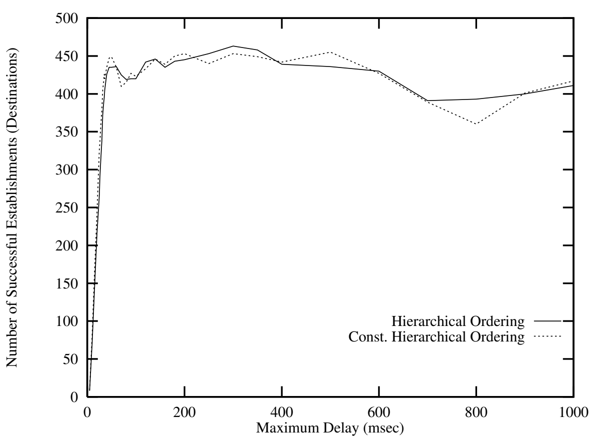
[MiESZ93] Mitzel, D., et al., "An Architectural Comparison of ST-II and RSVP," Unpublished draft, 1994.
[Moran93] Moran, M., personal communication, October 1993.
[ParFer93] Parris, C., and Ferrari, D., "A Dynamic Connection Management Scheme for Guaranteed Performance Services in Packet-Switching Integrated Services Networks," Tech. Rept. TR-93-005, International Computer Science Institute, Berkeley, CA, January 1993.
[Waxman93] Waxman, B., "Performance Evaluation of Multipoint Routing Algorithms," Proceedings of the I.E.E.E. INFOCOM Conference, San Francisco, 1993.
[Widyon93] Widyono, R., "Evaluation of Routing Algorithms for Connection-Oriented Multicast Channels," CS266 class project, University of California, Berkeley, May 1993.
[ZhaFer92] Zhang, H., and Ferrari, D., "Rate-Controlled Static-Priority Queuing," Tech. Rept. TR-92-003, International Computer Science Institute, Berkeley, CA, January 1992.

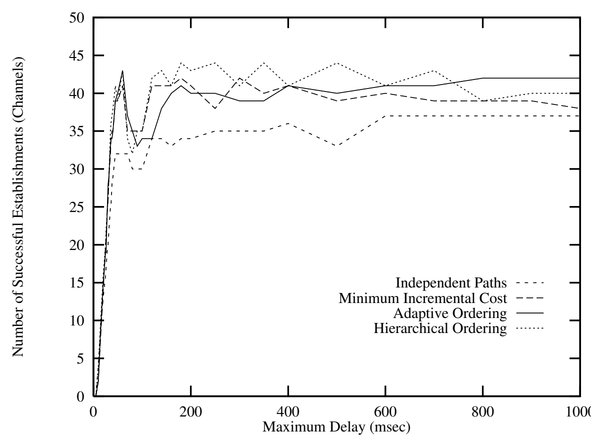
11.0 Appendix A: Graphs

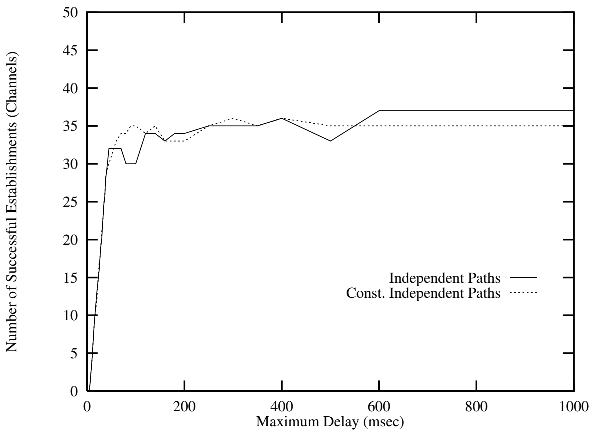
The graphs in this appendix complete the logical set of graphs, some of which were presented in the paper.

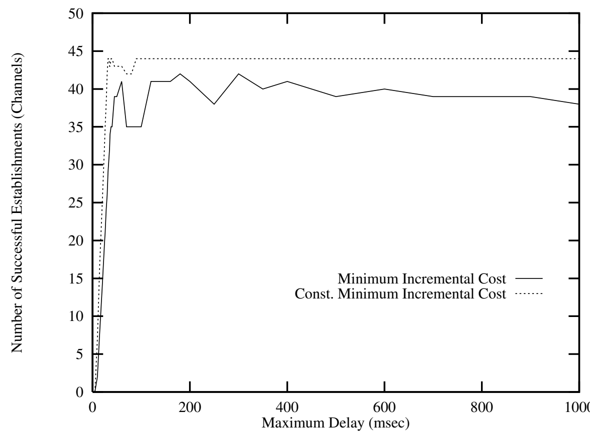


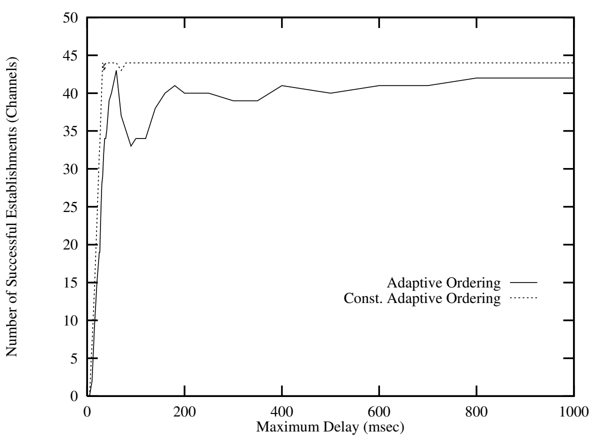


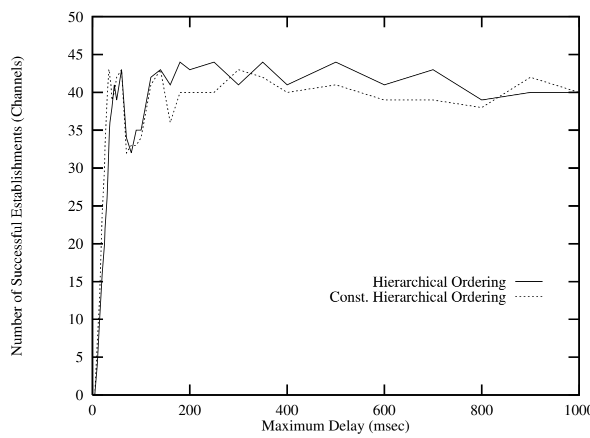





Number of Successful Establishments (Channels)
Maximum Delay (msec)
Independent Paths
Const. Independent Paths
0
5
10
15
20
25
30
35
40
45
50
0
200
400
600
800
1000


Number of Successful Establishments (Channels)
Maximum Delay (msec)
Minimum Incremental Cost
Const. Minimum Incremental Cost
0
5
10
15
20
25
30
35
40
45
50
0
200
400
600
800
1000


Number of Successful Establishments (Channels)
Maximum Delay (msec)
Adaptive Ordering
Const. Adaptive Ordering
0
5
10
15
20
25
30
35
40
45
50
0
200
400
600
800
1000


Number of Successful Establishments (Channels)
Maximum Delay (msec)
Hierarchical Ordering
Const. Hierarchical Ordering
0
5
10
15
20
25
30
35
40
45
50
0
200
400
600
800
1000