Scalable Overlay Multicast Tree Construction for QoS-Constrained Media Streaming

Gabriel Parmer, Richard West, Gerald Fry
Computer Science Department
Boston University
Boston, MA 02215
{gabep1,richwest,gfry}@cs.bu.edu

Abstract

Overlay networks have become popular in recent times for content distribution and end-system multicasting of media streams. In the latter case, the motivation is based on the lack of widespread deployment of IP multicast and the ability to perform end-host processing. However, constructing routes between various end-hosts, so that data can be streamed from content publishers to many thousands of subscribers, each having their own QoS constraints, is still a challenging problem. First, any routes between end-hosts using trees built on top of overlay networks can increase stress on the underlying physical network, due to multiple instances of the same data traversing a given physical link. Second, because overlay routes between end-hosts may traverse physical network links more than once, they increase the end-to-end latency compared to IP-level routing. Third, algorithms for constructing efficient, large-scale trees that reduce link stress and latency are typically more complex.

This paper therefore compares various methods to construct multicast trees between end-systems, that vary in terms of implementation costs and their ability to support per-subscriber QoS constraints. We describe several algorithms that make trade-offs between algorithmic complexity, physical link stress and latency. While no algorithm is best in all three cases we show how it is possible to efficiently build trees for several thousand subscribers with latencies within a factor of two of the optimal, and link stresses comparable to, or better than, existing technologies.

1 Introduction

This work addresses the problem of delivering real-time media streams on an Internet-scale, from one or more publishers to potentially many thousands of subscribers, each having their own service constraints. Such constraints may be in terms of latency bounds on the transfer of data from publishers to subscribers, but may also encompass jitter, loss and bandwidth requirements. Target applications for this work include multimedia streaming of live video broadcasts (e.g., Internet television), interactive distance learning and the exchange of time-critical data sets in large-scale scientific applications [20].

In recent years, there have been a number of research efforts focused on content distribution using end-system, or application-level, multicast techniques [6, 25, 7, 2, 29, 13, 14]. Such work is partly motivated by the lack of widespread deployment of IP multicast (at the network-level) and the inability of routers to employ application-specific stream processing services. In most cases, meshes or *overlays* form a logical interconnect between end-hosts, providing the basis for multicast trees or routes to deliver data in a scalable manner [12, 10, 21, 3, 5]. However, the problem with building routes between end-hosts using logical overlays is that data may be duplicated at the physical network level, increasing *physical link stress*. Similarly, the end-to-end delay of data transported along a logical overlay path between a pair of hosts might be significantly larger than an equivalent unicast path at the physical network level, thereby yielding a corresponding *relative delay penalty* [1] greater than 1.0.

Many approaches attempt to carefully match the overlay topology to the underlying physical network, to decrease the relative delay penalty and link stress [26]. Similarly, a number of end-system multicast techniques attempt to reduce physical link stress while maximizing bandwidth, but few have actually focused on the construction of communication paths between hosts that provide quality-of-service (QoS) guarantees, even in terms of latency (or delay) bounds. Of the few approaches that do consider QoS (e.g., RITA [25],

[1] Some researchers refer to a similar term called *stretch* that is the ratio of the cost of routing over an overlay tree, to the cost of routing over a shortest path tree at the network-level, e.g. using IP multicast.

OMNI [3], and ZIGZAG [23]), our work differs by focusing on a study of several approaches to build trees that vary in terms of their complexity, and ability to limit both link stress and relative delay penalties. Unlike the work on RITA, we also consider degree constraints of multicast tree nodes as dictated by the overlay topology that logically connects end-systems. Such degree constraints limit the fanout of tree nodes, thereby making tree depths potentially larger, especially for large-scale trees.

Given that we envision a distributed system of end-hosts built on the scale of the Internet, some method is required to maintain routing state and connectivity between hosts. Peer-to-peer (P2P) systems [22, 15, 19, 30, 4] tackle this problem by constructing distributed hash tables to lookup and retrieve content in $O(\lg(M))$ hops, where routing state at each peer is also $O(\lg(M))$ in size with respect to the number of peers, M. Implicitly, these P2P systems form logical interconnection networks, or overlay topologies, that include hypercubes [22], more generalized toroidal structures [15, 1] and *k-ary n-cubes* [19]. With this in mind, our approach for constructing scalable end-system multicast trees assumes a distributed system of hosts that are connected via a *regular* overlay topology. Specifically, no single host has a global snapshot of the entire system of M hosts, but may find a path between itself and another host within $O(\lg(M))$ hops.

In prior work we have studied the use of k-ary n-cubes and de Bruijn graphs for routing multimedia streams [8, 9, 24]. In this paper, we simply assume such overlay topologies exist for the basis of multicast tree construction, such that no tree can have a *fanout* greater than the *degree* of the corresponding overlay.

The problem addressed by this paper, then, is how to construct communication paths between end-systems in an efficient manner while: (1) minimizing both relative delay penalty and physical link stress, and (2) achieving the service requirements of most (if not all) subscribers of a given published media stream. We compare various methods to construct multicast trees between end-systems, that vary in terms of implementation costs and their ability to support per-subscriber QoS constraints. We describe several algorithms that make trade-offs between algorithmic complexity, physical link stress and latency. While no algorithm is best in all cases we show how it is possible to efficiently build trees for several thousand subscribers with latencies within a factor of two of the optimal, and link stresses comparable to, or better than, existing technologies.

The rest of the paper is organized as follows. In Section 2, various methods for multicast tree construction are discussed. Simulation results and analyses of each tree construction method are described in Section 3. This is followed by a brief overview of related work in Section 4. Finally, conclusions are drawn, and future work is outlined in Section 5.

2 Multicast Tree Construction

Our approach for constructing scalable end-system multicast trees assumes a distributed system of hosts that are connected via a *regular* overlay topology. We make this assumption because of the decentralized manner in which hosts need only maintain *partial* state about the entire system. In practice, we intend to use a regular overlay topology for publishers to announce the availability of streams, or media *channels*, using distributed hash tables (DHTs), and for various end-systems to discover which channels are active so they may selectively subscribe to them. With this in mind, we now describe a number of algorithms for building (or effectively embedding) multicast trees in overlays, that vary in terms of running times, link stress and relative delay penalties. For the purposes of this paper, the only importance of the overlay is that it corresponds to a regular structure whose node degree, or fanout, is limited to $O(\lg(M))$ for a system of M end-hosts.

2.1 Methods for Establishing Locality

Each multicast tree is constructed such that the link stress and relative delay penalties are minimized, with the intent of using end-systems to produce a routing structure that closely approximates that of IP multicast. Various metrics can be used to achieve this goal [18], although in this work we focus on the following two:

1. *Latency*, L, which is measured by using ICMP to send "ping" messages between two hosts to calculate an approximation of distance. This metric has the advantage of being simple and having a low overhead, both on the hosts and on the underlying network. Latency is a relation between two hosts, $L(H_1, H_2)$ and specifies the time it takes to traverse the shortest path between them.

2. *Traceroute*, T, which is measured by sending ICMP packets towards the destination with time to live (TTL) increasing from 0 to the hop distance to the destination. In this way, we can determine a list of physical routers through which a communication path between two hosts will traverse. Note that all modified ICMP messages can be sent out in parallel, enabling a traceroute measurement to be conducted in the same amount of time it takes to carry out a normal ping. Taking the measurements in parallel will not give us information about the order of routers traversed along a path. The traceroute metric, $T(H_1, H_2)$, returns the set of all routers traversed along the shortest path between the hosts.

2.2 Channel Subscription Algorithms

When a new host joins the system and wishes to subscribe to a multicast channel, a specific algorithm dictates the methodology for where this host is placed in the tree, and how it gets there. We term these policies *subscription* algorithms. The *subscription* algorithms determine not only the amount of time between each subscription request and when a stream is received, but also the ability of a tree to meet the QoS requirements of each subscriber. *Subscription* algorithms must make a difficult compromise: a very complex *subscription* algorithm which attempts to place a newly subscribing host at the optimal location in the tree may take an unacceptably large amount of time to place that host and might impose significant overhead in terms of cost measurements, whereas an extremely naive *subscription* algorithm which randomly assigns new hosts to positions in the tree could very quickly place nodes, but the resulting tree would be quite inefficient.

The first consideration, the subscription delay, is important because the system must scale not only in terms of the amount of hosts currently constituting a multicast tree, but also in terms of arrival (and departure [2]) rates of hosts [16]. The second consideration, the quality of the resulting multicast tree, is important to be able to scale in terms of amount of hosts supported, and to allow some form of guarantees to be made regarding bandwidth usage, and latency of delivery of content. A successful *subscription* algorithm will strike some balance between subscription complexity and resulting tree quality.

We present different policies, each of which has different goals and makes different trade-offs. Each of the *subscription* algorithms is separated into two parts. First, the subscription policy which determines, generally, where in the tree a new host begins searching for it's placement, and where it looks subsequently using a level-by-level traversal of the tree. Second, at each level of the tree that is examined, the children are evaluated and compared with the new subscriber, and conclusions are drawn regarding which subtree should be evaluated next. The policies which evaluate the children of a certain host are defined by two functions, `best` and `worst`. These functions identify, respectively, the best and worst child (as defined by the specific policy), and this information can be used by the subscription policy to make decisions regarding where to attempt to attach the next new subscriber.

In what follows, we describe the different policies we have investigated. Section 2.3 describes the subscription policies and Section 2.4 covers the different methods for choosing the best and worst children of a host. Figure 1 illustrates a simple example of how tree nodes are labeled, and this notation will be used throughout the rest of the text. The figure depicts a multicast tree consisting of the parent, P, the set of all of P's children denoted by $C_P = \{C_{P1}, C_{P2}, C_{P3}\}$, and a new subscriber, S. In general, we use C_{Pi} to represent the ith child of P, where $i \leq MAXFANOUT_P$ for the maximum fanout of P. Recursively, we then use C_{P_ij} to denote the jth child of C_{Pi}.

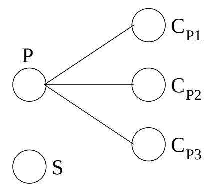


Figure 1. The relationship between hosts in the multicast tree, and their labels. S is a new subscriber, attempting to join the tree.

2.3 Subscription Policies

As stated in Section 2.2, the subscription policies govern where a host which wishes to join a multicast tree will start looking, and where it will look from there as it progressively converges on its eventual location. The first subscription policy is "bubble out" and can be seen in Figure 2. The "bubble out" policy always begins at the publisher of the data-stream and, as dictated by the best and worst functions, decides which child to recursively consider next. In this manner, a new node will cause a "bubble out" type subscription, whereby at each level in the tree a child is chosen and the algorithm is run recursively until a leaf is reached, where the unsubscribed host is added. More specifically, a new subscriber, S, attempts to join the multicast tree by contacting the publisher, P. The first iteration of the "bubble out" subscription policy will reflect this. The best and worst children of P are chosen via the `best` and `worst` functions, respectively. These functions consider the existing set of P's children, C_P, as well as S.

If it is possible to add S to the set of children of P without exceeding the fanout limit of the tree, then the `worst` function returns nil. If the worst is S itself, then it has been decided that S is not better than any of the current children. In this case, S is added to the best child's (i.e., B's) subtree by recursively calling the "bubble out" subscription policy with arguments S and B. However, if it is decided that the worst host, W, is one of P's current children (such that $W \neq S$), the implication is that S would improve the tree

[2]Here, we don't consider hosts leaving the system, as the focus is on efficient tree construction for a set of subscribers. However, in practice, tree membership will change dynamically as end-systems join and leave. We envision streams lasting up to several hours in our system, during which membership changes are assumed infrequent after the initial set of subscriptions. Previously published methods addressing host departure can be used [29] in our scheme.

if it replaced W as one of P's children. Thus, the logical positions of W and S are swapped, and the worst child is recursively bubbled out into the best node's subtree.

```
bubble_out_subscribe(S, P)
    B = best(P, S)
    W = worst(P, S)

    if (W = nil)
        add S to P's children
    else if (W ≠ S)
        swap(S, W)
        bubble_out_subscribe(W, B)
    else
        bubble_out_subscribe(S, B)
```

Figure 2. The "bubble out" subscription policy.

```
bubble_in_subscribe(S, P)
    reverse path along P2P network from S, towards
    P until an existing subscriber, S', to P is found

    if (S' is a leaf)
        bubble_out_subscribe(S, parent_S')
    else
        bubble_out_subscribe(S, S')
    swap_towards_pub(S, P)

swap_towards_pub(S, P)
    if (parent_S ≠ P and
        L(S, parent_parent_S) <
        L(parent_S, parent_parent_S)

        swap(S, parent_S)
        swap_towards_pub(parent_S, P)
```

Figure 3. The "bubble in" subscription policy.

It should be observed that using this subscription policy, the publisher of the data-stream is always contacted and measurements are always made starting from P. The cost measurements described in Section 2.1 are not particularly heavy weight, but we do not wish to put any restrictions on what can constitute a cost method. Thus we must assume that in certain circumstances we might want to mitigate the large amount of cost measurements made near the root of the tree. This is the motivation for the next subscription policy, "bubble in", described in Figure 2.3.

"Bubble in" is based on the assumption that there is some distributed hash table (in our case, based on a k-ary, n-cube) using which we can traverse a logical path towards P. A new subscriber, S, will follow a reverse path towards P until an existing subscriber, S' to P is found. At this point, the algorithm runs the normal "bubble out" method, treating S' as the publisher. This will add S to S's subtree. S will then attempt to move in towards P, using the function `swap_towards_pub`.

The function `swap_towards_pub` compares the latency, L, between S and its grandparent, $parent_{parent_S}$, with the latency between the parent of S and its grandparent. If the tree latency can be reduced, S is swapped with its parent, causing a "bubble in" effect towards P. Cost measurements are much more distributed when using this method as opposed to the "bubble out" policy, since the reference point in the tree where measurements are initially made is decided by the reverse path traversal over the peer-to-peer overlay. Results show that after 5000 hosts have subscribed to a tree, an order of magnitude less cost measurements to P are possible with the "bubble in" method.

2.4 Best and Worst Functions

The best and worst functions define if swaps will be made, and which host will be selected to (recursively) subscribe to a specific subtree, during the execution of the subscription algorithm. Each of the methods for finding the best and worst hosts abide by a specific goal, and make trade-offs to achieve that goal. We discuss these goals and trade-offs for each of the following policies considered in this paper:

Latency Onehop is based on the idea that hosts which are closest physically to a publisher (or any subsequent subtree roots), should be logically closest as well. The formal definition for this method is described in Figure 4. The *onehop latency* (henceforth called *onehop*) policy will choose as worst, either S or one of the hosts in C_P prior to S's arrival, whichever has the largest latency to P. A specific goal of this policy is to minimize the maximum latency provided to all hosts in the system, and therefore provide predictable delay bounds to subscribing hosts. Each host, h, in the multicast tree maintains it's *subtree cost*, which is the minimum latency between h and any host whose fanout is less than the maximum fanout in the tree rooted at h. The formal definition is shown in Figure 4. Using this *subtree cost*, the *onehop* policy decides the best child amongst a given set. The child with the lowest subtree cost, thus the lowest tree overhead for any host inserted into that subtree, is the child that is returned by the *onehop* best method.

The *onehop* method has several advantages. Latencies between the tree root and its children are minimized, which in turn has the potential to minimize latencies between the root and hosts further down the tree. Moreover, the onehop

$$\texttt{best}(P,S) = C_{Pi} \mid subtreeCost(P, C_{Pi}) = \min_{\forall j}(subtreeCost(P, C_{Pj}))$$

$$subtreeCost(P, C_{Pi}) = \begin{cases} 0 & if \mid C_P \mid < MAXFANOUT_P \\ L(C_{Pi}, P) + \min_{\forall j}(subtreeCost(C_{Pi}, C_{P_ij})) & otherwise \end{cases}$$

$$\texttt{worst}(P,S) = \begin{cases} nil & if \mid C_P \mid < MAXFANOUT_P \\ C_{Pi} \mid L(C_{Pi}, P) = \max_{\forall j}(L(C_{Pj}, P)) & otherwise \end{cases}$$

Figure 4. Methods for finding the best and worst children using the *latency onehop* policy.

method is relatively simple, requiring only one round-trip time measurement to find the latency between the root of the current subtree and the new subscriber for each level in the tree. The amount of time it takes for a new subscriber to actually be added to the tree is relatively small. However, the *onehop* method does not take link stress into account at all, and the amount of physical links that are concurrently used between multiple different subtrees will contribute to link stress. Further, it is possible that when we swap a current child with a new subscriber, latencies through the corresponding subtree may be increased. For example, if C_{Pi} is very close to all of it's children, and it is found that C_{Pi} is the worst of P's children, S will take C_{Pi}'s place; however, it is possible that even though S could be close to P, it might be very far from all of its children which it recently inherited.

Latency Twohop attempts to avoid the deficiency of *onehop* on the resultant tree. Figure 5 contains the formal definition of this policy. The *twohop* policy not only measures if S is closer to P than any hosts in C_P, but also measures how much the subtree rooted at each child, C_{Pi}, would be affected if S were swapped in to C_{Pi}'s position. If the overall latency to children of C_{Pi} is decreased after swapping C_{Pi} with S, then C_{Pi} is considered as a candidate for the worst child. The child with the largest decrease in latency to its subtree is returned as the worst child. If S does not improve any subtrees, then it is the worst. The best child is found in the same way as in *onehop*. It should be noticed that while *onehop* takes measurements between hosts that are one hop away from P, *twohop* takes measurements which consider hosts that are up to two hops away from P. The main deficiency of this policy is that it must make an extremely large amount of cost measurements. Whereas *onehop* has to make one cost measurement per level in the tree, *twohop* must make one to P, and one to all of the grandchildren of P, or $O(fanout^2)$ per level in the tree.

Closest Latency takes an opposite approach to building multicast trees. Instead of attempting to put physically close hosts logically close to the roots of the subtrees, *closest latency* attempts to place a new host, S, logically close to children that are physically close to S. Therefore, the worst is always S, and the best is that child, C_{Pi}, which is physically closest to S. In this way, S will be recursively subscribed to C_{Pi}'s subtree, and physically close hosts will become logically close. This policy of multicast tree construction is similar to the method in Host Multicast [29].

This policy has the benefit that it attempts to minimize the link stress implicitly by making a correspondence between children's logical and physical locations in relation to each other. Unfortunately, *closest latency* does require more cost measurements and more time per-level than *onehop*. Not only will S have to communicate with P to get a list of children, it will also need to make cost measurements to each of those children. The number of cost measurements per level in the tree is $O(fanout)$. The amount of time taken for measurements at a given level consists of a round trip time to contact the root of the subtree, in addition to time to take cost measurements to all of that host's children. Because *onehop* requires only a cost measurement to the subtree's root, the amount of time taken at each level is less.

Closest Latency Swap attempts to combine *onehop* and *closest latency* and can be seen in Figure 7. The closest child, C_{Pi}, to the subscriber, S, is found in the same way as in the *closest latency* policy. If S is closer in terms of latency to P than C_{Pi}, then the two hosts are logically swapped, with the swapped out host then subscribed to the swapped in host's subtree. In all other cases, the normal *closest latency* policy is utilized. This method has little more overhead than *closest latency*, and produces a better tree.

Closest Traceroute has the same goals as *closest latency*, but goes about finding the closest child to S in a different manner. Figure 8 illustrates this method. Instead of making a latency measurement to each of P's children, a single traceroute measurement is made to P. The traceroute measurement, $T(P, S)$ yields the set of physical (or IP-level) routing hops between hosts P and S. In this case, the child C_{Pi} whose traceroute path has the largest intersection with $T(P, S)$ is deemed the closest to S and returned as best. Other than this it is identical to *closest latency*. Because only one cost measurement is made and it is to the parent, the amount of time spent at each level in the tree is significantly less than that in *closest latency*. However, results show that the traceroute method of approximating which child is closest to S only finds the same child as the *closest latency* method 43% of the time in a tree of 200 hosts, so there is a price to be paid by the approximation.

$$\texttt{best}(P,S) = same\,as\,in\,Figure\,4.$$

$$onehopFurther(P, C_{Pi}, H) = L(P,H) + \Sigma_{\forall j}(L(H, C_{P_i j}) + L(P,H))$$

$$\texttt{worst}(P,S) = \begin{cases} nil & if \mid C_P \mid < MAXFANOUT_P \\ C_{Pi} \mid onehopFurther(P, C_{Pi}, C_{Pi}) - \\ \quad onehopFurther(P, C_{Pi}, S) = \\ \quad \max_{\forall j}(onehopFurther(P, C_{Pj}, C_{Pj}) - \\ \quad onehopFurther(P, C_{Pj}, S)) & otherwise \end{cases}$$

Figure 5. Methods for finding the best and worst children using the *latency twohop* policy.

$$\texttt{best}(P,S) = C_{Pi} \mid L(C_{Pi}, S) = \min_{\forall j}(L(C_{Pj}, S))$$

$$\texttt{worst}(P,S) = \begin{cases} nil & if \mid C_P \mid < MAXFANOUT_P \\ S & otherwise \end{cases}$$

Figure 6. Methods for finding the best and worst children using the *latency closest* policy.

$$\texttt{best}(P,S) = \begin{cases} C_{Pi} & if L(P,S) \geq L(P, C_{Pi}) \mid L(C_{Pi}, S) = \min_{\forall j}(L(C_{Pj}, S)) \\ S & otherwise \end{cases}$$

$$\texttt{worst}(P,S) = \begin{cases} nil & if \mid C_P \mid < MAXFANOUT_P \\ S & if L(P,S) \geq L(P, C_{Pi}) \mid L(C_{Pi}, S) = \min_{\forall j}(L(C_{Pj}, S)) \\ C_{Pi} & otherwise \end{cases}$$

Figure 7. Methods for finding the best and worst children using the *latency closest swap* policy. Notice that `best` and `worst` are opposites, with the addition of the nil case for worst.

Closest Traceroute Swap is illustrated in Figure 9, and makes the same modifications to the *closest traceroute* algorithm that the *closest latency swap* algorithm makes to *closest latency*.

3 Results and Analysis

In this section, we compare the different subscription algorithms, using simulations involving the GT-ITM software for generating transit-stub physical topologies [28]. Unless otherwise specified, each physical network is created with 5050 routers, 10 transit domains, 10 transit nodes per transit domain, 5 stub domains attached to each transit node, and 10 nodes in each stub domain. All hosts in the system are assigned to a random router, with the possibility that multiple hosts are assigned to the same router. One of the hosts is chosen to be the publisher. A set of subscribing hosts run the chosen subscription algorithm in a random order, which is indicative of a real application scenario where hosts may subscribe to published streams at arbitrary times. The properties of the resulting tree are measured and reported. Though the tree fanout (or maximum number of children) can be configured on a per-host basis, we choose a value of 12 for all hosts unless otherwise specified. All results are averaged over experiments run on three transit-stub graphs, with each graph being used twice for randomly chosen publishers and subscribers.

3.1 Comparison of Different Subscription Algorithms

One of the most important metrics, to compare the performance of different subscription algorithms, is the *relative delay penalty* for each subscriber in the resultant multicast tree. The relative delay penalty (or, simply, delay penalty) for each subscriber host, S, is the ratio of the latency of the tree path between the publisher, P, and S to the latency of the shortest unicast path between P and S (using, e.g., IP unicast). A delay penalty of 1.0 would imply the tree latency between P and S is the same as the latency of unicast routing. Although this would be ideal, it is unrealistic in practice, due to trees being constructed at the end-host level rather than physical network level. Figure 10(a) shows the cumulative distribution function of the delay penalty for each of the subscription algorithms for a population of 5000 hosts. The values on the y-axis represent the percentage of subscribers with a delay penalty of no more than the corresponding value on the x-axis.

Efficient use of network bandwidth is also an important metric for end-system multicast, particularly for the purposes of multimedia streaming to potentially many thousands of subscribers. We measure the impact of a multicast tree on link bandwidth by computing its *link stress*. The link stress is a measure of the average number of times a physical network link is traversed when delivering content along the tree paths from a publisher to each and every sub-

$$\texttt{best}(P,S) = C_{Pi} \mid \; \mid T(P,C_i) \cap T(P,S) \mid = \max_{\forall j}(\mid T(P,C_j) \cap T(P,S) \mid)$$

$$\texttt{worst}(P,S) = \begin{cases} nil & if \mid C_P \mid < MAXFANOUT_P \\ S & otherwise \end{cases}$$

Figure 8. Methods for finding the best and worst children using the *traceroute closest* **policy.**

$$\texttt{best}(P,S) = \begin{cases} C_{Pi} & if L(P,S) \geq L(P,C_{Pi}) \mid \; \mid T(P,C_i) \cap T(P,S) \mid = \max_{\forall j}(\mid T(P,C_j) \cap T(P,S) \mid) \\ S & otherwise \end{cases}$$

$$\texttt{worst}(P,S) = \begin{cases} nil & if \mid C_P \mid < MAXFANOUT_P \\ S & if L(P,S) \geq L(P,C_{Pi}) \mid \; \mid T(P,C_i) \cap T(P,S) \mid = \max_{\forall j}(\mid T(P,C_j) \cap T(P,S) \mid) \\ C_{Pi} & otherwise \end{cases}$$

Figure 9. Methods for finding the best and worst children using the *traceroute closest swap* **policy.**

scriber. A link stress of 1.0 is perfect and means that each network link is used only once in the dissemination of the data-stream. As with the relative delay penalty, an efficient end-system multicast tree should attempt to minimize the link stress. Figure 10(b) shows the cumulative distribution function of the link stress for each subscription algorithm.

In addition to link stress and relative delay penalty, it is also important to consider the number of cost measurements needed to construct a scalable end-system multicast tree. This is particularly important when a tree needs to be built or modified quickly in the presence of flash crowds (or bursts of subscriptions). The number of cost measurements taken per host in the system, after all 5000 hosts have subscribed, is represented as a cumulative distribution function in Figure 10(c).

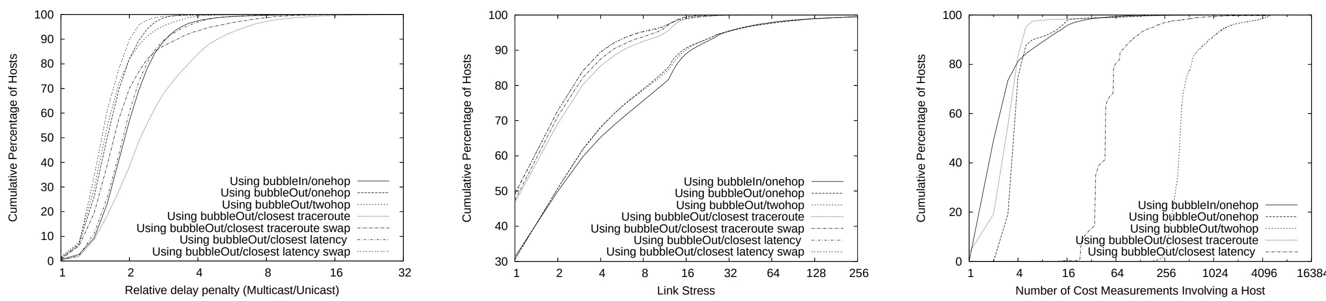
Discussion: We see that the *twohop* method, which takes a large amount of cost measurements does not achieve significantly better delay penalties than *onehop*, which takes very few cost measurements. It appears that the complexity added to make *twohop* does not translate into better performance. This confirms that the rough heuristic which is the basis for *onehop* – that swapping in hosts that are closer to the publisher – is valid. Also, it is apparent that both the *closer latency* and *closer traceroute* policies benefit by swapping nodes that are closer to the subtree root with the closest child. With this optimization, both *closer latency swap* and *closer traceroute swap* are able to not only put physically close children near each other in the resultant tree, but are also able to make a physical correspondence between the subtree roots and their children.

The "bubble in" *onehop* method demonstrates a lower maximum and average number of cost measurements than the other methods, but does not perform notably well in either delay penalty or link stress. The policies which require $O(fanout)$ cost measurements per level (such as *closer latency*) in the tree demonstrate a significant increase in the number of cost measurements over methods such as *onehop*. Recall that this will affect not only the stress on the system induced by cost measurements, but also on the amount of time it takes for a new host to complete its subscription request. These methods, however, do incur significantly less link stress on the system than *onehop*. The *closest traceroute* method, though it makes few cost measurements, does not approximate locality in terms of latency between children well enough. However, *closest traceroute swap* seems to make the approximation much more reasonable and achieves a respectable delay penalty while providing a low link stress.

3.2 Scalability

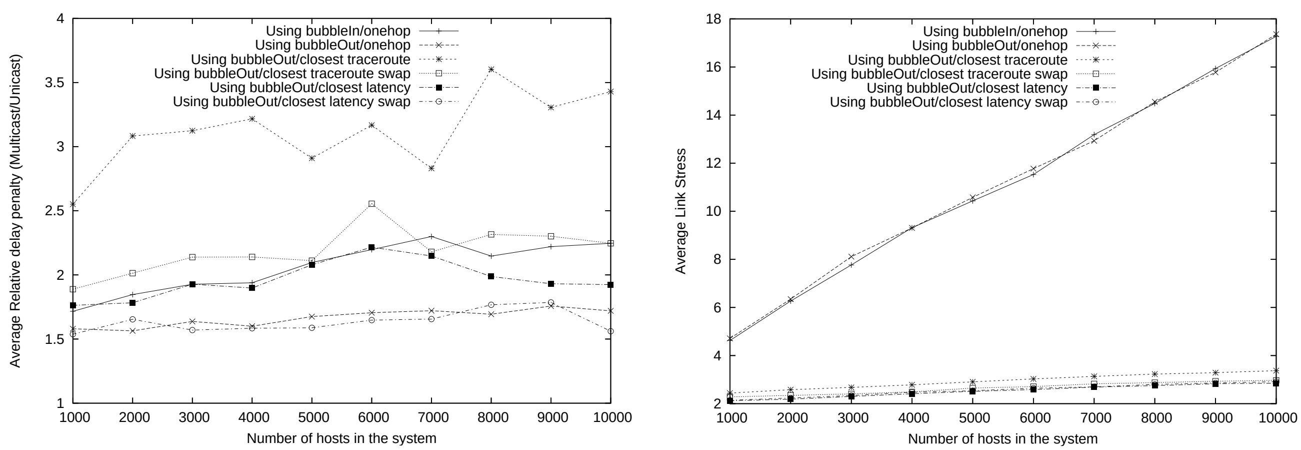
To understand the scalability of each of the subscription algorithms, simulations are run on population sizes ranging from 1000 to 10000. It is important that each of the policies scale in terms of latency of delivery to each host and also in terms of the bandwidth consumed. The delay penalty results for the different populations sizes are shown in Figure 11(a). With the exception of *closest traceroute*, all of the policies seem to increase slowly enough to imply scalability in terms of population size. Some of the delay penalties seem to decrease as the host size grows past 6000. It appears that as the amount of subscribed hosts exceeds the amount of routers in a system, the quality of the tree improves for some policies. This has implications for deployment in corporate environments where the number of employees might overshadow the number of routers. The *onehop* and *closest latency swap* policies demonstrate an extremely low delay penalty which is consistent across population sizes. All conclusions drawn in Section 3.1 are reinforced, and we can see that the delay penalty of many of the policies appears to scale well in terms of population size.

Figure 11(b) illustrates the link stress. All the methods which attempt to map physically close children to logically close positions (i.e., within a few logical hops) in the tree, demonstrate extremely low link stress. However, *onehop* (using "bubble out") and *onehop bubbleIn*, which only attempt to make a correlation between subtree roots and their children have significant link stresses. This is due



(a) Relative Delay Penalty (b) Link Stress (c) Number of Cost Measurements

Figure 10. Subscription algorithm comparison.



(a) Relative Delay Penalty (b) Link Stress

Figure 11. Scalability of the different subscription algorithms.

to the "cross-talk" between different subtrees. That is, different subtrees might utilize common physical links. This "cross-talk" is limited in the policies which cluster children together because each subtree will ideally represent non-overlapping subsections of the entire network.

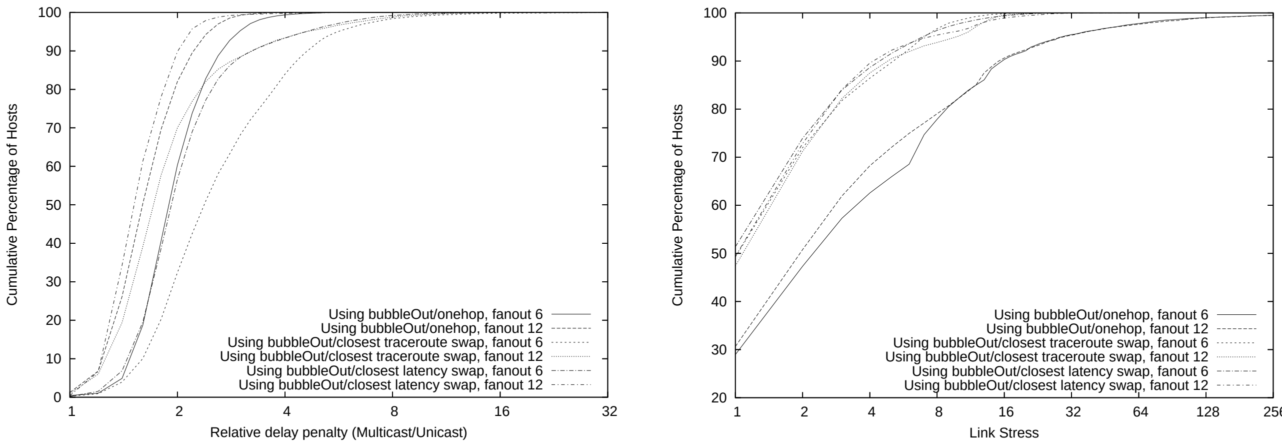
3.3 Variation of Fanout

As mentioned before, the fanout of the tree is, for simplicity, set to 12 in all of the preceding experiments. It is important, however, to understand the effect of fanout on the efficiency of the multicast tree. Figure 12 demonstrates this effect. Multicast trees are constructed using both a fanout of 6 and a fanout of 12, and the cumulative distribution functions for both delay penalty (Figure 12(a)) and link stress (Figure 12(b)) are measured. In these graphs, only a subset of the original subscription policies are tested, which are deemed most interesting.

The delay penalty plots indicate that a larger fanout is more beneficial to the construction of trees with better latency characteristics. This is due to two reasons: (1) as the fanout increases, the `best` and `worst` functions can see more children and can therefore make more informed decisions regarding which child is the best and which is the worst, and (2) because the depth of the tree is less ($log_{12}N$ instead of $log_6 N$ for a population size, N). The change in fanout does not seem to significantly impact the link stress. This is perhaps non-intuitive because as the fanout increases, it would seem that the link stress around that root would also increase. However, both of the methods that attempt to attach the subscriber, S, to a close child will simply have more information about which child to connect to S. In the case of *onehop*, the same "cross-talk" between subtrees will exist regardless of how many children each node has, and this is the primary contributor to its link stress.

3.4 Soft Real-Time Constraints

Given that overlay multicast techniques route between end-systems to deliver content from publishers to subscribers, they lack control at the network level to make hard real-time service guarantees. However, it is possible to use



(a) Relative Delay Penalty

(b) Link Stress

Figure 12. Comparison of the effect of different fanout values.

overlay multicasting to provide soft real-time guarantees to subscribers (e.g., to guarantee a significant percentage of deadlines are met on the delivery of a media stream). With this in mind, we study the ability of each tree construction policy to meet subscriber-specified delay bounds (or latency constraints). For each subscriber, S, latency constraints are generated uniform and randomly in the range $[l_{lower}, l_{upper}]$, where l_{lower} is the unicast latency between publisher, P, and S, and l_{upper} is five times the average unicast latency between all subscribers and P. The factor of five was chosen because it is the upper bound on the average depth of the multicast trees constructed using each of the subscription algorithms. The delivery latency over the multicast tree is computed and compared to it's latency constraint. A $success$ is recorded if the achieved cost does not surpass the constraint. The total number of $successes$ is divided by the population size to obtain a $success$ ratio. Figure 13(a) shows the success ratios of the various tree construction methods.

For each host, S, with a corresponding latency constraint, c, and delivery latency over the multicast tree, D_S, a normalized lateness value, $L(S, c)$ is calculated using the following formula:

$$L(S,c) = \begin{cases} 0 & if D_S \leq c \\ \frac{D_S - c}{c} & if D_S > c \end{cases}$$

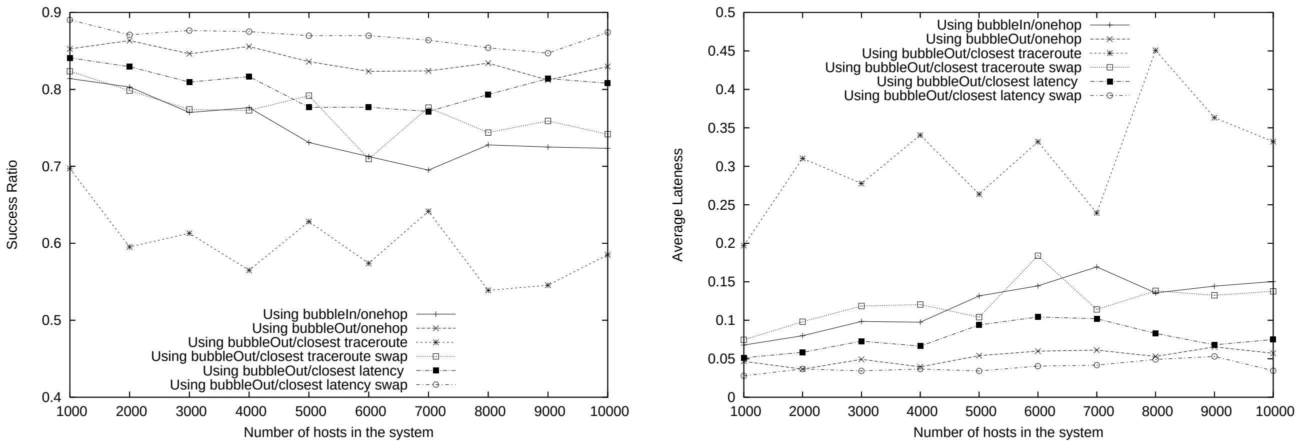
The lateness values are normalized in order to eliminate bias towards subscribers with large latency constraints, relative to the other hosts, and all hosts with satisfied constraints are assigned a normalized lateness of zero. Figure 13(b) shows the normalized lateness values for each subscription algorithm.

As expected, those policies which demonstrate low delay penalties meet the most constraints and have the lowest lateness averages. It should be noted that even those hosts which cannot make their constraints, miss them by relatively small amounts. This improves on previous results [8] which use "greedy-based routing" (rather than explicit multicast trees), to deliver data over a peer-to-peer network based on a k-ary n-cube topology. That said, our earlier greedy-based routing approach was extremely quick at finding paths between a publisher and each subscriber, which may be beneficial for initial setup of paths for media streaming when there are flash crowds.

3.5 Variation of Physical Topology

So as to ensure that results obtained here are not dependent on the size or shape of the physical topology used, we re-run the experiments from Section 3.2 on a different graph with 10100 routers. The graph consists of 25 transit domains, five transit nodes per transit domain, four stub domains attached to each transit node, and 20 nodes in each stub domain. Further, we wish to compare our results with the RITA [25] tree generation method which approaches the problem in a completely different manner. RITA uses landmarking information and a peer-to-peer network to place nodes into the multicast tree. The evaluation of RITA utilizes a physical graph with the same properties and the tests are run for smaller numbers of hosts, with a maximum population size of 2000. It is important to note that RITA measures the *stretch* in latency of the end-system multicast tree compared to the latency of an equivalent shortest path tree at the network-level. This is a slightly different metric than our notion of relative delay penalty. We compare against unicast latency so that we will be measured against the best possible delivery latency for each subscriber.

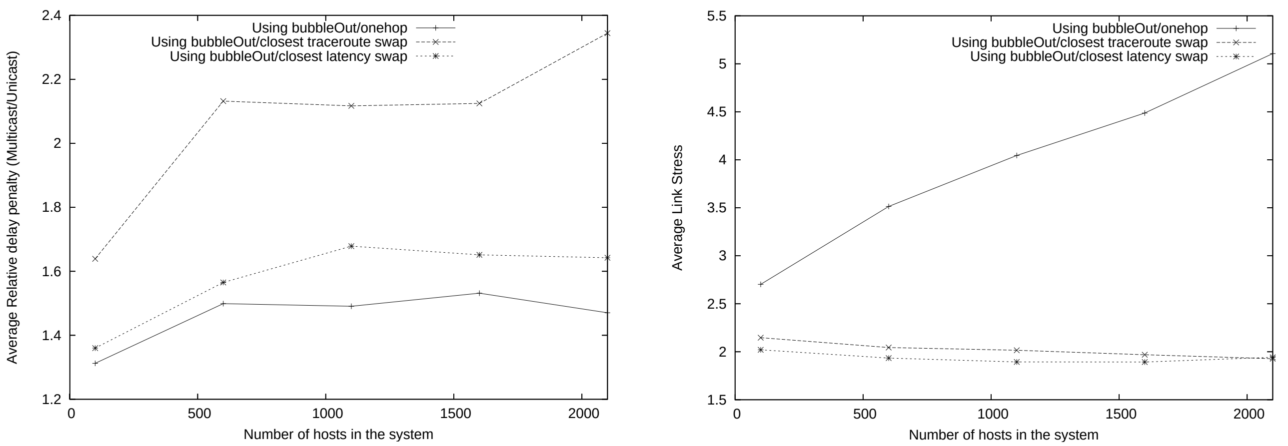
Figure 14(a) shows the delay penalty from a population



(a) Fraction of Deadlines Made

(b) Normalized Lateness

Figure 13. Amount of deadlines made and the average normalized lateness of delivery.



(a) Relative Delay Penalty

(b) Link Stress

Figure 14. Multicast tree construction for smaller population sizes and a larger physical topology.

size of 100 to 2000. Though the delay penalties cannot be compared directly, the values for *closest latency swap* and *onehop* seem to compare favorably to those reported in the work on RITA [25]. Figure 14(b) illustrates the link stress. The *onehop* method seems to achieve a comparable link stress to RITA at 2000 nodes. However, both other policies demonstrate a significantly better average link stress of approximately two. Results show that in terms of delay penalty and link stress, *closest latency swap* achieves average delay penalties comparable to RITA, and link stresses which are better. *closest traceroute swap* seems superior in terms of link stress, but not necessarily in terms of delay penalty. *onehop* is competitive in terms of delay penalty, and link stress (though the authors believe that after 2000 hosts, *onehop*s link stress will be larger than that demonstrated by RITA.)

4 Related Work

There have been several recent methods for building end-system multicast trees and graphs [6, 25, 7, 2, 29, 13, 14, 27, 17] that account for heterogeneous QoS constraints and/or attempt to reduce delay penalties and link stress. Some systems, such as Pastry/Scribe[19, 4] implicitly form distribution trees using logical links that comprise a structured overlay. The advantage of this approach is that it is scalable in the sense that individual participating hosts must only store routing state that is logarithmic in the number of hosts in the system. In addition, many such regular overlay structures can route a message between any pair of hosts in an asymptotically logarithmic number of hops with respect to the number of participants, thus ensuring reasonable routing latencies.

In contrast, we propose algorithms for building dense multicast trees directly, in the context of a limited fanout at each level. Such distribution trees outperform those implicitly built using the union of routes in an overlay such as CHORD [22], CAN [15], and Tapestry [30], but in our scheme a structured overlay is still helpful for routing control messages and may be useful for finding routes around failed nodes. Note that the tree construction algorithms presented in this work produce multicast trees that can be embedded onto a regular overlay graph (e.g., a k-ary n-cube), in which the subscribing hosts are *swapped* into logical positions in correspondence with the tree structure.

NARADA [6] also focuses on tree formation across a logical mesh, that is dynamically constructed by considering all point-to-point connections between end-systems. Logical links are added to, and deleted from, the mesh depending on their benefits (or lack, thereof) to the delivery of content between hosts. However, this method does not achieve the scalability of approaches involving more regularly structured overlays, partly due to the significant number of probe messages involved in constructing the mesh. Our methods do not limit possible connections between hosts in the multicast tree to the particular links available in a pre-constructed overlay network except in the requirement that maximum fanout is held constant. Thus, distribution trees built using the algorithms discussed in our work are not as dependent upon the quality of the initial overlay structure.

Other related work, such as RITA [25], HMTP [29], Yoid [7], and Bullet [11], present approaches that attempt to form multicast trees, without being strictly limited to a fixed logical topology. In these systems a number of methods are proposed for inserting newly subscribing hosts into the tree at positions that fulfill specified QoS constraints and/or result in low-cost data distribution. Some systems, such as OMNI [3], use a set of pre-selected service nodes to form a low-cost distribution tree that connects subscriber hosts. Our approach differs from OMNI in that all hosts are peers, rather than being arranged in a two-tier hierarchy, which allows for greater flexibility in the placement of hosts in a tree structure. While focusing on tree construction algorithms to help meet (statistically, at least) the QoS constraints of subscribers, we explicitly consider the impacts of tree fanout, which has not been addressed by similar work such as RITA. Of all the bodies of work mentioned above, none have provided a comparative study of different tree construction algorithms that emphasize the trade-offs in cost, link stress, delay penalty and ability to meet per-subscriber QoS constraints.

5 Conclusions and Future Work

This paper describes several end-system multicast tree construction algorithms, that vary in terms of cost, relative delay penalty and link stress. We show how it is possible to efficiently build trees that support many thousands of subscribers each with their own QoS constraints. While no algorithm is best in all cases we show how it is possible to efficiently build large-scale trees with latencies within a factor of two of the optimal, and link stresses comparable to, or better than, existing technologies such as RITA and HMTP. Several of our approaches support the swapping of hosts into different tree positions, to adaptively improve the overall tree quality without incurring significant overhead. While no one algorithm is best in all cases for link stress, latency and cost, our *closest latency swap* policy is shown to do reasonably well in all cases. For the policies considered in this paper, experimental results suggest that increasing tree fanout improves the overall latency characteristics without significantly impacting the link stress.

Future work will investigate the costs of constructing efficient tress that minimize link stress, while meeting per-subscriber QoS constraints in the presence of burst joins and departures. Dealing with departures is particularly problematic, as the tree can become partitioned, such that a media stream may not be able to reach its destination. The cost of some of the tree construction algorithms discussed in this paper will also be affected by additional measurements that are needed to repair a tree after one or more hosts departs. During the process of repairing a tree, we will study the ability to continue routing a media stream over alternative links selected from an overlay topology between hosts. Such an approach will alleviate the need to restart the transmission of a stream to one or more subscribers affected by changes to the tree.

References

[1] D. Banerjee, B. Mukherjee, and R. Suryanarayan. The multidimensional torus: Analysis of average hop distance and application as multihop lightwave network. *Proc. ICC '94, Vol. 3*, pages 1675–1680, May 1994.

[2] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proceedings of ACM SIGCOMM*, August 2002.

[3] S. Banerjee, C. Kommareddy, K. Kar, B. Bhattacharjee, and S. Khuller. Construction of an efficient overlay multicast infrastructure for real-time applications. In *IEEE INFOCOM*, San Francisco, CA, April 2003.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC) – Special issue on Network Support for Multicast Communications*, 2002.

[5] Y. Chawathe. *Scattercast: An Architecture for Internet Broadcast Distribution as an Infrastructure Service*. PhD thesis, University of California, Berkeley, December 2000.

[6] Y.-H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *ACM SIGMETRICS 2000*, pages 1–12, June 2000.

[7] P. Francis. Yoid: Extending the multicast Internet architecture, 1999. On-line documentation:
http://www.aciri.org/yoid/.

[8] G. Fry and R. West. Adaptive routing of QoS-constrained media streams over scalable overlay topologies. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, May 2004.

[9] G. Fry and R. West. Dynamic characteristics of k-ary n-cube networks for real-time communication. In *Proceedings of the 5th International Conference on Communications in Computing*, June 2004.

[10] J. Jannotti, D. Gifford, K. Johnson, M. Kaashoek, and J. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, October 2000.

[11] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, volume 37, 5 of *Operating Systems Review*, pages 282–297, New York, Oct. 19–22 2003. ACM Press.

[12] N.J.A.Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[13] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai. Distributing streaming media content using cooperative networking. In *Proceedings of the 12th International Workshop on Network and Operating Systems for Digital Audio and Video (NOSSDAV)*, May 2002.

[14] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: an application-level multicast infrastructure. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, March 2001.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Computer Communication Review*, volume 31, pages 161–172. Dept. of Elec. Eng. and Comp. Sci., University of California, Berkeley, 2001.

[16] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the 2004 USENIX Technical Conference, Boston, MA, USA*, June 2004.

[17] A. Riabov, Z. Liu, and L. Zhang. Overlay multicast trees of minimal delay. In *The 24th International Conference on Distributed Computing Systems, ICDCS*, pages 654–661, 2004.

[18] A. Rodriguez, D. Kostic, and A. Vahdat. Scalability in adaptive multi-metric overlays. In *The 24th International Conference on Distributed Computing Systems, ICDCS*, pages 112–121, 2004.

[19] A. Rowstron and P. Druschel. Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001.

[20] SETI@home: http://setiathome.ssl.berkeley.edu/.

[21] S. Shi and J. Turner. Routing in overlay multicast networks. In *IEEE INFOCOM*, June 2002.

[22] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, D. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb. 2003.

[23] D. Tran, K. Hua, and T. Do. ZIGZAG: an efficient peer-to-peer scheme for media streaming. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, April 2003.

[24] R. West, G. Wong, and G. Fry. Comparison of k-ary n-cube and de bruijn overlays in qos-constrained multicast applications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, June 2005.

[25] Z. Xu, C. Tang, S. Banerjee, and S.-J. Lee. RITA: receiver initiated just-in-time tree adaptation for rich media distribution. In *Proceedings of the 13th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June 2003.

[26] Z. Xu, C. Tang, and Z. Zhang. Building topology-aware overlays using global soft-stat. In *The 23th International Conference on Distributed Computing Systems, ICDCS*, pages 500–508, 2003.

[27] J. Yang. Deliver multimedia streams with flexible qos via a multicast dag. In *The 23th International Conference on Distributed Computing Systems, ICDCS*, pages 126–137, 2003.

[28] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *IEEE INFOCOM*, volume 2, pages 594–602, San Francisco, CA, March 1996.

[29] B. Zhang, S. Jamin, and L. Zhang. Host multicast: A framework for delivering multicast to end-users. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, June 2002.

[30] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001.