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

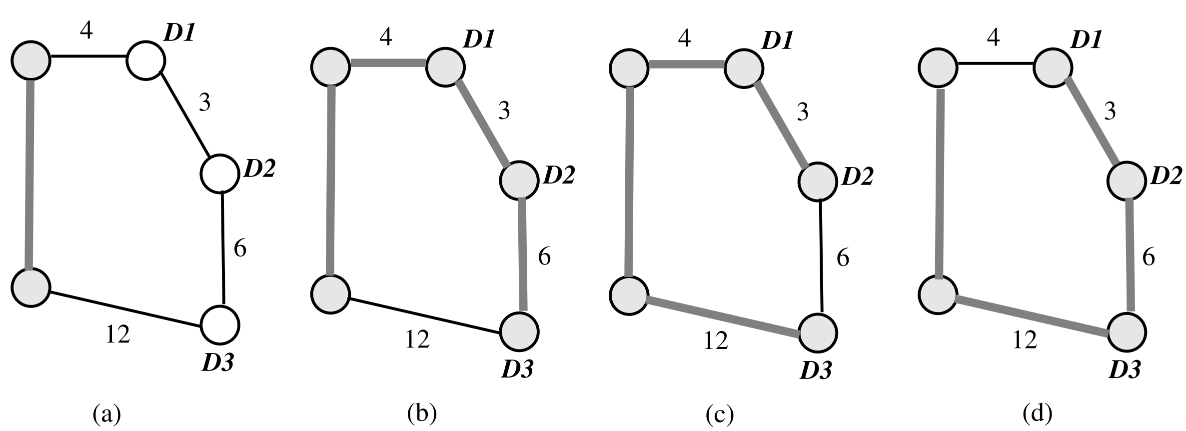

Figure 1.

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

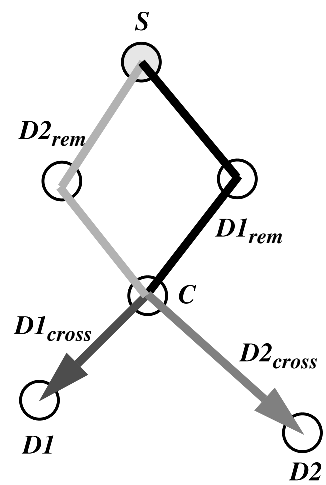

Figure 2.

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 $\boldsymbol{D1}_{\mathrm{cross}}$, and the remainder of that path to the source has cost $\boldsymbol{D1}_{\mathrm{rem}}$. Similarly, the path between ***D2*** and ***S*** has segments of cost $\boldsymbol{D2}_{\mathrm{cross}}$ and $\boldsymbol{D2}_{\mathrm{rem}}$. 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 $\boldsymbol{D1}_{\mathrm{rem}} < \boldsymbol{D2}_{\mathrm{rem}}$—then we could construct a new route for ***D2***, using the remainder of ***D1***'s path from ***C,*** with cost $\boldsymbol{D2}_{\mathrm{cross}} + \boldsymbol{D1}_{\mathrm{rem}}$. The cost would be less than ***D2***'s original cost $\boldsymbol{D2}_{\mathrm{cross}} + \boldsymbol{D2}_{\mathrm{rem}}$, 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).