Computer Science at Kent

MeshTree: A Delay-optimised Overlay Multicast Tree Building Protocol

Su-Wei Tan, Gill Waters, John Crawford

Technical Report No. 5-05
April 2005

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

Abstract

We study decentralised low delay degree-constrained overlay multicast tree construction for single source real-time applications. This optimisation problem is NP-hard even if computed centrally. We identify two problems in traditional distributed solutions, namely the greedy problem and delay-cost trade-off. By offering solutions to these problems, we propose a new self-organising distributed tree building protocol called MeshTree. The main idea is to embed the delivery tree in a degree-bounded mesh containing many low cost links. Our simulation results show that MeshTree is comparable to the centralised Compact Tree algorithm, and always outperforms existing distributed solutions in delay optimisation. In addition, it always yields trees with lower cost and traffic redundancy.

1 Introduction

This paper considers the problem of constructing "good" distribution trees for real-time applications such as audio/video conferencing and live webcasting from a data source. For these applications, low-latency delivery is of paramount importance. To accommodate large member populations, a cost-effective delivery mechanism such as multicasting is necessary. Since IP multicast has not been widely available, we consider the problem in the context of application layer multicast (ALM).

ALM implements multicast functions such as membership management and packet replication directly at the end systems. The end systems are organised into a logical overlay network, and multicast data using the overlay edges which are unicast tunnels. Hence ALM bypasses the need for network layer multicast support.

Creating an efficient overlay multicast tree is a challenging task. First, routing on top of the overlay results in redundant data traffic and prolonged end-to-end delay [8]. Secondly, end systems lack knowledge of the underlying topology, which is the key to building efficient overlays. As end systems often have limited processing power and available bandwidth, a degree constraint must be enforced in the delivery structure. In addition, the overlay structure is highly dynamic as it is formed by end systems that are prone to failures and may join/leave the session at will.

The above challenges and the requirement of low delay delivery in the real-time applications considered suggest that a solution must exhibit the following features.

- *Scalability*. This requires a decentralised scheme which imposes little protocol overhead (in terms of messages used to infer the node-to-node distances and to exchange the state information between the nodes).

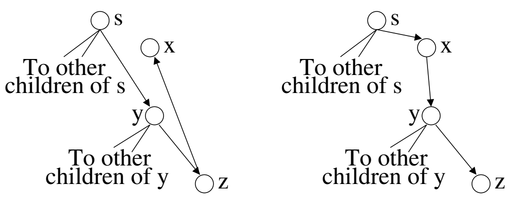
- *Optimised Structure*. The overlay tree must be degree-bounded while providing low source to receivers latency. This however, is an NP-hard optimisation problem [20].
- *Adaptability*. The solution must be able to react quickly to changes in the overlay membership (join/leave/failure) and network conditions.

We propose MeshTree, which fulfils the above properties in the following manner. First, MeshTree constructs overlay trees in a self-organising and fully distributed manner. The process uses limited network measurements and limited coordination between the nodes, hence has low overhead. MeshTree follows a mesh-based approach where the degree-bounded delivery tree is derived from a mesh. The mesh structure is inspired by the greedy problem and delay-cost trade-off (to be explained shortly) observed in distributed delay optimisation. We believe the structure offers a solution to the problems, and hence has low latency property. A mesh is also adaptable and provides good robustness.

Using detailed simulation experiments, we show that MeshTree is comparable to the centralised Compact Tree algorithm [20]. In addition, it always performs better than a distributed scheme proposed by Banerjee et al. [5] — which we have previously shown [21] to outperform switch-trees, HostCast, TBCP [15], HMTP [25] and AOM [23] in delay optimisation. We also show that MeshTree offers quick failure recovery and is very responsive to changes in group membership.

Problems in Distributed Delay Minimisation Intuitively, a good solution to the delay optimisation problem can be achieved with a delay-centric approach where nodes are placed as close as possible to the root. However, in a distributed environment where the degree-bounded nodes need to make decisions based on limited coordination using little knowledge about the topology, this approach can result in a greedy problem.

We explain this problem by using two existing decentralised protocols, i.e. switch-trees [12] and HostCast [14]. In these protocols, every node (except the root) maintains the delay from the root via the overlay tree to itself. Periodically, a node will try to improve its on-tree position by finding a better parent, i.e. a non-descendant node that provides a lower delay to the root. However, when a node has reached its degree bound, it will reject any new request from potential children. This can force nodes that are ideally placed near to the root to be placed far from the root. For example, Fig. 1(a) shows node x which is topologically close to the root is positioned under z that is far from the root. As y (as well as other children of s) has found the best possible parent (the root), it will greedily stick to s. This may prevent a better configuration (e.g. see Fig. 1(b)) from happening.



(a) Sample Configuration (b) A Better Configuration

Figure 1: The greedy problem

The configuration in Fig. 1(b) suggests that the greedy problem can be avoided if the end systems are connected based on their relative position in the underlying topology, i.e. if nodes close by are connected together. As the aim is to construct a tree, we can view this as the minimum spanning tree problem with the delay between two nodes as the cost function. First, creating a degree-bounded minimum spanning tree is an NP-hard problem [13]. In addition, a low cost tree often results in high end-to-end delay [21]. Our proposed mesh structure addresses the above two problems.

In the rest of this paper, we first discuss related work in next section. Section 3 describes MeshTree while Section 4 presents our evaluation results and discussion. Finally, Section 5 concludes the paper.

2 Related Work

In general, ALM protocols can be classified as either tree-based or mesh-based [9]. Several distributed tree-based protocols also attempt to create optimised delivery trees. Most notably, HMTP constructs low cost trees; HostCast, on the other hand, creates low delay trees; AOM attempts to achieve a balance between cost and delay; TBCP and switch-trees define generic techniques which can be adapted to different metrics, e.g. cost and delay. Our previous study has shown that the Banerjee at al. scheme that we use to compare with MeshTree, has superior delay performance to the above protocols. Yoid [11], another tree-based protocol, includes additional links to improve the tree robustness. However, these additional links are added without considering the degree constraint — hence, may not be useful in degree-bounded tree restoration. In addition, Yoid does not focus on overlay optimisation. NICE [4] and ZIGZAG [22] use a hierarchical cluster-based approach to construct overlay trees for large-scale applications. However, the resultant overlays are not degree-bounded based on a individual node's capacity constraint.

Several projects also consider the mesh-based approach for overlay construction. Narada [8] and Gossamer [7] run the path-vector protocol over a mesh overlay to derive source-specific tree for each node, which is more suitable for many-to-many applications. Scribe [6] and CAN-multicast [17] build trees on top of the mesh overlay built with the distributed-hash table (DHT) techniques (i.e. Pastry [19] and CAN [16] respectively). However the DHT-based overlays provide scalable and robust data distribution at the expense of added difficulties in achieving a tree-wide optimisation [18]. The HyperCast project [3] studies the use of overlays based on geometric properties of logical graphs, e.g. hypercube and Delaunay triangles. The performance of these overlays depends highly on the mapping between the underlying network metrics and the geometric space.

3 MeshTree

MeshTree is motivated by the greedy problem and the delay-cost trade-offs discussed previously. It attempts to solve the problem by creating an overlay mesh which consists of two main components: (i) a backbone structure; and (ii) additional links to form the mesh. The backbone is a low cost spanning tree[1] rooted at the source node. The low cost backbone connects nodes that are topologically close together, and thus helps to avoid the greedy problem. As a spanning tree uses the minimal number of links, additional links can be included in the overlay to improve the delay properties of the low cost tree. The resultant mesh is degree-bounded based on each individual node's capacity constraint. Finally, the actual delivery tree is derived from the mesh using a path-vector routing protocol.

Indeed, one could use more flexible overlay reconfiguration operations and additional information such as subtree delay to improve upon switch-trees and HostCast performance. Banerjee et al. [5] provide such a scheme. In Section 4, we show that our approach achieves better performance than their solution.

Maintaining a mesh has several advantages over maintaining only the delivery tree structure — an approach adopted by several tree-based protocols (e.g. HMTP and switch-trees). First, a mesh consists of multiple paths, and hence is more robust than a tree structure which can be partitioned by a single node failure. The multiple paths property is also useful for the overlay optimisation (see Section 3.3.1). Finally, we can take advantage of the standard routing protocol to construct the delivery tree. The routing protocol automatically handles the potential looping problem in distributed tree maintenance.

[1] In fact, the backbone tree is only loosely maintained, and it is possible that the backbone may be partitioned into a forest of trees (see 3.3.2).

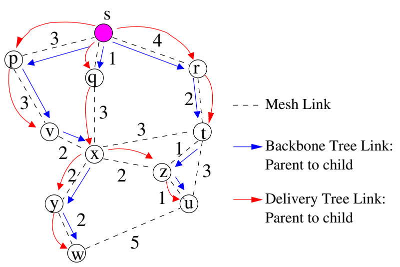


Figure 2: An example of MeshTree overlay

3.1 MeshTree Overlay Structure

MeshTree's overlay includes a backbone tree and a delivery tree, both rooted at the data source, s. The degree bound for a node, i is given by f_i^{max} which includes the incoming link from the parent (except when $i = s$) and the maximum number of out-going links to the set of downstream children.

Two nodes are said to have a neighbouring relationship when the overlay link between them exists in the constructed mesh. In general, the set of neighbours for a node, i is represented by N_i. We then define N_i^b as the set of i's neighbours in the backbone tree, where $N_i^b = \{p_i^b\} \cup C_i^b$, p_i^b and C_i^b are i's parent and children in the backbone tree respectively. Similarly, N_i^d, p_i^d and C_i^d are used for the delivery tree. N_i^o represents i's non-tree neighbours, where $N_i^o = N_i \setminus (N_i^b \cup N_i^d)$.

Hence, $N_i = N_i^b \cup N_i^d \cup N_i^o$. In addition, we define N_i^w as the set of nodes that i has agreed to accept as neighbours, while waiting for the neighbour setup process to be completed. During the execution of the protocol, i strictly enforces the fan-out constraint, f_i^{max} by $|N_i| + |N_i^w| \leq f_i^{max}$. As the delivery tree is derived from the degree-bounded mesh, the fan-out constraint for each node is guaranteed.

Fig. 2 shows a sample MeshTree overlay where s is the data source. The dashed lines define the mesh links. The backbone and delivery trees links are shown as blue and red lines connecting a parent to its child, respectively. The value beside a link represents its delay. For node x, we can see that $N_x = \{q, t, v, y, z\}$; $N_x^b = \{v, y\}$ with $p_x^b = v$; $N_x^d = \{q, y, z\}$ with $p_x^d = q$, and $N_x^o = \{t\}$. The example shows that some of the mesh links provide shortcuts which reduce the delay from the source. It also shows that a neighbour of a node can assume more than one role.

3.2 State at a MeshTree Node

For a node i, we use $\Upsilon_i(j)$ to denote the overlay distance from s to i via i's neighbour, j via the delivery tree. As the shortest path routing protocol is used, the distance via the delivery tree parent, p_i^d is the shortest. We denote the maximum subtree delay, Λ_i as the distance from i to its farthest descendant via the delivery tree. We use $\Lambda_i(c)$ to represent the subtree delay observed by i via its delivery tree child, c, i.e. $\Lambda_i(c) = d(i, c) + \Lambda_c$, where $d(i, c)$ denotes the unicast distance between i and c. If i is a leaf node, $\Lambda_i = 0$. Node i can estimate the total tree height in delay due to itself, H_i as: $H_i = \Upsilon_i(p_i^d) + \Lambda_i$. Referring to Fig. 2, for node x: $\Upsilon_x(q) = 4$, $\Upsilon_x(v) = 8$, $\Upsilon_x(t) = 9$, $\Lambda_x(y) = 4$, $\Lambda_x(z) = 3$. Hence, $\Lambda_x = \Lambda_x(y) = 4$ and $H_x = \Upsilon_x(q) + \Lambda_x = 8$.

Each node, i keeps the following state information: (i) *Backbone information*: The overlay path from s to i via the backbone tree, i.e. backbone root path; (ii) *Routing information*: The best overlay distance ($\Upsilon_i(p_i^d)$) and path to s; (iii) *Alternate path information*: The overlay distance of the best alternative path to s, if any; (iv) *Subtree information*: The maximum subtree delay, Λ_i; (v) *Members list*: Node i keeps a small list of members currently in the overlay. The list comprises nodes within a predefined neighbourhood of i, and nodes learned via a simple gossip-style node discovery technique [7]. The predefined neighbourhood for i consists of i's grandparent, siblings, and its parent's siblings in the backbone tree. The members list is used in overlay optimisation and is only lazily maintained.

Each node also maintains the routing information for all of its mesh neighbours, and the alternative path and subtree information for its delivery tree children. This information is obtained from the routing and refresh messages exchanged with the respective neighbours.

The alternative path and subtree information of the delivery tree children is needed for the overlay optimisation process to be described in Section 3.3.1. Basically, a node needs to estimate the tree height for the children using an alternative parent. Referring to Fig. 2 again, node z has an alternative path to s via t, $\Upsilon_z(t) = 7$, and has a maximum subtree delay, Λ_z = 1. Hence, x can calculate the tree height for z (using an alternative parent) as $H_z = \Upsilon_z(t) + \Lambda_z = 8$.

3.3 MeshTree Protocol Description

MeshTree initially constructs a random structure and relies on a periodical overlay reconfiguration process to improve the overlay towards the desired structure.

The random mesh is constructed in the following manner. Each newcomer, say x contacts a well-known Rendezvous Point [11] to obtain a small list of overlay members. From the given list, x randomly picks a number of joining targets (limited by f_x^{max}) and sends to each of them a joining message. It repeats the process (from other nodes) until it is accepted by at least one of the targets. On receiving a join request message from x, a node say y will accept x if it has spare fan-out, or if it has a neighbour, say z in N_y^o (in this case, y will drop z to maintain its degree bound). Node x will take the first node that accepts itself as its parent node in the backbone and delivery trees. For other nodes that are also able to accept x as a child, x converts them into non-tree neighbours. The randomised joining process quickly attaches new nodes into the overlay. In addition, it distributes the joining overhead among the members, rather than overloading a single node (e.g. the root).

Once joined to the overlay, x participates in the path-vector routing protocol [8] (with s as the only destination in the routing table) to derive the data delivery tree rooted at s. Using the whole path prevents the well-known count-to-infinity problem. We next explain how the overlay is optimised and maintained.

3.3.1 Overlay Improvement

The improvement process is needed to optimise the initial random structure as well as to adapt the overlay to the changes in the overlay memberships (join/leave/fail) and in the underlying network conditions.

The improvement process involves adding/deleting links to/from the overlay using a set of local rules running at each node. The rules prioritise the minimisation of the (backbone tree) cost over the (delivery tree) delay so as to obtain a low cost backbone augmented with short-cut links — our desired structure. Each reconfiguration operation consists of a request-reply-acknowledgement cycle between the end-points of a link, and requires no global coordination with other nodes in the overlay.

We next describe the request-reply-acknowledgement sequence. Hereafter, we refer to x as the node that is initiating an improvement process.

3.3.1.1 Request Component In this component, x selects a potential neighbour (hereafter refer to as y), and initiates a peering request to y.

In order to select y, x first picks a fixed number of non-neighbour nodes (5 in our implementation) as candidates from its known member list (see Section 3.2). Node x then probes the candidates to estimate the distance from itself to these nodes. From the measurement results, if there exists a candidate that is nearer to x than p_x^b, the node is selected as y. Otherwise, x randomly picks a node from the candidates list as y. Node x will then send a peering request message to y. The message contains the distance measured (i.e. $d(x, y)$) and the root paths and costs of x's backbone and delivery trees.

3.3.1.2 Reply Component When y receives x's request, it decides whether to accept x as a mesh neighbour, or a backbone tree child or parent, or simply reject x's request. The result will be conveyed to x by a peering reply message. If y

accepts x, the message will also contains y's backbone and delivery trees' information. In addition, y will add x into N_y^w while waiting for the acknowledgement from x.

The admission control algorithm consists of two main parts: (i) determine the neighbouring relationship with x; and (ii) decide whether to accept or reject x's request.

Determining the Neighbour Type This process checks if x should be regarded by y as a backbone tree parent or child, or a mesh neighbour. The decision rules prioritise the backbone peering.

We first determine the relationship between x and y on the backbone tree, i.e. if x is an ancestor of y, or vice-versa, or neither of these two. This is to prevent an ancestor from using its descendant as parent in the tree, i.e. a simple loop prevention. The relationship can be easily inferred from the backbone root paths for x (included in the request message) and for y. As a descendant, y will treat the ancestor (x) as a potential backbone parent if $d(x, y) < d(y, p_y^b)$. (A similar consideration is needed in the case that y is x's ancestor). Otherwise, y will regard x as a mesh neighbour.

If x and y are unrelated in the backbone tree, one can freely become parent or child of the other node and vice-versa. In this case, we compare the distance between x and y, x and p_x^b (given in the request message), and y and p_y^b. If $d(x, y)$ is smaller than one of the $d(x, p_x^b)$ and $d(y, p_y^b)$, or both of them, the node (x or y) that has a larger distance from its parent will become the child node. This helps to reduce the overall cost. Finally, if $d(x, y)$ is the largest among the distances, y will regard x as a mesh neighbour.

Accept / Reject The main decision making is based y's available capacity. Node y will accept x if: (i) $|N_y| + |N_y^w| < f_y^{max}$; or (ii) $|N_y| + |N_y^w| - |N_y^o| < f_y^{max}$. The first condition happens when y still has spare fan-out for new neighbours. Otherwise, if there exists a neighbour in N_y^o, then x can still be accepted. In this case, a randomly selected node from N_y^o will be dropped. In other cases, y executes a pruning decision to determine if it is beneficial to drop an existing neighbour to accept x. N_y^w is added into the equations above to enforce y's degree constraint.

By referring to the definitions described in Section 3.2, a neighbour v is said to be *prunable* if: (i) $v \equiv p_y^b$ and x is accepted as the new backbone parent, i.e. y is trying to switch to a closer backbone parent; (ii) $v \equiv p_y^d$ and x provides a shorter route to the root, i.e. $\Upsilon_y(x) < \Upsilon_y(p_y^d)$; (iii) $v \in C_y^d$ and v has an alternate path to s, and the total tree height due to v (estimated by y) does not exceed that of y, estimated after omitting v. This is to prevent an increase in the delivery tree height;

if $(p_y^b \notin N_y^d) \Rightarrow$ **return** p_y^b;
if $(p_y^b \equiv p_y^d \wedge \exists$ alt. path to $s) \Rightarrow$ **return** p_y^b;
if $(p_y^b \in C_y^d \wedge H_{p_y^b} < H_y) \Rightarrow$ **return** p_y^b;
if $(\Upsilon_y(x) < \Upsilon_y(p_y^d))$
  if $(p_y^d \notin C_y^b) \Rightarrow$ **return** p_y^d;
  elsif $(p_y^d \in C_y^b \wedge d(y,x) < d(y,p_y^d)) \Rightarrow$ **return** p_y^d;
if $(H_c < H_y : c \in C_y^d \wedge c \notin C_y^b) \Rightarrow$ **return** c;
if $(d(y,x) < d(y,c) : c \in C_y^b \wedge c \notin C_y^d) \Rightarrow$ **return** c;
if $(H_c < H_y \wedge d(y,x) < d(y,c) : c \in C_y^b \wedge c \in C_y^d) \Rightarrow$ **return** c;
return nil;

Figure 3: Conditions used by y to determine a prunable neighbour so as to accept x as the backbone tree parent

(iv) $v \in C_y^b$ and $d(x,y) < d(y,v)$ — an attempt to minimise the backbone cost.

Based on the above criteria and the new neighbouring relationship to be established, we devise a set of conditions to determine if an existing neighbour can be pruned. Fig. 3 depicts the pruning conditions if x is to be added as y's backbone tree parent. The conditions find a prunable neighbour (if any) by trying to maintain the tree structure of the backbone and prioritise cost over delay optimisation. In particular, y considers dropping p_y^b only if x is replacing it, and the delivery tree neighbours are considered before the backbone children. For the case that y is accepting x as a backbone child, the test for the backbone parent will be excluded; If x is to be added as a mesh neighbour, y will only consider the cases for the delivery tree parent and children.

Referring back to Fig. 1, the configuration in panel (b) can be achieved if node s can accept x by pruning an existing child. Otherwise, when one of the children finds a closer node as backbone parent, it may detach itself from s to allow an alternative configuration.

The prunable neighbour will not be dropped until y receives a positive feedback from x. To prevent transient disruption to the data delivery, a parent node continues to transmit data to a pruned child for a short time.

3.3.1.3 Acknowledgement Component When x receives the acceptance reply from y, it tries to admit y using the same admission control procedure described in the previous section (using the neighbour type determined by y). This is to prevent any discrepancy between the nodes. Based on the admission result, x finalises the whole process by either adding a link to y or rejecting the link establishment.

3.3.2 Overlay Maintenance

The overlay maintenance takes care of the peering relationship between a node and its neighbours. If a node leaves the session, it needs to inform its neighbours so that they can adapt to the change quickly. On the other hand, if a node fails, it is the responsibility of its neighbours to detect the case. This is done by periodically exchanging refresh messages between two neighbours. When a node stops receiving refresh messages from a neighbour for a predefined time, it regards the neighbour as having failed. The node, say x that detects the departure of its neighbour, say y, will trigger the following failure handling mechanism.

First, if $y \in N_x^o$ or $y \in C_x^d$, x only needs to update the corresponding list. If y is p_x^d, x will first try to recompute an alternate path to the root. If no alternate path exists, x will trigger a rejoin process from nodes contained in its delivery tree root path, starting from its grandparent.

Unlike the delivery tree, the backbone structure is only loosely maintained. Normally, the joining and optimisation procedures will result in a loop-free backbone tree. However, occasionally, a loop may be formed due to multiple simultaneous transformations or a transformation which is done based on stale information [11]. If x finds that a loop has been formed (by inspecting the root path carried in the refresh message from its p_x^b) or detects the departure of its p_x^b, it will update its descendants with a new root path which contains only the information of itself. This essentially results in a forest of low cost trees (rooted at various nodes), rather than a single spanning tree (rooted at s). In the case of loop, x withdraws its child status from the parent node while keeping the overlay link and x will attempt to acquire a new parent during the next improvement round.

4 Performance Evaluation

We conduct extensive simulations on a set of Transit-Stub topologies generated using the GT-ITM topology generator [2] as well as the power-law topologies generated by the Inet generator [1]. We report representative results from a 10100 nodes transit-stub network. (Similar performance trends were observed in the power-law topologies.) For all experiments, each end system is randomly attached to one of the routers. For all the results (except Fig. 5(a)) to be presented, each data point in the graph represents averages over 50 independent runs.

We first compare the quality of the delivery trees built by MeshTree with the following two schemes.

Compact Tree Algorithm (CPT) [20]: CPT is a centralised greedy heuristic for the degree-bounded minimum tree-diameter (i.e. the maximum distance on the tree between any two nodes) problem. Since we study the single source application, we modify the objective function to minimise the maximum delay from the root, rather than the tree diameter.

Banerjee et al.'s scheme : This is a simple variant of the distributed iterative tree-based scheme proposed in [5] . Each on-tree node maintains the overlay distance from the root and the subtree information as in MeshTree. A set of local transformations such as switching and swapping are defined to reconfigure nodes within two levels of each other towards the optimisation objective. In addition, swapping between two randomly selected nodes is done probabilistically so as to avoid local minima. In [5], the scheme is used to solve the minimum average delay problem which is more suitable for a proxy-based architecture. It has been shown to outperform a variant of CPT for the problem. We modify the objective function based on the suggestion in the paper to minimise the root delay. In order to capture the main property of the scheme while avoiding complication in distributed tree maintenance, we use a flow-level approach for the transformation process. Specifically, a transformation operation is made directly without actual message exchanges between the nodes involved.

The quality of the overlay tree is judged by the following metrics: (i) *RMP and RAP*; (ii) *tree cost ratio (TCR)*; and (iii) *link stress*. RMP and RAP are two variants of relative delay penalty [8]. RMP (RAP) is the ratio between the maximum (average) overlay delay and the maximum (average) delay using unicast from s to all other nodes. Hence, RMP represents our optimisation objective, while RAP indicates the average delay observed by the receivers; Tree cost [25] is defined as the sum of delays on the tree's links. It provides a simplified view of the total network resource consumption of a tree. TCR is the ratio of the cost between an overlay tree and the corresponding network layer multicast tree; Finally, link stress is defined as the number of duplicated copies of an identical packet sent over a single link.

In the experiments, all members randomly join to the session within the first 50s. The first member automatically becomes the data source. The out-degrees of the overlay nodes are uniformly distributed between 2 and 10. For MeshTree, we use one initial join target (see Section 3.3) and 5 candidates per improvement round (see Section 3.3.1). Both MeshTree and Banerjee et al.'s scheme use an improvement period of 30s, and the results are collected after the trees converge.

In terms of delay performance (RMP and RAP as in Fig. 4(a) and (b)), we can see that MeshTree always outperforms the Banerjee et al.'s scheme. For group sizes from 32 to 256, it produces trees with lower RMP and similar RAP compared to CPT. For larger group sizes where we expect a centralised approach to be unsuitable, MeshTree still shows reasonably good delay properties.

Fig. 4(c), (d) and (e) depict the worst-case and average link stress, and the TCR performance. We can now observe that CPT produces low delay trees at the expense of high traffic redundancy and network resource usage. The fact that its worst-case stress grows rapidly also suggests that it is not suitable for larger group sizes. MeshTree shows a much lower maximum stress performance, which is close to that of the Banerjee et al.'s scheme. In addition, it always shows the lowest average link stress and tree cost properties.

In the results presented, each data point in the figures represents the average for experiments using different sets of randomly chosen members (We make sure that all schemes were run with the similar set). For an iterative improvement scheme, it is desirable that given the same data source and members, a solution should converge to the same point regardless of the joining sequence. We randomly chose a set of members and conducted 50 runs using different joining sequences. The result in Fig. 4(f) shows that MeshTree consistently produces trees with about the same delay property, compared to the Banerjee et al.'s scheme. This suggests that it can avoid the inefficient structure better than the delay-centric approach.

We also conducted experiments where the source uses a direct unicast connection to each of the receivers. Obviously, this has the best delay performance. However, it overloads the source node, and results in a worst-case stress that is as high as the group size. It also incurs much higher resource usage, for example, its TCR is about 16.5 for a group size of 1024 members — an order higher than the ALM solutions. The rest of the results present several interesting properties of MeshTree. Fig. 5(a) shows the evolution of the TCR of the backbone tree, the RAP and RMP of the delivery tree, for an 1024-node overlay. We can see that these values increase quickly as nodes are joining the overlay. This is because the initial overlay is randomly connected. In the experiment, each receiver has a improvement period of 30s. We can see that the RAP and RMP values rapidly decrease to a value less than 2 within the first 200s, i.e. less than 10 improvement rounds per node. This indicates that MeshTree can converge very quickly. The result also shows that MeshTree can achieve low backbone tree cost ratio, which suggests that the overlay contains a lot of short links between the members. This helps to reduce the delivery tree cost and link stress, as observed previously.

The high delay and cost at the early stage is obviously undesirable. While not shown here, we have found that this can be greatly improved by using a larger number of joining targets. For example, we can achieve a 50% improvement by using two initial join targets per node.

An overlay tree needs to be restored after a non-leaf node, say x departs (fails or leaves). It is important that x's children can quickly locate a new parent to

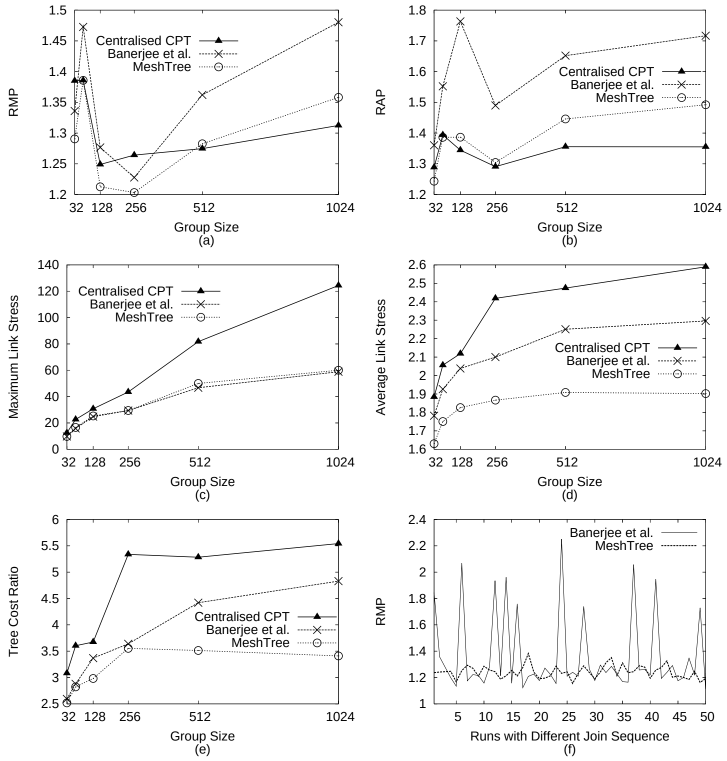


Figure 4: Comparison results: (a) RMP; (b) RAP; (c) Maximum link stress; (d) Average link stress; (e) Tree cost ratio; (f) Impacts of join sequence.

resume the data flow. In addition, the recovery process should not result in a fan-out violation in any node. We compare MeshTree with the grandparent recovery scheme studied in [24][2], in which the children of the departed node first try to attach to their grandparent. The grandparent will try to accommodate them as long as it has spare capacity. Otherwise, it will redirect them to its descendants.

In the experiments, we run the grandparent recovery scheme on trees constructed by MeshTree before a departure event. A number of randomly selected nodes are instructed to leave the group at the same time. We calculate the average recovery time as the average time for an affected node to find a new parent. We use a 256-node overlay, and the out-degrees of the nodes are randomly distributed between 2 and 6. Hence, even a small number of departures will result in overlay reconstruction. The result depicted in Fig. 5(b) shows that MeshTree can recover faster than the grandparent scheme.

Fig. 5(c) shows the protocol overhead (control message traffic sent and received) per overlay node, in kbps. In the experiments, each node has a maximum of 10 neighbours. We assume that each message is carried using TCP over IPv4, which incurs a base size of 40 bytes per packet. The result shows that the per node overhead is reasonably small, i.e. less than 1kbps, and it increases very slowly with the group sizes. We note that the result does not include overheads due to network measurement. First, each node only needs to know the distance to a small number of members. The distances can be *cached* to reduce measurement overhead. In addition, it may be possible to obtain the distance information from the Internet distance services (e.g. IDMaps [10]).

5 Concluding Remarks

This paper studies the problem of creating overlay multicast trees for real-time applications. We present MeshTree, which constructs overlay trees in a fully distributed manner using only limited overlay member information. It has low protocol overhead, and shows fast convergence and failure recovery properties. The constructed trees are degree-bounded based on an individual node's capacity constraint. Our simulation experiments reveal that the trees built with MeshTree have delay properties comparable with (and sometimes better than) a centralised algorithm, and always have lower delay than a decentralised scheme. Both alternative schemes compared have previously been shown to perform well in their class. In addition, trees constructed with MeshTree consume fewer network resources.

[2] In [24], the authors proposed a proactive tree recovery approach for creating minimum cost trees, hence it is not used here. The grandparent scheme is the best reactive approach as shown in the paper

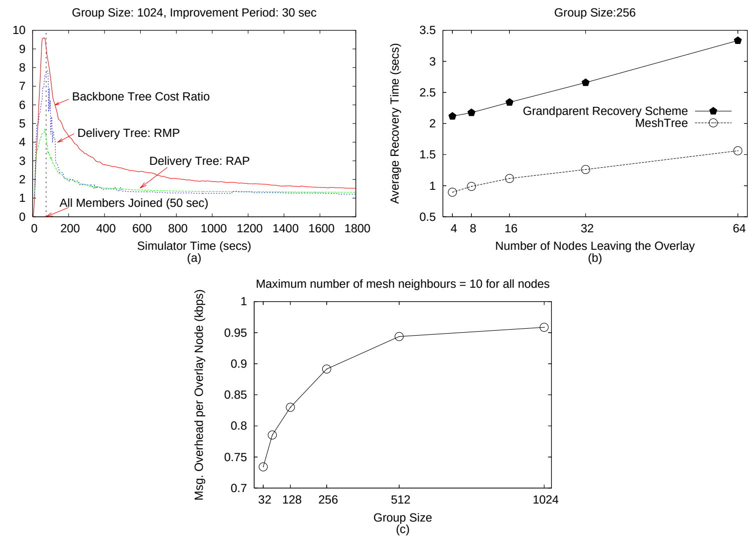


Figure 5: MeshTree properties: (a) Convergence; (b) Failure recovery; (c) Protocol overhead.

References

[1] Inet topology generator, available at: topology.eecs.umich.edu/inet.

[2] Gt-itm topology generator, available at: www.c.gatech.edu/Ellen.Zegura/graphs.html.

[3] The hypercast project, http://www.cs.virginia.edu/ mngroup/hypercast/.

[4] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *ACM SIGCOMM*, pages 205–220, Pittsburgh, PA, 2002. ACM.

[5] 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, USA, 2003.

[6] M. Casto, P. Druschel, A. M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralised application-level multicast infrastructure. *IEEE JSAC*, 20(8), Oct 2002.

[7] Y. Chawathe. *An Architecture for Internet Broadcast Distribution as an Infrastructure Service*. PhD thesis, University of California, 2000.

[8] Y. H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *ACM SIGMETRICS*, Santa Clara, CA, 2000.

[9] A. El-Sayed, V. Roca, and L. Mathy. A survey of proposals for an alternative group communication service. *IEEE Network*, 17(1):46–51, 2003.

[10] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A global internet host distance estimation service. *IEEE/ACM Trans. on Networking*, 9(5):525–540, Oct 2001.

[11] P. Francis, Y. Pryadkin, P. Radoslavov, R. Govindan, and B. Lndell. Yoid: Your own internet distribution. Unpublished, ISI, 2000.

[12] D. A. Helder and S. Jamin. End-host multicast communication using switch-trees protocols. In *GP2PC*, 2002.

[13] J. Konemann. *Approximation Algorithms for Minimum-cost Low-degree Subgraphs*. PhD thesis, CMU, 2003.

[14] Z. Li and P. Mohapatra. Hostcast: A new overlay multicast protocol. In *IEEE ICC*, Alaska, USA, 2003.

[15] L. Mathy, R. Canonico, and D. Hutchison. An overlay tree building control protocol. In *NGC*, London, UK, 2001.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, San Diego, Carlifornia, 2001.

[17] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *NGC*, London, UK, 2001.

[18] S. Ratnasamy, S. Shenker, and I. Stoica. Routing algorithms for DHTs: Some open questions. In *1st IPTPS'02*, Cambridge, MA, 2002.

[19] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware*, 2001.

[20] S. Y. Shi, J. S. Turner, and M. Waldvogel. Dimensioning server access bandwidth and multicast routing in overlay networks. In *NOSSDAV*, New York, USA, 2001. ACM.

[21] S. W. Tan, G. Waters, and J. Crawford. A study of distributed low latency application layer multicast tree construction. In *LCS'04*, London, UK, 2004.

[22] D. A. Tran, K. A. Hua, and T. T. Do. Zigzag: An efficient peer-to-peer scheme for media streaming. In *IEEE INFOCOM*, San Francisco, USA, 2003.

[23] S. Wu, S. Banerjee, X. Hou, and R. A. Thompson. Active delay and loss adaptation in overlay multicast tree. In *IEEE ICC*, Paris, France, 2004.

[24] M. Yang and Z. Fei. A proactive approach to reconstructing overlay multicast trees. In *IEEE INFOCOM*, H.K., 2004.

[25] B. Zhang, S. Jamin, and L. Zhang. Host multicast: A framework for delivering multicast to end users. In *IEEE INFOCOM*, pages 1366–1375, New York, USA, 2002.