Overlay Mesh Construction Using Interleaved Spanning Trees

Anthony Young, Jiang Chen, Zheng Ma, Arvind Krishnamurthy
Department of Computer Science
Yale University

Larry Peterson, Randolph Y. Wang
Department of Computer Science
Princeton University

***Abstract*— In this paper we evaluate a method of using interleaved spanning trees to compose a resilient, high performance overlay mesh. Though spanning trees of arbitrary type could be used to construct an overlay mesh, we focus on a distributed algorithm that computes k minimum spanning trees on an arbitrary graph. The principal motivation behind this strategy is to provide applications with a k-redundant, high quality mesh suitable for demanding applications like A/V broadcast, video conferencing, data collection, multi-path routing, and file mirroring/transfer. We elaborate details of k-MST, pointing out advantages and potential problem points of the protocol, and then analyze its performance using a variety of metrics with simulation as well as a functional PlanetLab implementation.**

Keywords: System design, Experimentation with real networks/Testbeds.

I. Introduction

Over the past several years, considerable work has been done in designing overlay networks to optimize and enable applications over the Internet. Detour [1], for instance, improves routing efficiency by exchanging congestion information between nodes and adaptively routing through overlay paths that correspond to better routes. Resilient Overlay Network (RON) [2] allows distributed applications to perform overlay path selection in an application-specific manner, detect path failures, and recover by routing data through other overlay paths. High-performance streaming media systems [3], [4] enable a pair of nodes to communicate through multiple overlay paths simultaneously. Overlay networks have also emerged as a powerful method for delivering content [5]–[9] and coordinating multi-point communication algorithms. CoopNet [10] streams media and employs striping over multiple overlay trees to enhance both performance and reliability. Byers *et al.* [11] propose a system for performing multi-point transfers across richly connected overlays by judiciously coordinating delivery of subsets of data in a highly distributed and concurrent fashion. Cheng *et al.* [12] address the dual problem of collecting data from several hosts by carefully scheduling the movement of data.

The performance of these applications is highly dependent on the ability to operate over good quality overlay paths. A richly connected overlay network comprised of high quality virtual paths provides the ideal setting for these applications; applications can then react quickly to fluctuating performance, use application-specific path selection, or coordinate concurrent communication over multiple paths.

Proposed in this paper is a mesh-first approach, where the dense graph of all possible overlay links is reduced to a minimal topology composed of k trees. In particular, we focus on a distributed algorithm to compute k Minimum Spanning Trees (k-MST), where edge weights correspond to any one of the standard performance metrics, such as latency, bandwidth, and loss rate. The mesh is constructed using initial estimates of network properties and refined over time. Like other unstructured overlays, such as End-System Multicast [5], we view the tasks of mesh construction and routing as independent. A standard routing protocol is used to propagate the performance characteristics of the selected links.

The primary motivation to construct an overlay mesh of k trees is to ensure the existence of k edge disjoint overlay paths between any two nodes, to promote fault- and performance-tolerance, and to enable path diversity. Though k-MST includes the k best links of every node in the overlay, it does not bound the diameter or the degree of the network. However, it is possible to address these additional constraints with other distributed heuristics that will be discussed in later sections. Also, trees have important properties that simplify aspects of mesh maintenance in the distributed environment. For example, the addition of any link creates a fundamental cycle, the removal of any link along a fundamental cycle restores the tree property, and any edge failure partitions the tree into two components. All these properties can be detected and maintained with limited local knowledge.

Though we have been developing a prototype overlay network that utilizes a k-tree methodology for mesh construction, this paper focuses not on our architecture but primarily on discussion and analysis of the k-tree strategy for high performance mesh applications. We are not interested in using this strategy in peer-to-peer file sharing or distributed object location settings, but are more interested in applications requiring resilient, high performance data delivery like A/V multicast, video conferencing, data collection, multi-path routing, and file transfer. The need for such applications is most pressing in infrastructure-type settings like PlanetLab as well as in large corporations where overlay networks of less than 1000 nodes suffice. In such environments, a large amount of data concern-

Arvind Krishnamurthy is supported by NSF grants CCR-9985304, ANI-0207399, and CCR-0209122.

Randy Wang is supported by NSF grants CCR-9984790 and CCR-0313089.

ing neighbors and link quality is available or obtainable, and we seek a distributed solution that makes efficient use of this rich quantity of information.

II. Background and Related Work

Over the last several years, extensive work has been done in the realm of overlay network design. In general, these systems can be broadly classified into two categories: tree-first strategies and mesh-first strategies. Here we summarize a few systems that are most relevant to our study.

A. Tree Construction Strategies

A number of projects have studied strategies for building application-specific overlay trees, particularly for multicast applications. The trees are constructed directly, and they define, implicitly, an unique communication path between every pair of nodes in the overlay network. These approaches have the advantage of not requiring the execution of a distributed routing algorithm.

The ALMI (Application Level Multicast) project [13] leverages a technique to compute a minimum spanning tree (MST) at a centralized administration point based on information retrieved from inter-node probes, which is subsequently programmed into the network.

Yoid, HMTP, and Overcast use the technique of constructing a distribution tree by having group members explicitly choose parents from their known set of neighbors [14] [15] [6]. Yoid's tree is static, but HMTP and Overcast trees adapt to changes in link characteristics and peer addition/removal by periodically trying to swap local links in the overlay. A good discussion of switch-tree improvement protocols may be found in [16].

TMesh [17] improves upon single shared tree overlays by randomly inserting "shortcut" links into the topology. Effectively, this reduces the diameter of the network and better supports multicast groups in which only some of the nodes are participants.

OMNI [9] considers a two-tier multicasting infrastructure consisting of service nodes and clients that connect to service nodes. The system provides algorithms for organizing the service nodes into an appropriate overlay structure based on the changing distributions of client connectivity. An initial centralized computation is followed by a distributed iterative refinement to obtain a tree that minimizes the minimum average latency and controls node degree.

NICE [18] partitions nodes into clusters of fixed size, elects representative nodes to represent the cluster, and repeats the process recursively in order to create a hierarchy of nodes. Tree maintenance for n nodes costs only $O(\log n)$ resulting in a highly scalable system.

B. Mesh Construction Strategies

Mesh-first strategies construct a richly connected graph first and then compute overlay paths or source-specific multicast trees using well known distributed routing algorithms. We now review existing methods of mesh-first overlay construction, summarizing the positive and negative properties of each and highlighting previous approaches and results where relevant.

a) Complete Graph: It is possible to perform overlay routing calculations based upon information that describes all available network links. The RON project at MIT is evaluating the benefits of overlay routing atop such a graph. Their results indicate that a connected overlay provides improved fault tolerance and performance to that observed in the underlying infrastructure [2]. Though the routes provided by the overlay are very near to optimal, from the standpoint of minimizing each of several metrics, the actual execution of a routing algorithm across a dense network, with $O(n^2)$ edges, is rather expensive: inter-server traffic becomes unacceptably large with networks even as small as 50 to 100 nodes [2].

b) k Random Links Graph: The *k-Random* links strategy is simple and fully distributed: every node randomly selects k of its edges for inclusion in the graph, resulting in a global edge count that is less than or equal to $k \cdot n$. This approach allows for high partition resilience with little overhead, for if a node detects a link failure, it may select another random link from its pool without coordinating with another peer. If a failed node returns to the network, it is similarly simple for it to rejoin. A *k-Random* graph also has probabilistically uniform degree distribution, which tends to promote load balancing for multicast networks, as well as shallow diameter. The main problem with this strategy is that the selected links may be bad, leading to routing inefficiencies. The approach of choosing k random peers is seen in numerous architectures. Narada, for example, chooses links at random initially, and then improves the quality of the mesh incrementally [5].

c) k Best Links Graph: This strategy is similar to the k-Random graph, except that each node autonomously chooses its k best links. Because pairs of nodes will independently include each other in their selections, the *k-Best-Links* graph is composed of as few as $(k \cdot n)/2$ and at most $k \cdot n$ low weight links. While this graph has low aggregate weight, it generally does not guarantee connectivity, and it might exhibit large diameters even if it is connected. We have found this to be especially true in real world scenarios like PlanetLab.

d) Short-Long Strategy: To select k links, a node may choose $k/2$ of the best (short) links of which it is aware and $k/2$ random (long) links. This strategy provides a low-weight topology with probabilistic connectivity properties. This is closely related to the k-MST strategy, which also includes many of the best links of the graph.

e) Connect-Improve Strategies: Several architectures take the approach of quickly choosing links, often randomly, to provide fast joining and subsequently rely on local improvement to improve the "goodness" of the mesh. In the Narada protocol, for example, a node joins the network by choosing random peers [5]. As the node discovers more peers, it evolves its randomly selected edges to include links of higher utility, while preventing partitions through a special mechanism. However, the initial graph is far below optimal and every node must continuously run the improvement algorithm, for evolution to proceed properly, causing relatively slow convergence and high bandwidth utilization [5], [18].

f) DHT Mesh: Structured overlays like distributed hash tables (DHTs) [19]–[22] view the overlay as a distributed data structure that dictates both the network topology and message routing. This integrated view has been shown to be massively scalable, requiring $O(\log n)$ neighbor information and guaranteeing $O(\log n)$ diameter for arbitrarily sized networks. Though originally developed for scalable object location, these overlays are now being used for applications traditionally supported on unstructured overlays [7], [8], [23]. The address structure of systems like CAN, Pastry, and Tapestry provides massive scalability and failure redundancy without the need for a traditional routing algorithm. These systems are designed with a scalability-first philosophy that focuses primarily on routing based upon an arbitrary addressing scheme and secondarily on routing to maximize link performance. Consequently, these systems tend to exhibit higher RDP, the overlay routing latency compared to unicast routes provided by the infrastructure. The locality properties of these systems are currently being investigated and improved upon [24]–[26] but it is as yet unclear as to whether structured overlays can support the minimization of arbitrary application-specific metrics and provide the desired levels of path diversity.

III. kTREE OVERLAYS

In this section we describe major aspects of our architecture, first giving an overview of the system, and then briefly decomposing important aspects of the system.

A. Overview

The goals of our architecture are several:

High Performance: The mesh should retain many of the best links (*i.e.*, links with low latency, high bandwidth, and low loss rate) from the original dense overlay network. In addition, the mesh should exhibit bounded degree and diameter.

Multiple Paths: A configurable number of multiple paths should be guaranteed by the network to improve real-time performance in the event of failure, to tolerate fluctuations in network performance, to enable application-specific path-selection, and to allow for concurrent use of network resources.

Exploit Network Information: Whatever information is available at startup and throughout the lifespan of the overlay should be fully utilized to maximize the efficiency of the network as quickly as possible.

Self Organizing: We seek a fully distributed solution that allows for incremental peer addition, unexpected peer failure, and other topological uncertainties.

We take a novel mesh-first approach of computing a subgraph of the known links that is composed of k spanning trees. Because a k-tree network is composed in this manner, it contains $O(kn)$ edges and is able to provide k overlay paths between a pair of nodes. Though it is possible to compose a network of k spanning trees of arbitrary type, we focus primarily on the method of using k minimum spanning trees after associating a cost metric to the communication links between pairs of nodes. The k-MST strategy bears striking resemblance to the best known approximation algorithm for finding a k-connected minimum-weight subgraph, which is a well known NP-Hard problem for $k \geq 2$ [27]. The resulting mesh, therefore, partially satisfies some of our design goals by providing multi-connectivity using many of the best links in the overlay network. We also take advantage of the extant foundation of research that has been devoted to the distributed computation of minimum spanning trees beginning with Gallagher, Humblet, and Spira in [28]. We build on their work by developing an algorithm that computes k minimum spanning trees in a concurrent and distributed fashion.

Some have criticized the use of Minimum Spanning Tree algorithms in Internet contexts for several valid reasons [16]:

Criticism - MSTs require a large amount of information to compute: In order to find the MST of a graph with m edges, $O(m)$ information is required. For a complete graph, this value is $O(n^2)$. However, it is possible to reduce the amount of information considered by the algorithm by "binning" [29] or by only feeding a small percentage of random links to the algorithm. Though this does not compute a true MST, our results indicate that the quality of the tree does not suffer much with incomplete information.

Criticism - MSTs must be recomputed in the event of node addition: In order to address this problem, we have developed a multi-hop tree improvement protocol that can restore a sub-optimal MST to an MST.

Criticism - MSTs do not bound degree or diameter, and therefore are subject to "hotspots" and long latency: This is true, but it is possible to add greedy degree bound and diameter heuristics to the protocol to provide these constraints.

Criticism - MSTs take a long time to compute and have high message complexity: The fastest algorithm for the distributed MST algorithm produces a solution in $O((D+d)\log n)$ time, where D is the maximum node degree of the input graph and d is the diameter of the constructed tree [30]. When the diameter heuristic is enforced and when the initial MST is computed using $O(\log n)$ connectivity information for each node, the resulting spanning tree can be computed in $O(\log^2 n)$ time. The message complexity of the algorithm is $O(n \log n + m)$, where m is the number of edges in the input graph. By limiting D, the message complexity could be lowered to $O(n \log n)$. $O(\log^2 n)$ convergence and $O(n \log n)$ message complexity for sparse graphs makes the system viable for a wide range of applications. We also observe that $O(n)$ convergence and $O(n^2)$ messages on complete graphs may be considered expensive in many circumstances; however, for high performance infrastructure applications, where the mesh must be very good before it will perform acceptably, the k-MST approach is typically much faster and consumes fewer messages than random-improve strategies, which take considerable time and bandwidth to converge to a similar "goodness" metric. This will be discussed more later.

Criticism - MST protocols are too complex: In comparison to many network protocols, MST algorithms are indeed complex. However, they can be implemented correctly; our implementation performs well in real-world scenarios like PlanetLab.

B. k-MST Background

k-MSTs are computed in the following greedy fashion. Let $MST(G)$ represent the edges of the minimum weight spanning tree of graph G; if graph G is not connected, then $MST(G)$ would correspond to a forest of disconnected components. Furthermore, let $G - MST(G)$ refer to a subgraph of G obtained after removing the edges in $MST(G)$. We compute:

$G_0 = G$
$F^0 = null$
for $j := (1 \dots k)$ **do**
  $F_j = MST(G_{j-1})$
  $F^j = F^{j-1} \cup F_j$
  $G_j = G_{j-1} - F_j$
end for

More simply, the j^{th} MST of the composite graph is the minimum spanning tree of the initial graph excluding the edges of previously computed MSTs. The algorithm then outputs a subgraph $F^k = F_1 \cup F_2 \dots F_k$ that is comprised of the k minimum spanning trees.

The motivation for pruning the original graph to a k-MST is three-fold. First, the k-MST provides an approximation to the best known theoretical algorithms for graph pruning. Second, the k-MST approach extends previous efforts that employ centrally computed minimum spanning trees for streaming applications. Third, a k-MST sub-graph has a number of desirable properties that facilitate the task of mesh maintenance in a distributed setting. We now examine each one of these considerations.

1) Related Graph Theoretic Results: Techniques for using interleaved spanning trees to compute constrained minimum-weight connected subgraphs have been well explored in previous literature. The algorithm with the lowest known approximation factor for the k-connected minimum weight subgraph problem is by Khuller and Vishkin [27] and uses a graph algorithm developed by Gabow [31]. The approximation algorithm works as follows. It takes the undirected graph and forms a directed graph G where every undirected edge is replaced by two anti-parallel directed edges of the same weight. A polynomial-time algorithm, discovered by Gabow, is then used to find k edge-disjoint directed trees with the smallest cumulative weight. If at least one of the directed edges is picked by Gabow's algorithm, the corresponding undirected edge is included in the pruned subgraph. It can be shown that the pruned subgraph is k-connected and is at least a 2-approximation to the minimum-weight k-connected subgraph of G. The resulting subgraph has between $k \cdot n/2$ to $k(n-1)$ edges.

A related approach is the algorithm by Roskind and Tarjan [32], which can be used to find k disjoint spanning trees of minimum total weight given an undirected graph. These spanning trees accumulate to generate a k-connected subgraph. This approach differs from the previous one in that it performs this computation on an undirected graph and yields $k(n-1)$ edges. Consequently, it might result in a subgraph with more edges than the Khuller-Vishkin strategy.

Both algorithms exhibit limited concurrency, do not lend themselves to distributed computation, and do not factor in diameter and degree. However, given their similarity to k-MST, they become an interesting point of reference. In fact, we have empirically observed that the k-MST algorithm, when executed without degree and diameter constraints, computes solutions that closely approximates the solutions computed by the Roskind-Tarjan algorithm for the various network topologies that we have studied.

2) Related Overlay Construction Strategies: Other closely related work includes ALMI, in which the authors demonstrate that a single MST, computed in a centralized fashion and then disseminated to overlay peers, provides for a good multicast tree. Also relevant is the work done on TMesh, which augments a multicast tree with random "shortcut" links to improve latency performance by reducing network diameter [17]. However, instead of augmenting the initial structure with only sparse links, we augment with complete trees that provide "shortcuts" while also increasing global redundancy.

3) k-MST Properties: We briefly note a few properties of the subgraphs computed by the k-MST algorithm when executed without the degree and diameter constraints. While these properties hold only in an approximate manner if the degree and diameter constraints are imposed, they still provide insight into a number of interesting properties that are useful for maintaining the mesh. All of these properties can be proved with simple graph-theoretic refutation arguments, which are omitted for brevity.

Property 1: k-MST includes the k best links of every node in the graph.

Property 2: Let e be an edge belonging to the tree F_j. Let e connect two sets of nodes S_1 and S_2 in F_j. If the cost of e rises dramatically (due to a fault or fluctuating network conditions), then F_j can be repaired by adding the minimum weight edge connecting S_1 and S_2 in F_{j+1}. One does not need to consider any other edges to repair F_j.

Property 3: Consider an edge e that is in F_j. Let the cost associated with e be lowered, potentially when new information is revealed about the overlay network. Let e' be the edge with the maximum cost in the cycle created in $F_j \cup \{e\}$. If *weight(e)* $\leq$ *weight(e')*, then the minimum-weight property of F_j could be restored by swapping in e for e'. One does not need to make any other changes to F_j.

C. Distributed Minimum Spanning Tree Algorithm

The distributed computation of a minimum spanning tree was solved by Gallagher, Humblet, and Spira in their groundbreaking paper [28]. We now provide a brief review of their algorithm, referred to as GHS in this paper. Please refer to [28] or [34] for a thorough discussion of GHS.

We assume that neighbor information comes from some out-of-band mechanism, and that all probing has been done by the start of the algorithm.

The algorithm builds the components in *levels*. At the start of the algorithm, each individual node comprises a component of size one and is at level 0. At each step, one of the nodes in each component is elected as a *leader* of the component. The *leader's* UID and the component's *level* number is used as the component identity, *compid*. The *leader* broadcasts an *initiate* message along the spanning tree edges of the component to start a search to find the *minimum weight outgoing edge*, or *mwoe*. On receipt of the *initiate* message, each node probes its remaining links in order of increasing weight to identify the lowest-cost edge that leads to a different component. A node performs a probe by sending a *test* message containing the node's *compid* to its best neighbor. The neighbor sends back an *accept* message only if it belongs to a different component. If it belongs to the same component, it responds with a *reject* message. The results of the search are convergecast back to the *leader* through *report* messages. The *leader* then identifies the *mwoe* of the entire component and sends a *changeroot* message to the node that is adjacent to the *mwoe*. On receipt of the *changeroot* message, a node sends a *connect* message across its *mwoe*. When *connect* messages have been sent both ways along this edge between two components of the same *level*, a *merge* operation occurs to create a new component with *level = level + 1*, and one of the two endpoints of the *mwoe* is elected as the *leader* of this new component. If a *connect* message is sent to a component that is at a higher level than the sender, an *absorb* operation occurs, and the lower level component gets incorporated into the higher level component. When the algorithm terminates, there is just one component that includes all the nodes in the graph connected by the minimum spanning tree.

GHS is message efficient in that it uses $O(n \log n + m)$ messages, which is optimal. However, its time complexity, $O(n \log n)$, is not optimal. Subsequently, several others have developed faster versions of the protocol [30], [33].

D. Distributed k-MST

The distributed k-MST algorithm may be considered as k instances of the distributed MST algorithm, labeled $MST_1, MST_2, \ldots, MST_k$, which generate k forests $F_1, F_2, \ldots, F_k$. Each process sorts its $n-1$ links and feeds them in non-decreasing order by weight to MST_1. If an edge e can be added to F_1 without creating a cycle and if it is identified as the *mwoe* by a component in F_1, then e is added to F_1. Otherwise, MST_1 hands e off to MST_2, which performs the same operation, and the edge is similarly promoted to subsequent instances of MST until it can be successfully added to a tree. If MST_k is unable to add the edge to F_k, the edge is tagged in a manner similar to the original MST algorithm so as to exclude it from further computations.

The k-MST algorithm computes the k trees concurrently, and therefore, at any given time, a single process has membership in k trees. For this reason, additional state with respect to each MST must be kept internally and additional information must be appended to messages. First, the component identifier requires an additional field *treeid* denoting the MST with which the subtree is associated. Second, each process p maintains $treeid_p[q]$ to store the tree to which the link (p, q) belongs or is being considered for. In GHS, every link belongs to one of three categories: *branch* denoting a spanning tree edge, *rejected* denoting a non-outgoing edge, or *basic* denoting the initial state. In the k-MST algorithm, $reject$ messages cause link $treeid$s to be incremented, thereby allowing the links to be processed by higher instances of MST. When $treeid_p[q] = k+1$, the link has failed to join all trees $j \in (1, \ldots, k)$ and may therefore be considered as rejected.

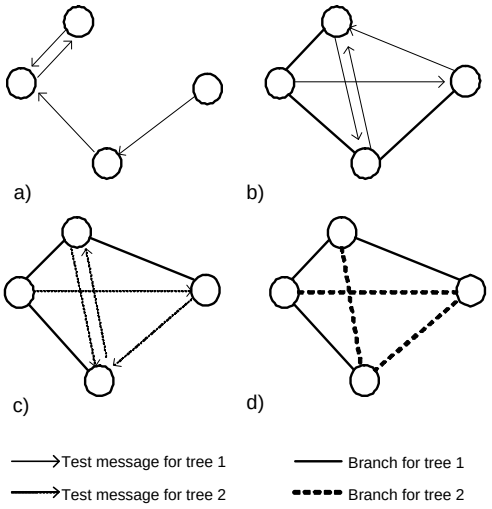

Fig. 1. The minimum spanning trees computed by 2-MST for a 4 node network. In (a), peers test their closest edges. Since no peer yet has any branches, all these edges are included in the first MST. In (b), peers test the next closest neighbors. Because these are interior edges, they are rejected and promoted to the second MST. In (c), the second instance of the MST algorithm tests the newly injected edges, and because the second tree has no links yet, all these edges are accepted. (d) shows the completed 2-MST.

At the start of the algorithm, every process in the network wakes up and begins searching for peers to join to. At $t = 0$ in GHS, there are n subtrees of size one; in k-MST, there are kn subtrees, each node constituting k trees. Each process p stores a vector $compid_p[1, \ldots, k]$ to specify the k different components to which it belongs.

Let p be a particular process in the execution. For each $j \in (1 \ldots k)$, the process searches through its links to find the lowest weight *basic* edge, *best*, that also has the property $treeid_p[q] = j$. This edge is the minimum weight outgoing edge under consideration by the process p for inclusion in forest F_j. Let q be the remote node associated with *best*. p sends a $test$ message to q containing $compid_p[j] = (leader_p[j], level_p[j], j)$. Node q processes the message as follows:

if $(treeid_p[q] > treeid_q[p])$ **then** *wait*
if $(treeid_p[q] < treeid_q[p])$ **then** *reject*
if $(treeid_p[q] == treeid_q[p])$ **then** *DoMSTCheck*

where *DoMSTCheck* checks whether the two subtrees belong to the same component.

When process p receives a *Reject* message from q, the link (p, q) is not a part of F_j. It must then increment the $treeid$

associated with that link so that the algorithm can attempt to join it to the next forest. When $j = k + 1$, the algorithm has failed to join the link to any tree, and may be considered *rejected*.

As links are rejected from early trees, they become available for evaluation by higher instances of the algorithm. Because edges are always selected in order of minimum weight, each successive instantiation of MST receives as input the edges with the lowest possible weights. When all links in the graph have state either $branch$ or $rejected$, *i.e.*, $treeid_p[q] = k+1$, the algorithm is complete and outputs the graph consisting of k forests.

E. Routing Atop k-Trees

We utilize a link state protocol to compute robust, dynamically adaptive routes within our constrained mesh. Link state protocol enables quick dissemination of network conditions and allows for flexible, application-specific, path selection. The protocol we use is based upon Radia Perlman's *New, Improved Link State Distribution Protocol* [35]. Link state routing also facilitates our fault recovery and improvement mechanisms, which we describe in the following sections.

F. Peer Addition

The GHS algorithm can be easily extended to allow for incremental node addition. Because each node begins at level 0 upon startup for each tree that is being constructed, the first node it tests will always absorb it. This does not guarantee optimal placement, however, and the improvement phase of the protocol must be used to reposition the node into the global MST. Most MST algorithms offer a similar facility.

G. Fault Recovery

Consider a single GHS tree. If a node or link failure occurs, each adjacent node independently detects this failure and performs an action according to whether or not the failed link is its parent, meaning a link that points towards the *leader*, or a child. If the link pointing to the tree's *leader* fails, the node p that detects the failure designates itself as the new *leader* of the component consisting of itself and its children. If the failed link points to a child, p sends a *failure* message across the overlay to the *leader* of its component. The leaders of the two components then change their component identifiers and broadcast a message along each tree to tell the nodes to update their component identifier and roll back *rejected* edges to *basic*. When the leaders receive a convergecast acknowledgement from each of its children, they broadcast an *initiate* message to begin a new search for the *mwoe*.

In the case of multiple concurrent faults at different parts of the tree, or failures taking place during a *merge* or *absorb*, several *initiate* messages may traverse a component concurrently, each tagged with a different component identifier. If a node receives a different *initiate* with the same *level* as itself, it must come from upstream. Such messages are accepted only if they are more recent than other *initiate* messages that it receives, the ordering of which is determined by a simple sequence counter. *initiate* messages of higher level result only from *merge* and *absorb* operations, and are accepted without exception.

The protocol described above rolls GHS back to an expensive phase, where nodes must test all interior edges of their component. This drawback could be addressed in one of three ways. First, the message count complexity may be avoided by disseminating internal node member knowledge, which we describe later, so that interior edges are not probed. Second, the system could roll back only some of the *rejected* edges to *basic*, thereby allowing the partition to be repaired using a possibly sub-optimal link. The issue of optimizing the sub-optimal tree could then be addressed by the improvement phase of the protocol. Third, one could exploit Property 2 described in Section III-B.3, which states that the lowest weight link e that could repair the partition in F_j is in F_{j+1}. In the event of a single link failure in j, the process p that discovers the fault could identify the edge e from F_{j+1} that connects the two partitions of F_j and notify the endpoints of e to shift the edge from F_{j+1} to F_j. This procedure would continue until either no link can be added or until the failure shifts into the last forest, at which point it executes the full-blown repair procedure described above.

H. Incremental Improvement

Our improvement protocol exploits the properties of trees to restore the global optimality after peer addition or changes in link characteristics. Central to this is the use of a link state protocol to propagate information about the current conditions of a spanning tree.

Assume that the overlay is currently in a sub-optimal state. Once in a while, each node checks to see if it is able to make an improvement using local information gathered from periodic probes and the link state that has been propagated throughout the network. Locally, a node picks a non-branch link e and adds it to this tree, thus creating a fundamental cycle. If any of the links along the cycle are heavier than e, then an improvement could be performed by swapping this link for e. We will call the decrease in weight associated with this change *delta*. The node then performs a similar computation using each of its other non-branch edges, and the improvement that is found with the largest *delta* is the improvement that will be made; if no such link exists, then the tree is already a MST and nothing is done.

Once the best improvement is determined locally, a peer may then execute the swap on the network. In a distributed setting, we must be careful to avoid possible partitions and loops that may form as a result of disagreement about the network's link state. Figure 2 shows a tree where two separate processes, A and B, wish to perform an improvement at the same time on different links, but there are shared links in the cycles created by the improvement process. Note that if the two processes modify the same link or links on the shared path, it is possible that a loop and/or partition may occur, thus obliterating the spanning tree.

The protocol to execute the improvement, without creating loops or partitions, works as follows. A process p first sends a

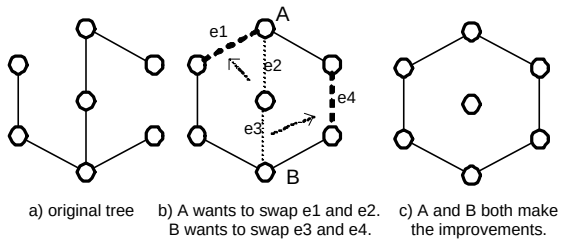

Fig. 2. Possible concurrency errors resulting in loops and/or partitions.

lock message along the improvement cycle to all of the edges that may be affected by the improvement. Next, it sends an *improve* message along the cycle to add the improvement link, remove the bad link, and readjust parent pointers appropriately. Finally it forwards an *unlock* message in the reverse direction to finalize the improvement. A full description of this protocol is beyond the scope of this paper, but we have devised simple ways to ensure that no concurrent improvements take place on shared links, and that the tree remains in a survivable state at all times; parent pointers are always such that the fault recovery mechanism works if any link or node along the improvement cycle goes down.

I. Degree and Diameter Bound Heuristics

On real world topologies, it is possible that an MST could exhibit hot spots and suffer from high diameter. However, it is possible to modify the distributed MST algorithms to add heuristics that approximately limit the degree and diameter of the MST.

To limit degree, for example, a node can reject $test$ messages from peers if its degree target has been reached. However, this is not sufficient by itself. Consider that there may be a "hub" node p that is close to a large number of peers. In such a case, at $t = 0$, many nodes will test p, and because p has no links at the start of the algorithm, it will accept all requests. To bound its degree, p will accept the first D $connect$ messages, and will then send a special $restart$ message to the remaining peers, notifying them that their respective components must restart the search round. All subsequent $test$ messages will then be rejected. We note that the greedy algorithm does not attempt to build the lowest-weight spanning tree for a given degree bound, which is an NP-Hard problem.

A similar methodology may be used to control network diameter. Let $size(C)$ denote the number of nodes in component C, and let $dia(x)$ denote the distance from a node $x \in C$ to the node farthest from x within the same component. A component can compute the $size$ and dia values for each node inside the component through a pair of carefully designed broadcast-convergecast operations. The diameter constraint is then enforced by requiring that a *merge* of two components C_1 and C_2 along an edge (u, v) can happen only if $dia(u) + dia(v) < c \cdot \log(size(C_1) + size(C_2))$, for some fixed constant c. Again, this is a greedy approach, which does not necessarily compute the lowest weight spanning tree for a given diameter bound.

J. Reducing kMST Complexity

All techniques for improving the performance of distributed MST algorithms may be applied to the k-MST algorithm. Such techniques may be found in [36] [30].

An additional method that can be used to reduce the message count and time complexity is to simply let every node know the members of its component. This may be done in the Report and Initiate phases of the protocol by having all nodes report a list of their children to the root, who subsequently broadcasts this list when initiating the next search phase. This eliminates intra-component testing, which always leads to rejection, thus reducing the message count to $O(n \log n)$ at the expense of increasing the size of report and initiate messages from $O(1)$ to $O(n)$.

A similar technique can be used by employing Bloom filters instead of complete lists, thus providing nodes an approximate view of component membership; some configurable percentage of the time, a node will conclude that a remote peer is already a member of its component, even when it is not, and mistakenly reject it in a pre-emptive fashion. However, with high probability, the final tree will still be connected and close to the optimal result.

K. Bandwidth and Loss Rate Metrics

The designers of RON pointed out the benefits of being able to route according to multiple metrics, such as latency, bandwidth, and loss rate. Because pruning a graph based on only one metric could limit a routing algorithm's ability to optimize other important factors, we take the approach of computing multiple overlay networks, each composed of trees that either minimize latency or loss rate or maximize bandwidth. This allows several applications to execute concurrently, each routing atop a graph optimized for an appropriate metric (as illustrated in Figure 3). For bandwidth-intensive applications, it would be ideal if the mesh provides paths that have few bottlenecks at the physical link layer. We are currently extending our algorithms to incorporate physical topology information, if such information is available.

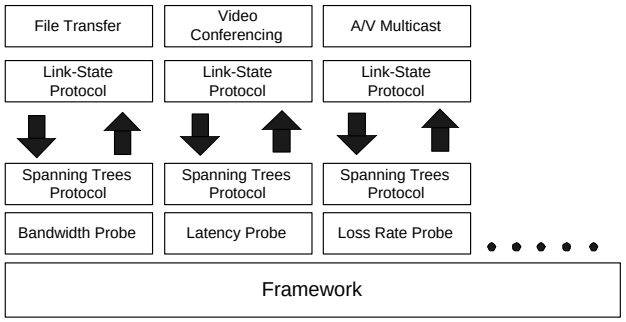

Fig. 3. Interaction Between k-MST Maintenance and Routing

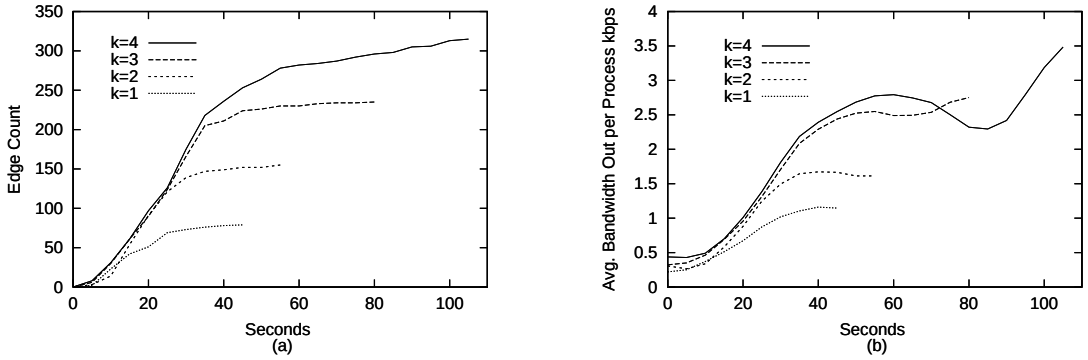

Fig. 4. (a) Time to compute number of links for various values of k, $n = 80$. (b) Bandwidth consumed by k-MST construction for various values of k. Data values are plotted until the k-MST construction is completed.

IV. Evaluation

In this section, we evaluate various aspects of k-MST meshes in simulation and also study the PlanetLab deployment of a k-MST overlay network.

A. k-MST Complexity Measurements from PlanetLab

To evaluate the performance of k-MST, we use an 80 node deployment of the algorithm across PlanetLab. The algorithm is implemented as described in Section III-D. Protocol messages are queued at nodes and handled in a batched mode once every second. Traffic measurements include the message overhead for constructing the mesh as well as heartbeat and other maintenance traffic.

Figure 4(a) shows global link count as a function of time for several values of k. Near time $t = 0$, there is very little parallelism in the construction of the different trees. This is because before tree $j+1$ may test any edges, it must first wait for tree j to reject some. Once there are some links in the pipeline, higher tree instances begin to successfully execute, creating branches and rejecting links. Finally, the tail end of the curve again demonstrates limited concurrency, as the last trees are waiting for lower instances to process their links, most of which will be interior.

Figure 4(b) illustrates the bandwidth consumption of the algorithm. The bandwidth utilization of MST is at its highest when components are probing for outgoing edges, particularly the later stages where interior edges become increasingly numerous. At the start of the run, when all sub-trees are composed of a small number of nodes, a larger proportion of *test* messages are accepted by peers. As components form into larger trees, more of a process's *test* messages result in rejection, since its better links correspond to nodes that are already part of its component. Towards the end of the run, when components span much of the network, the rejection rate is at its highest, particularly when the network demonstrates large localized pockets.

In Figure 4(b), the "bumpy" pattern corresponds to points in time when large local subtrees have formed. In the first steep portion of each curve, bandwidth is increasing as trees in the Western US, Eastern US, and Europe form. Once these regions finish testing their relatively local links, they begin merging with one another, decreasing bandwidth utilization temporarily and causing the downturn in the first "hill." Bandwidth consumption again increases once these inter-locality links have formed, and is highest during the last phases of the algorithm as nodes test peers to which they are already connected. We will continue our analysis of k-MST's bandwidth utilization in the following section with simulations of larger executions.

B. k-MST Complexity Measurements from Simulation

In order to study more general trends of k-MST in networks of various sizes, we executed a k-MST simulation across two topologies. The first was created by using Internet router information obtained by the Internet Mapping Project at Lucent Bell Laboratories circa November 1999. From this map, 1000 edge-nodes were chosen and a pair-wise distance, measured in terms of hop count, was used as the cost metric to compute the subgraph, since actual latency data was unavailable (as discussed in [37], [38]). The other topology was a Transit-Stub network produced by Georgia Tech's GT-ITM topology generator; we generated a 50,000 node network and then chose 1000 nodes at random to perform our experiments. For both topologies, when fewer than 1000 nodes were needed, the required set of nodes was chosen randomly from these 1000 node pools.

Although in practice input meshes for k-MST will likely be sparse, in these experiments we use fully connected graphs, which give worst-case measures for convergence and communication. Recall that the message complexity of an optimal implementation of MST is $O(n \log n + m)$, and with a fully connected graph, the m term is n^2. For this reason, we are interested primarily in the time and the number of messages required before all trees are formed, when the mesh is connected and usable, not the complexity required for the algorithm to exhaust all internal edges in the last phase. The n^2 bandwidth that is used by the algorithm can be dealt with by reducing the number of edges considered by the algorithm.

In our simulation, one "count" indicates the time required to perform a send-receive operation or internal action. For example, at time $t = 0$, process p may wake up and send a message to q, which q will receive at time $t = 1$. Broadcasts and convergecasts, therefore, each take $O(\textit{diameter})$ time.

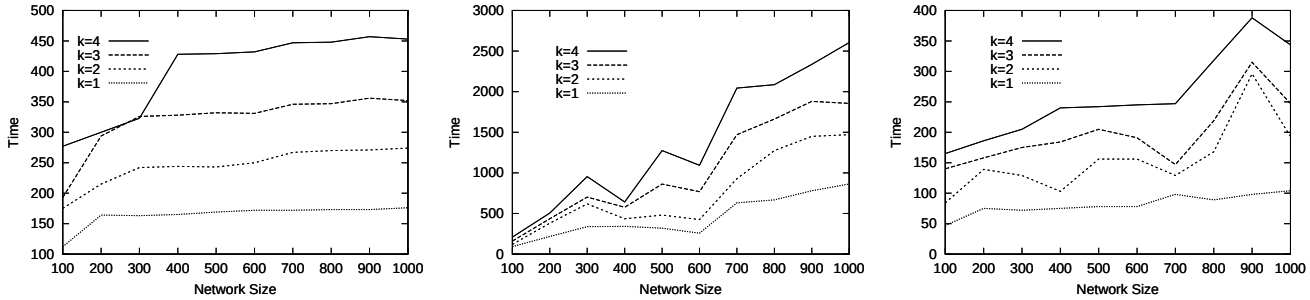

Fig. 5. Time to compute k-MSTs on (a) router topology, (b) transit stub topology, (c) transit stub topology with group member knowledge.

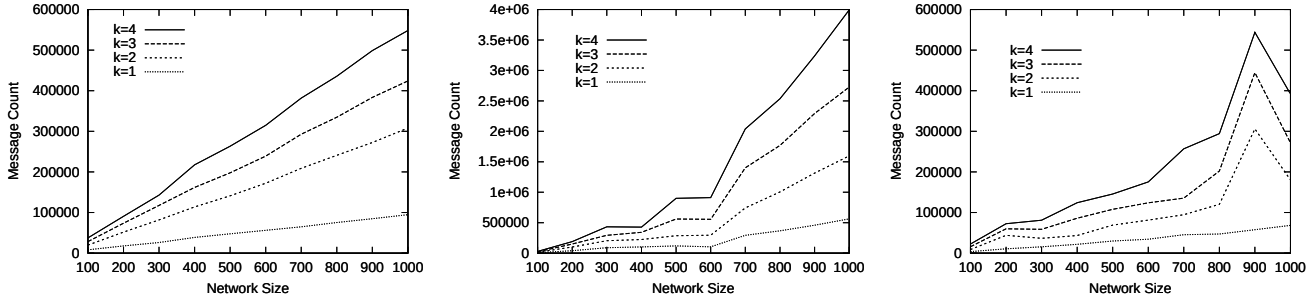

Fig. 6. Messages required to compute k-MSTs on (a) router topology, (b) transit stub topology, (c) transit stub topology with group member knowledge.

We first executed k-MST on the Internet Mapping Project router topology with network sizes up to 1000 nodes and values of k varying from 1 to 4. On this topology, the protocol performed surprisingly well, demonstrating fast tree building on networks of all sizes and showing constant changes in time for various ks. Also, there was only a linear increase in bandwidth as the number of nodes in the network increased, which is the best possible case; this indicates that very few interior nodes were tested at each level of the algorithm.

On the Transit-Stub topology, however, we observed worse behavior. As the number of nodes increased, the time required to compute k trees increased much more steeply than in the previous topology; the time required to compute k trees is roughly five times that required to perform a similar execution on the Internet Mapping Project topology. Furthermore, the number of messages required to perform the distributed computation increased quadratically with the node count. This is because the Transit-Stub graph exhibits more localization than the previous topology, causing an increase in intra-component testing and pushing the complexity of creating the mesh close to the worst case of $O(n^2)$.

In order to reduce the worst case complexity of MST algorithms, one could reduce edge count (by choosing a subset of links) or disseminate group membership information. The latter approach automatically forces the message size complexity of the algorithm to $O(n^2)$ for even sparse graphs, but reduces the message count complexity to $O(n \log n)$. This is because if peers know which edges are already interior, they need not test them. Despite the seemingly expensive nature of this strategy, in a practical setting it adds only minimal overhead. Because the network is a tree, when a peer joins to another peer through a *merge* or an *absorb* operation, the affected edges can exchange membership information and then broadcast new member updates to the rest of the sub-component, thereby requiring membership information to traverse a link only once. Consider that for a 1024 node network, where each peer has a 4 byte address, a total of 4KB of information must be sent to each node during a run of MST, amortized over the duration of the execution.

We used simulation to test the effectiveness of group membership dissemination for the poorly performing Transit-Stub topology. Because there is no need to test interior edges using this method, the algorithm is extremely fast, requiring only 100 steps for a 1000 node tree, versus over 800 steps without the optimization. Also note that the message count produced with enhanced group knowledge is along the same order of magnitude as the message count from the Internet Mapping Project graph.

C. k-MST Performance Results from Simulation

We next evaluated the quality of the k-MST mesh using simulation. First, we compared the average cost of redundant, edge-disjoint, shortest paths for the meshes generated by several different mesh construction strategies. We also evaluated the quality of the computed paths when there is incomplete knowledge regarding edge costs, a situation that might arise when one wants to minimize the probing costs as well as lower the time required to compute k trees. In such situations, k-MST simply computes the MSTs of the sparse graph.

Figure 7 plots the average shortest-path hop count of four edge-disjoint shortest paths for three pruning strategies while varying the number of mesh edges. The experiments were performed on the router topology from the Internet Mapping project. The three plots correspond to hop counts when there is

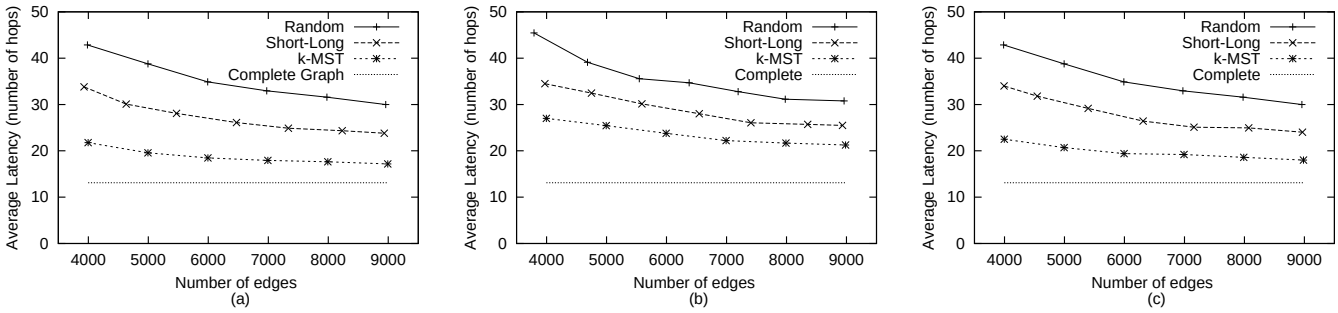

Fig. 7. Average hop count of four edge-disjoint shortest paths under different levels of information for router-level topology from the Internet Mapping Project. (a) All the edge weights in the original graph are known. (b) Each node knows the edge weights to forty other nodes in the system. (c) Each node knows the edge weights to forty other nodes, including five of its closest neighbors.

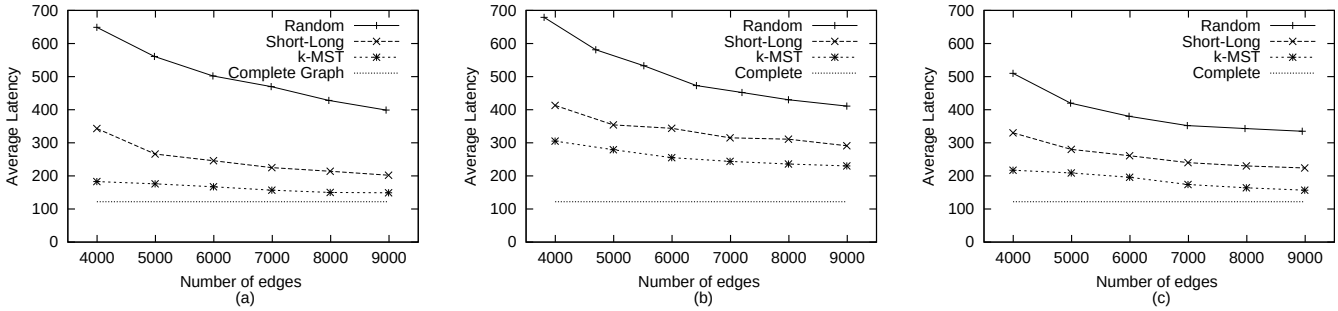

Fig. 8. Average latency of four edge-disjoint shortest paths under different levels of information for topology from GT-ITM. (a) All the edge weights in the original graph are known. (b) Each node knows the edge weights to forty other nodes in the system. (c) Each node knows the edge weights to forty other nodes, including five of its closest neighbors.

complete edge information, when each node knows of only 40 edges, and lastly when each node knows of 40 edges of which 5 are closest neighbors. In all cases, k-MST demonstrates a lower hop count, with other strategies performing as expected.

In Figure 8 we run a similar test across the GT-ITM topology. On this topology, k-MST performs well, while the random strategy performs very poorly. As we had found in our other simulation experiments, the Transit Stub graph demonstrates a large amount of localization, where groups of nodes are in relatively dense areas that are quite distant from similar groups. Choosing random links in this scenario ignores this locality property; though picking random neighbors upon startup is fast, it may not provide adequate performance for demanding applications.

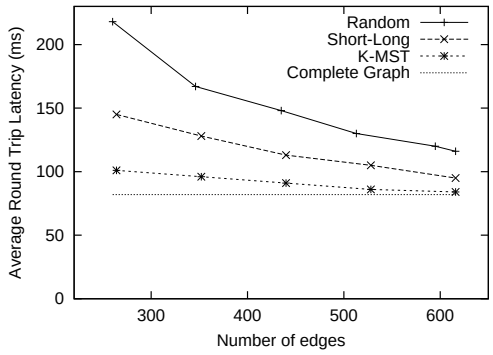

Fig. 9. Round trip latency of three edge-disjoint shortest paths for varying edge counts.

D. k-MST Performance Results from PlanetLab

1) Latency Measurements: We extended our testing to a practical environment using an 89 node PlanetLab network. In Figure 9, the round trip latency of three edge-disjoint shortest paths were computed for meshes with various edge counts. The results we obtained from this were similar to that of the GT-ITM topology, with k-MST performing much better than the other strategies (as shown in Figure 9).

2) Mesh Improvement: Choosing random links generally produces a poor quality mesh, which means that it must be improved considerably before it is of reasonably good quality. We illustrate this in Figure 10, where we run a Narada improvement [5] simulation atop 3-Random and 3-MST graphs and plot how the average latency of the shortest paths changes over time. For each mesh we apply incremental improvements by having each node periodically probe a random non-neighbor, and add the corresponding link if the resulting improvement to the shortest paths is above some threshold. For the 3-Random mesh, the improvement algorithm periodically drops low-utility links while preventing partitions using the heuristic mechanisms proposed by Narada. For the k-MST mesh, the improvement algorithm drops a low-utility link immediately after adding a high-utility link, with the link to be dropped chosen from the fundamental cycle created by the newly added link; the ability to detect this cycle is unique to k-MST.

At time $t = 0$, the k-MST mesh is already very close to

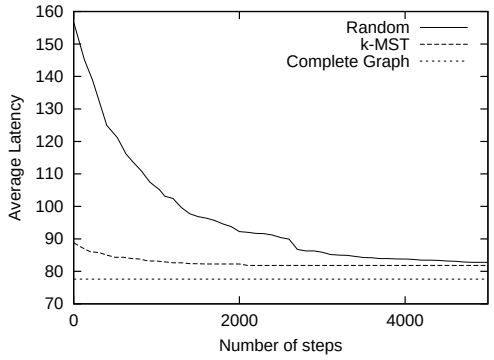

Fig. 10. Narada improvement strategy performed on Random and k-MST graphs.

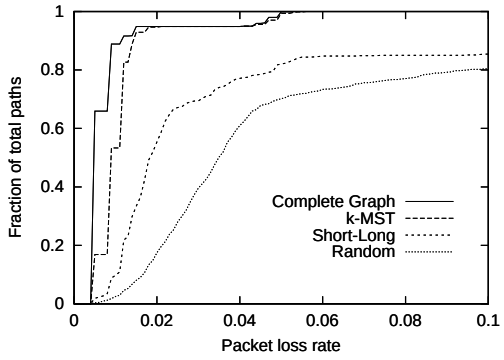

Fig. 11. Cumulative distribution function of the loss-rates on overlay paths.

optimal, though the random mesh must go through over 3000 improvement cycles before the mesh quality is similar to k-MST. Of course, in order to compute k-MST in the first place, the upfront expense of the MST algorithms must be incurred. But because Narada probes require transfer of routing tables with $O(n)$ values, such an improvement strategy can be much more expensive and time consuming than MST computation. The Narada improvement algorithm also does not ensure k-connectivity as the mesh is refined. Even though every pair of nodes has three edge-disjoint paths in the random mesh at $t = 0$, our experiments indicate that fewer than 50% of the redundant paths remain after 3000 improvement cycles. The k-MST mesh, on the other hand, is always composed of k trees, thereby ensuring the existence of k edge-disjoint overlay paths between any two nodes.

3) Loss-rate Measurements: We then evaluated the quality of the k-MST mesh for transmitting data streams over overlay paths. We used observed loss-rates from periodic PlanetLab probes as the link-weight metric in order to compute the k-MST. We then transferred a constant bit-rate data stream of 1 Mbps through k distinct overlay paths between every pair of nodes and monitored the packet loss. Figure 11 compares the different mesh construction strategies by calculating the cumulative distribution function of the loss-rates associated with the overlay paths. k-MST overlay paths exhibited significantly lower loss-rates than the other pruning schemes, with loss-rates that were lower by more than 5% for about 20% of the paths. A 5% improvement is considered substantial for streaming applications.

E. Application Performance

As a benchmark application, we implemented a multicast file mirroring utility to reliably send 100 megabytes of files to each overlay node along the edges of degree-constrained multicast trees that were computed atop meshes built using the bandwidth cost metric. In the transfer protocol, each child computes its shortest path, using link state information, to a file source and periodically requests an advertised file until it becomes available from its parent. When the file becomes available, the child transfers it using TCP and immediately writes each packet to disk so as to make it available to its own children. Though this methodology is not a real-time multicast transfer scheme, it still gives insight into the efficiency of the system by illustrating its ability to delegate transfer load along the network.

The average data rates (as shown in Figure 12) for all three strategies were surprisingly high, with the constrained k-MST topology and the complete graph producing trees that sustained over 2.5 megabits-per-second for the bulk of the transfer. The pipelined TCP transfer method we employed seems to allow for substantial transfer concurrency. For trees computed on both the k-MST and complete graph, 100 megabytes were transferred to each of 80 geographically dispersed nodes in less than 8 minutes, for a total transfer of 8 gigabytes. The tree computed atop the random graph finished most of its transfers within 12 minutes, though the remaining nodes took over 30 minutes to synchronize.

V. CONCLUSIONS

In our paper, we proposed using interleaved spanning trees to compose an overlay mesh. We focus specifically on a strategy of using k minimum spanning trees for mesh construction, and evaluate the methodology using simulation and a working implementation, which was run atop PlanetLab. Our results show that k-MST mesh demonstrates good characteristics, including low weight and good path diversity, despite the sparseness of the graph, for networks of less than 1000 nodes. On our PlanetLab implementation, our prototype demonstrated reasonable bandwidth utilization during construction, low loss-rates for data streams, and high performance in a realistic multicast file transfer scenario. Though there is an initial expenditure to construct a high quality network, seen both in terms of message count and time complexity, this cost is modest compared to the cost required for random-improve strategies to converge to a similar result. Though our preliminary results are promising, we are still investigating the behavior of k spanning tree networks in medium-scale, real-world applications.

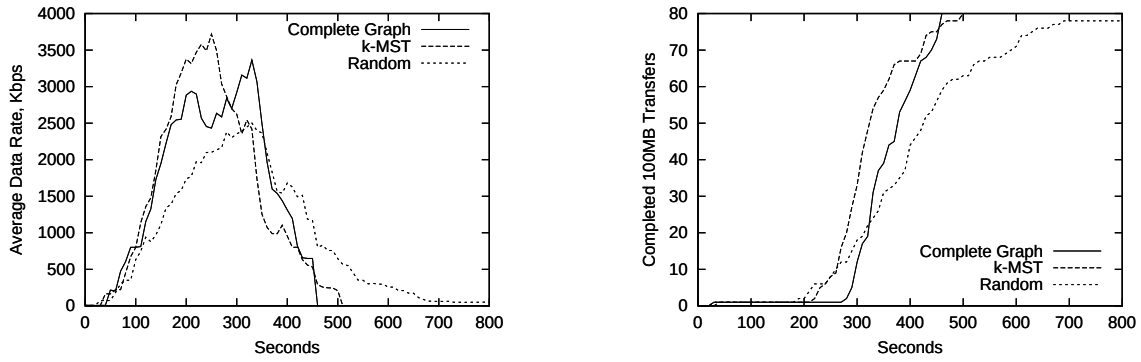

Fig. 12. Results from the file mirroring application. (a) Plot of average data transfer rate vs. Time. (b) Time required to complete data transfer to 80 nodes.

REFERENCES

[1] Stefan Savage, Andy Collins, Eric Hoffman, John Snell, and Thomas E. Anderson, "The end-to-end effects of internet path selection," in *SIGCOMM*, 1999, pp. 289–299.

[2] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris, "Resilient overlay networks," in *Proceedings of SOSP*, 2001, pp. 131–145.

[3] T. Nguyen and A. Zakhor, "Path diversity with forward error correction (pdf) system for packet switched networks," in *Proceedings of IEEE INFOCOM*, 2003.

[4] J. Apostolopoulos, T. Wong, W. Tan, and S. Wee, "On multiple description streaming with content delivery networks," in *Proceedings of IEEE INFOCOM*, 2002.

[5] Yang hua Chu, Sanjay G. Rao, and Hui Zhang, "A case for end system multicast," in *Proceedings of SIGMETRICS*, 2000.

[6] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr., "Overcast: Reliable multicasting with an overlay network," in *Proceedings of OSDI*, 2000, pp. 197–212.

[7] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz, "Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination," in *Proceedings of NOSSDAV*, 2001.

[8] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Application-level multicast using content-addressable networks," in *Proceedings of NGC*, 2001.

[9] Suman Banerjee, Christopher Kommareddy, Koushik Kar, Bobby Bhattacharjee, and Samir Khuller, "Construction of an efficient overlay multicast infrastructure for real-time applications," in *INFOCOM*, 2003.

[10] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai, "Distributing streaming media content using cooperative networking," in *Proceedings of NOSSDAV*, 2002.

[11] John Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost, "Informed content delivery across adaptive overlay networks," in *Proceedings of ACM SIGCOMM*, 2002.

[12] W. C. Cheng, C. Chou, L. Golubchik, S. Khuller, and Y. Wan, "Large-scale data collection: a coordinated approach," in *Proceedings of IEEE INFOCOM*, 2003.

[13] Dimitris Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel, "ALMI: An application level multicast infrastructure," in *Proceedings of the USITS*, 2001.

[14] Paul Francis, "Yoid: Extending the internet multicast architecture," 2000.

[15] Beichuan Zhang, Sugih Jamin, and Lixia Zhang, "Host multicast: A framework for delivering multicast to end users," in *INFOCOM*, 2002.

[16] David A. Helder and Sugih Jamin, "End-host multicast communication using switch-trees protocols," in *Global and Peer-to-Peer COmputing on Large Scale Distributed Systems*, 2002.

[17] Wnejie Wang, David Helder, Sugih Jamin, and Lixia Zhang, "Overlay optimizations for end-host multicast," in *Proceedings of Fourth International Workshop on Networked Group Communication*, 2002.

[18] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy, "Scalable application layer multicast," in *Proceedings of ACM SIGCOMM*, 2002.

[19] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of SIGCOMM*, 2001.

[20] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, "A scalable content addressable network," in *Proceedings of ACM SIGCOMM*, 2001.

[21] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*.

[22] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *International Conference on Distributed Systems Platforms*, 2002.

[23] 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*, 2002.

[24] M. Castro, P. Druschel, Y. Hu, and A. Rowstron, "Topology aware routing in structured peer-to-peer overlay networks," 2002.

[25] P. Keleher, B. Bhattacharjee, and B. Silaghi, "Are virtualized overlay networks too much of a good thing," 2002.

[26] Sushant Jain, Ratul Mahajan, and David Wetherall, "A study of the performance potential of dht-based overlays," in *Proc. of USITS*, 2003.

[27] Samir Khuller and Uzi Vishkin, "Biconnectivity approximations and graph carvings," in *Proceedings of ACM STOC*, 1992.

[28] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM TOPLAS*, vol. 5, no. 1, pp. 66–77, January 1983.

[29] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker, "Topologically-aware overlay construction and server selection," in *Proc. of IEEE INFOCOM*, 2002.

[30] Gurdip Singh and Arthur J. Bernstein, "A highly asynchronous minimum spanning tree protocol," *Distributed Computing*, vol. 8, no. 3, 1995.

[31] H. N. Gabow, "A matroid approach to finding edge connectivity and packing arborescences," *J. Computer & System Sciences*, 1995.

[32] James Roskind and Robert E. Tarjan, "A note on finding minimum cost edge disjoint spanning trees," *Mathematics of Operations Research*, vol. 10, no. 4, pp. 701–708, November 1985.

[33] B. Awerbuch, "Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems," in *Proceedings of STOC*, 1987, pp. 230–240.

[34] Nancy Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, Inc., 1996.

[35] Radia Perlman, *Interconnections: bridges and routers*, Addison Wesley Longman Publishing Co., Inc., 1992.

[36] Michalis Faloutsos and Mart Molle, "What features really make distributed algorithms efficient," *International Conference on Parallel and Distributed Systems*, 1996.

[37] Ramesh Govindan and Hongsuda Tangmunarunkit, "Heuristics for internet map discovery," in *Proc IEEE Infocom*, 2000.

[38] Hongsuda Tangmunarunkit, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger, "Network topology generators: Degree based vs. structural," in *Proceedings of ACM SIGCOMM*, 2002.