Eliminating Bottlenecks in Overlay Multicast*

Min Sik Kim Yi Li Simon S. Lam

Department of Computer Sciences
The University of Texas at Austin
{minskim,ylee,lam}@cs.utexas.edu

TR–04–47
November 2004

Abstract

Recently many overlay multicast systems have been proposed to overcome limited availability of IP multicast. Because they perform multicast forwarding without support from routers, data may be delivered multiple times over the same physical link, causing a bottleneck. This problem is more serious for applications demanding high bandwidth such as multimedia distribution. Although such bottlenecks can be removed by changing overlay topology, a naïve approach may create bottlenecks in other parts of the network. In this paper, We propose an algorithm that removes all bottlenecks caused by the redundant data delivery of overlay multicast, detecting such bottlenecks using a wavelet-based technique we recently proposed. In a case where the source rate is constant and the available bandwidth of each link is not less than the rate, our algorithm guarantees that every node receives at the full source rate. Simulation results show that even in a network with a dense receiver population, our algorithm finds a tree that satisfies all the receiving nodes while other heuristic-based approaches often fail.

1 Introduction

Recently, many overlay multicast systems have been proposed as alternatives to IP multicast. Overlay multicast systems provide more flexibility in topology construction, but consume more bandwidth of an underlying network because data is often delivered multiple times over the same physical link, causing a bottleneck. This problem is more serious for applications demanding high bandwidth such as multimedia distribution. One way to mitigate the problem is to limit the fan-out of internal nodes in a multicast tree [2]. However, deciding the right number of children is non-trivial; fan-out should be a function of the available bandwidth to each child and the network topology. Furthermore, a bottleneck may be caused by flows from different source (parent) nodes. In such a case, fan-out has little to do with the bottleneck. Therefore, a better way to avoid bottlenecks is to identify them by finding unicast flows in the multicast session that traverse those bottlenecks, and remove them by changing the multicast tree topology. We recently presented a wavelet-based technique, called DCW (Delay Correlation with Wavelet denoising), which can be used to detect flows sharing bottlenecks (or congested links) [7]. While other techniques require that flows should share a common end point [4, 6, 9], DCW can be used for any pair of Internet flows with different sources and different sinks.

Identified shared bottlenecks should be eliminated by changing the overlay tree topology. However, it should be done very carefully. When a tree edge is cut, another edge must be added to maintain connectivity. But the newly added edge may cause another bottleneck. Even worse, eliminating the new bottleneck may reincarnate the old one, resulting in oscillation. The algorithm we propose in this paper removes shared bottlenecks without incurring such oscillation. We prove that the algorithm always terminates, and that on termination there is no shared bottleneck in the multicast tree. In a case where the source rate is constant and the available bandwidth of each link is not less than the source rate, our algorithm guarantees that every node receives at the full source rate. We implement our algorithm in a distributed fashion, and compare its performance with other heuristically-built multicast trees. Simulation results show that even in a network with a dense

*Research sponsored by National Science Foundation ANI–0319168 and CNS–0434515.

receiver population, our algorithm finds a tree that satisfies all the receiving nodes while other heuristic-based approaches often fail.

The remainder of this paper is organized as follows. Section 2 presents our network model. Section 3 proposes an algorithm to eliminate shared bottlenecks, and Section 4 sketches the implementation of a distributed protocol based on the proposed algorithm. Section 5 shows experimental results, and we conclude in Section 6.

2 Model

Our network model consists of two layers. The lower layer represents the underlying traditional network with links and nodes, where routing between nodes is done through the lowest-cost path. The upper layer is an overlay network, where a subset of the nodes in the lower layer form a multicast tree.

2.1 Underlying Network

An underlying network is given as a directed graph $G = (N, L)$, where N is a set of nodes in the network, and L is a set of unidirectional links between two nodes in N. Each link $(m, n) \in L$ has two properties: $B(m, n)$, the bandwidth of the link available to overlay multicast, and $c(m, n)$, the cost of the link. The cost is a positive constant and used as a routing metric to compute shortest paths. We assume symmetric routing, i.e. $c(m, n) = c(n, m)$.

Given two nodes u and v in N, the shortest path between them is specified as a set of links $P_L(u, v) = \{(u, n_1), (n_1, n_2), \ldots, (n_i, v)\}$, which are chosen to minimize the total cost of the links in the set. If there are more than one such paths, we assume that the routing algorithm always selects the same path among them.

2.2 Overlay Multicast Tree

A multicast tree is built on top of the underlying network G, using a set of end-hosts H. H is a subset of N, and consists of end-hosts participating the multicast session. The multicast tree is represented as a set $T = \{(u, v) | u, v \in H,\ v \text{ is a child of } u \text{ in the tree}\}$. We call each element of T an edge of the tree.

Similarly to P_L, P_T is defined as a path in a multicast tree T. Formally, $P_T(u, v) = \{(u, h_1), (h_1, h_2), \ldots, (h_i, v)\}$, where $P_T(u, v) \subset T$.

2.3 Bottleneck

We model multicast traffic as a set of flows; every edge of an overlay multicast tree has an associated flow for data delivery. Each flow f has a source node $Src(f)$, a sink node $Snk(f)$, and the rate of the flow $Rate(f)$.

Let $F(m, n)$ be a set of flows passing through the link $(m, n) \in L$. Formally, $F(m, n) = \{f | (m, n) \in P_L(Src(f), Snk(f))\}$. A link (m, n) is a *bottleneck* of the multicast session if and only if $B(m, n) < \sum_{f \in F(m,n)} Rate(f)$. The bottleneck link (m, n) is also called a *shared bottleneck* if multiple flows are passing through the link, or $|F(m, n)| > 1$, where the notation $|S|$ denotes the number of elements of a set (or vector) S.

3 Algorithm

The goal of our algorithm is to remove shared bottlenecks in a multicast tree, so that they cannot throttle throughput. In the algorithm we assume that each bottleneck shared by multiple flows can be detected accurately using a technique such as DCW [7]. Before we describe the algorithm, we define notation to be used.

- $r \in H$ denotes the root node of a multicast tree.
- $d(u, v)$ is the distance between u and v on T, namely $d(u, v) = |P_T(u, v)|$.
- $Parent(u)$ is the parent node of u in the tree.
- $SLeaf(u)$ is one of the shallowest leaves in a subtree rooted at u. In other words, $SLeaf(u)$ is a leaf node closest to u in the subtree.
- $Ram(u)$ is the node that has caused ramification of the branch of u in the tree, or $\varnothing$ if there is no such node. Formally, $Ram(u)$ is a node along the path from r to u such that $Parent(Ram(u))$ has more than one child, and all the nodes between $Ram(u)$ and $Parent(u)$, inclusively, have only one child. See Figure 1.

Shared bottlenecks need to be treated differently depending on their relative locations in the tree. There are two types of shared bottlenecks: intra-path and inter-path shared bottlenecks, as shown in Figure 2, where thick arrows represent flows sharing the same bottleneck. Suppose that a link $(m, n) \in L$ is a shared bottleneck. Then there exist two edges (u_1, v_1) and (u_2, v_2) such that $(u_1, v_1), (u_2, v_2) \in T$ and $(m, n) \in P_L(u_1, v_1) \cap P_L(u_2, v_2)$. Without loss of generality, we assume $d(r, u_1) \leq d(r, u_2)$. A shared

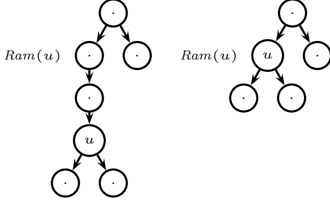

Figure 1: Ramification point

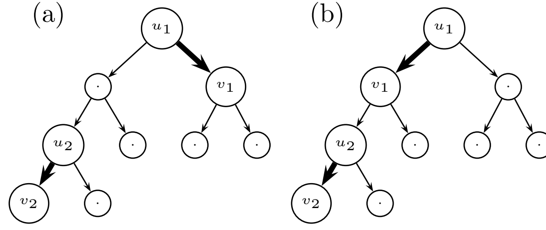

Figure 2: (a) Inter- and (b) intra-path shared bottlenecks

REMOVE-INTER-PATH-SHARED-BOTTLENECK
1 ▷ (u_1, v_1) and (u_2, v_2) in T are sharing a bottleneck, and $d(r, u_1) \leq d(r, u_2)$.
2 **if** $d(r, SLeaf(v_1)) \leq d(r, v_2)$
3 $\quad T \leftarrow T \cup \{SLeaf(v_1), v_2\} - \{(u_2, v_2)\}$
4 **else**
5 $\quad t \leftarrow$ a node such that $d(r,t) = d(r, v_2)$ and $\exists P_T(u_1, t) \neq \varnothing$, $P_T(u_1, t) \subset P_T(u_1, SLeaf(v_1))$
6 $\quad T \leftarrow T \cup \{(t, v_2)\} - \{(u_2, v_2)\}$
7 **if** $\{(u_2, x) | (u_2, x) \in T\} = \varnothing$
8 $\quad u, v \leftarrow Parent(Ram(u_2)), Ram(u_2)$
9 $\quad c \leftarrow \arg\min_{i \in \{w | Parent(w) = u, w \neq v\}} d(i, SLeaf(i))$
10 $\quad T \leftarrow T \cup \{(SLeaf(c), v)\} - \{(u, v)\}$

Figure 3: Removal of an inter-path shared bottleneck

bottleneck (m, n) is called an *intra-path shared bottleneck* of (u_1, v_1) and (u_2, v_2) if $(u_1, v_1) \in P_T(r, u_2)$, and otherwise an *inter-path shared bottleneck*. In this section, we describe first the algorithm for the more general case, inter-path shared bottlenecks, and then the algorithm for intra-path shared bottlenecks. By applying these algorithms iteratively, we can remove all shared bottlenecks in finite iterations. To make sure that it terminates, we will prove that the tree after each iteration is different from any tree in previous iterations.

For proof, we define two properties of a multicast tree: the leaf distance vector and total cost. The leaf distance vector is defined as $D = (d(r, u_1), d(r, u_2), \ldots, d(r, u_k))$, where $u_1, u_2, \ldots, u_k$ are all the leaf nodes in T, and $d(r, u_i) \leq d(r, u_{i+1})$ for every $i < k$. Distance vectors are ordered as follows. For two distance vectors, D and D', D precedes D' ($D \prec D'$) if and only if (i) $|D| > |D'|$, or (ii) $|D| = |D'|$ and D precedes D' in lexicographical order.

The second property, total cost C, is defined to be the sum of costs of all edges in the tree, where the cost of an edge is the sum of all link costs along the edge. Formally, $C = \sum_{(u,v) \in T} \sum_{(m,n) \in P_L(u,v)} c(m, n)$. For each link shared by multiple edges, its link cost is counted multiple times.

3.1 Inter-path Shared Bottleneck

The algorithm to remove an inter-path shared bottleneck is shown in Figure 3. See also Figure 2(a) for illustration. When an inter-path shared bottleneck is detected between two edges, (u_2, v_2), the edge farther from the root, is removed and the detached subtree rooted at v_2 is moved to the subtree rooted at v_1. If the shallowest leaf in v_1's subtree is not deeper than v_2, than it is chosen as the node to which v_2's subtree is attached. In this way, we can avoid increasing the fan-out of an internal node, which may affect flows from the node to the existing child nodes. However, since we don't want the tree to become too tall, we also avoid attaching v_2 to a very deep node. Therefore, if the shallowest leaf of v_1 is deeper than v_2, v_2 is attached to a node on the path from v_1 to its shallowest leaf such that the depth of v_2 increases at most by one.

If u_2 becomes a leaf after removing (u_2, v_2), we relocate its branch (the path from $Ram(u_2)$ to u_2) under another leaf in Lines 8–10, because leaving behind u_2's branch may cause oscillation. Suppose that the edge added to connect v_2 to the tree causes another shared bottleneck. Then it is possible that v_2 is detached once again and moved back to u_2, if u_2 is the chosen shallowest leaf in this case. Thus the change made to remove the shared bottleneck between (u_1, v_1) and (u_2, v_2) is reverted, and it revives the bottleneck that we removed earlier. By relocating u_2's branch when u_2 becomes a leaf, we can avoid such oscillations. The following lemma states that the leaf distance vector before REMOVE-INTER-PATH-SHARED-BOTTLENECK algorithm always precedes that after REMOVE-INTER-PATH-SHARED-BOTTLENECK. The proof is in Appendix A.

Lemma 1 *Let D and D' denote leaf distance vectors before and after* REMOVE-INTER-PATH-SHARED-BOTTLENECK *respectively. Then we have $D \prec D'$.*

3.2 Intra-path Shared Bottleneck

Figure 4 presents the algorithm to remove an intra-path shared bottleneck. See also Figure 2(b) for illus-

```
Remove-Intra-Path-Shared-Bottleneck
 1  ▷ (u_1, v_1) and (u_2, v_2) in T are sharing a bottleneck,
    and d(r, u_1) ≤ d(r, u_2).
 2  if SLeaf(v_1) ≠ SLeaf(v_2)
 3     Remove-Inter-Path-Shared-Bottleneck
 4  else if Ram(v_1) ≠ Ram(v_2)
 5     u, v ← Parent(Ram(v_2)), Ram(v_2)
 6     c ← arg min_{i∈{w|Parent(w)=u, w≠v}} d(i, SLeaf(i))
 7     if d(u, SLeaf(c)) ≤ d(r, u) + d(v, v_2) + 1
 8        T ← T ∪ {(SLeaf(c), v_2)} − {(u_2, v_2)}
 9     else
10        t ← a node such that
               d(r, t) = d(r, u) + d(v, v_2) + 1 and
               P_T(u, t) ⊂ P_T(u, SLeaf(c))
11        T ← T ∪ {(t, v_2)} − {(u_2, v_2)}
12     if {(u_2, x)|(u_2, x) ∈ T} = ∅
13        T ← T ∪ {(SLeaf(c), v)} − {(u, v)}
14  else
15     T ← T ∪ {(u_1, u_2), (v_1, v_2)} − {(u_1, v_1), (u_2, v_2)}
16     ∀(x, y) ∈ P_T(v_1, u_2), T ← T ∪ {(y, x)} − {(x, y)}
```

Figure 4: Removal of an intra-path shared bottleneck

tration. Some intra-path shared bottlenecks may be treated like inter-path shared bottlenecks, but others should be treated differently.

In the case of an intra-path shared bottleneck, the shallowest leaf of v_1 may be v_2 itself or a node in its subtree. If v_1's shallowest leaf is not in v_2's subtree, Remove-Inter-Path-Shared-Bottleneck is applied to attach v_2 to the shallowest leaf of u_1. Otherwise, we have two cases in Line 4, depending on whether there is any branch between v_1 and v_2. If there is, v_2's subtree is attached under that branch similarly as in an inter-path shared bottleneck case. Otherwise, it becomes a child of a node in the middle so that the depth of v_2 in increased at most by one. As in Lemma 1, this ensures that the leaf distance vector before Remove-Intra-Path-Shared-Bottleneck precedes that after Remove-Intra-Path-Shared-Bottleneck.

If there is no branch between v_1 and v_2, edges (u_1, v_1) and (u_2, v_2) are replaced with (u_1, u_2) and (v_1, v_2), and the edges in the middle are reversed so that the flow traverses in the opposite direction in Lines 15–16. Shortest-path routing guarantees that this reduces the total cost.

From the two cases above, we conclude the following lemma. Its proof is in Appendix B.

Lemma 2 *Let D and D' denote leaf distance vectors before and after* Remove-Intra-Path-Shared-Bottleneck *respectively, and C and C' be total costs before and after* Remove-Intra-Path-Shared-Bottleneck. *Then we have either $D \prec D'$ or $D = D'$ and $C < C'$.*

3.3 Shared Bottleneck Elimination

Using the previous two lemmas, we prove that our algorithm removes all shared bottlenecks from a multicast tree.

Theorem 1 *By applying* Remove-Inter-Path-Shared-Bottleneck *or* Remove-Intra-Path-Shared-Bottleneck *iteratively, all shared bottlenecks will be removed in a finite number of iterations.*

Proof If there exists a shared bottleneck in a multicast tree, either Remove-Inter-Path- or Remove-Intra-Path-Shared-Bottleneck can always be applied to remove it. Each of them changes the leaf distance vector or decreases the total cost while maintaining the same leaf distance vector by Lemma 1 and Lemma 2. For a leaf distance vector D, there are only a finite number of leaf distance vectors D' such that $D \prec D'$. And the total cost C can be reduced only a finite number of times because it is lower-bounded, and each time the amount of reduction is also lower-bounded by the minimum link cost, which is a non-zero constant. Therefore, all shared bottlenecks are removed within a finite number of iterations. □

Note that our algorithms remove shared bottlenecks, providing that the available bandwidth B and cost c of each link remain constant. In practice, however, since available bandwidth keeps varying and the set of participating hosts H changes, the multicast tree that the algorithm converges to may also change. Nevertheless, we believe that the algorithm that converges to the desired target in a static environment is a good starting point for a dynamic environment, and we will show empirically that the actual protocol based on our algorithm is in fact able to adapt as available bandwidth and node membership changes occur.

4 Protocol Implementation

Our algorithm is based on assumptions that a shared bottleneck is detectable, that information such as shallowest nodes and ramification points is available, and that each execution of Remove-Inter-Path-Shared-Bottleneck or Remove-Intra-Path-Shared-Bottleneck does not interfere with another execution. In this section, we explain how these assumptions can be satisfied in a protocol implementation. In addition, we briefly describe how our protocol handles node joins and leaves in a dynamic environment.

4.1 Shared congestion removal

In real networks, a shared bottleneck is a congested link shared by multiple flows belonging to the same multicast session. DCW [7] determines whether two flows are sharing such "shared congestion" with high accuracy ($> 95\%$) if one-way delay for each flow is measured for 10 seconds with a sampling frequency of 10 Hz. It tolerates a synchronization offset up to one second between different flows, which is achievable with loose synchronization among participating nodes as follows.

For loose synchronization, each non-root node sends a packet to its parent periodically, and the parent replies with a timestamp. On receiving the reply, the node calculates the round-trip time and sets its clock to the timestamp plus half the round-trip time.

Shared congestion is detected and removed on a round-basis. The start time of each round is publicized by the root node; each node obtains information on the epoch T_0 and round interval T_r from its parent, and starts a round at $T_0 + nT_r$ with its local clock, where n is an integer. In every round, a node performs the following three tasks sequentially. (i) At the beginning of each round, one-way delay from a node to each of its children that are experiencing congestion is sampled for 10 seconds with a frequency of 10 Hz as recommended for DCW [7]. (ii) After measurement, a node waits for reports from all child nodes; the reports contain delay samples of edges experiencing congestion in the subtree of the corresponding child node. Once all reports are received, the node selects edge pairs such that the edges in each pair share a bottleneck link with each other. The node must ensure that executions of the bottleneck removal algorithms do not interfere with each other. Since bottleneck elimination relocates nodes in the subtree of $Ram(v_2)$ only, the node can select as many pairs as it can, as long as such subtrees of selected pairs do not overlap. Then, among all congested edges in its subtree, the node reports delay samples of those edges that "would not interfere" with selected pairs. Because edges involved in removing a bottleneck are (u_1, v_1) and those in the subtrees of $Ram(v_2)$ and v_2 only, shared bottlenecks in other edges can be removed concurrently. Therefore, the node sends to its parent the delay samples of those congested edges that are not involved in removing a bottleneck of any selected pair. (iii) Finally, the node removes shared bottlenecks in its subtree by running the algorithm for every selected pair.

4.2 Information update

The algorithm requires that each node v should know $d(r, v)$, $SLeaf(v)$, and $Ram(v)$. These values are updated at each node u by exchanging information with its parent and with its child nodes when the values change. An information update packet from a parent u and a child v contains $d(r, u)$ and $Ram(v)$, which are used to update v's local information on $d(r, v)$ and $Ram(v)$. Similarly, an information update packet from a child v to its parent u contains $SLeaf(v)$, and u updates $SLeaf(u)$, $d(u, SLeaf(u))$, $SLeaf(v)$, $d(u, SLeaf(v))$, and the child node whose subtree $SLeaf(v)$ belongs to.

4.3 Membership management

We assume that a joining node obtains the address of the root node through an out-of-band channel, such as WWW. When it sends a join request to the root node, it is accepted as a temporary child. If the new node doesn't experience congestion during the next round, it becomes a permanent child. Otherwise it is forwarded to one of the existing children of the root node. This procedure is propagated along the tree until the joining node becomes a permanent child of an existing node. One concern is that congestion caused by a temporary child may affect other children. This can be avoided if a parent node uses a priority queue for its outgoing flows, in which packets to the temporary child have lower priority than others.

When a node leaves, its children become temporary children of the parent of the leaving node. Then the temporary children are treated as joining nodes. Node failures are handled in the same way.

5 Evaluation

To evaluate our protocol, we compare it against two heuristic-based schemes. The first one optimizes the multicast tree using bandwidth estimation as in Overcast [5]. Each node estimates available bandwidth from the grandparent, parent and its siblings using 10 kB TCP throughput, and then relocates below the one with the highest estimation. However, 10 kB TCP throughput doesn't have very strong correlation with available bandwidth. Taking into account that a path chosen using 10 kB TCP throughput provides only half of the bandwidth of the best path [8], we optimistically assume that the bandwidth estimation has maximum error of 20%.

The second heuristic scheme is based on delay measurement. It is similar to the bandwidth-based

one except that it selects the node with the shortest delay instead of the highest bandwidth and that the number of children each node can have is limited to four to avoid high fan-out.

For fair comparison, we also introduce errors in shared congestion detection. Since our goal is to show that our protocol performs better than heuristic-based ones, we conservatively assume that the detection error is 5%, which is higher than actual error rate (almost zero when measurement interval is longer than 10 seconds) of DCW [7]. Then we measure performance of each scheme using a flow-level simulator we wrote, where bandwidth allocation to flows is max-min fair. The relationship between the tree performance and error rate will also be presented.

5.1 Tree performance comparison

To demonstrate tree performance under heavy load, we run simulations on a network with a dense receiver population. The network topology is generated with GT-ITM [1]. There are 24 transit routers, 576 stub routers, and 1152 hosts participating the multicast session. The bandwidth (Mbps) of each link is randomly drawn from four different intervals: $[300, 1400)$ between transit routers, $[40, 70)$ between a transit and a stub router, $[5, 15)$ between stub routers, and $[1, 5)$ between a stub router and an end host. The source rate is set to 1 Mbps. Initially, the tree consists of the source (root) host only. All the other hosts then join the tree.

We use the following metrics to evaluate a multicast tree.

Link stress The number of flows in a multicast session that traverse a physical link. Defined in [3].

Link load The sum of required rates for all flows in a link divided by the bandwidth of the link.

Relative delay penalty (RDP) The ratio of the delay from the root to a node in a multicast tree to the unicast delay between the same nodes. Defined in [3].

Receiving rate The max-min fair rate assigned to a flow from the root to a node divided by the maximum rate of the flow (the source rate).

Below we show distributions of these metrics for trees built with the three different schemes: delay heuristic, bandwidth heuristic, and our bottleneck-free tree protocol. We run the bottleneck-free tree protocol until there is no shared congestion. The delay heuristic scheme is run until the tree doesn't change any more. However, the bandwidth heuristic may oscillate as shown in Overcast [5] because changing tree topology affects bandwidth estimation. Since Overcast becomes relatively stable after 20 rounds, we run the bandwidth heuristic up to 30 rounds.

Figure 5 shows the link stress distribution for links used by the multicast session. Since the delay heuristic tends to build a tree well-matched with the underlying topology, its link stress is far better than other schemes. Note that the bottleneck-free tree shows the worst performance in terms of link stress. However, this does not necessarily mean that it is abusing the network, because having a large number of flows in a link (high link stress) is totally acceptable if the link has available bandwidth to accommodate all of them. The next figure shows this point clearly.

Figure 6 presents the distribution of link load, which is the amount of bandwidth required to carry all flows traversing a link divided by the link's available bandwidth. Contrary to the previous result, the delay heuristic is the worst among the three; on some links, the required bandwidth to support the multicast session is more than 3.5 times the available bandwidth. This is because the delay heuristic often chooses a link with small bandwidth if the delay of a path going through the link is short. Note that link load with the bottleneck-free tree is always less than one. The bandwidth heuristic maintains similar performance as the bottleneck-free scheme, but some links have load higher than one. Since each of such links throttles receiving rates of the entire subtree connected upstream through the link, even a few of them may affect a large number of receiving nodes.

In Figure 7, the receiving rate distribution is shown. Due to high link load, all the receiving nodes in the delay heuristic tree receive less than half of the source rate. The distribution for the bandwidth heuristic shows the impact of the few links with high load in Figure 6; only less than 40% of the receiving nodes can receive data at the full source rate. The other 60% experience quality degradation due to bandwidth shortage. However, in the bottleneck-free tree, 100% of the receiving nodes receive at the full source rate since it maintains link load less than one. Usually such gain in receiving rate comes with the cost of longer delay. However, our algorithm is very careful in changing the tree topology not to increase depth of a relocated node unnecessarily. As a result, its RDP is only a little worse than the tree built with the delay heuristic, as illustrated in the distribution of RDP in Figure 8. Because the bandwidth heuristic pays little attention to delay, its RDP is worse than the others.

We have to mention that this experiment regarding

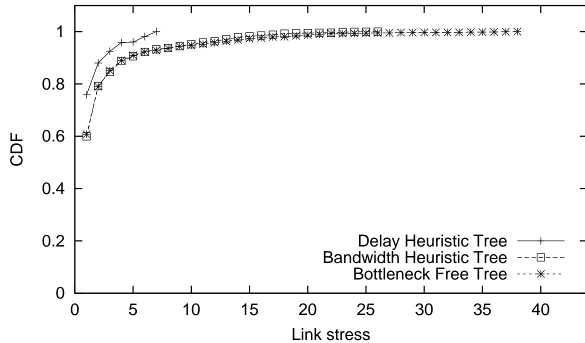

Figure 5: Link stress distribution

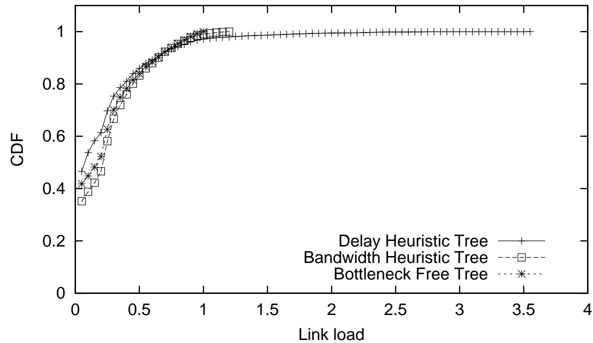

Figure 6: Link load distribution

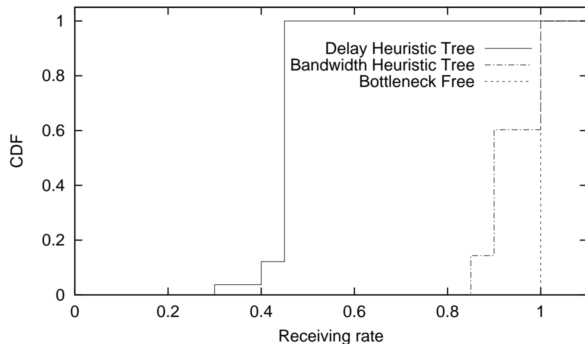

Figure 7: Receiving rate distribution

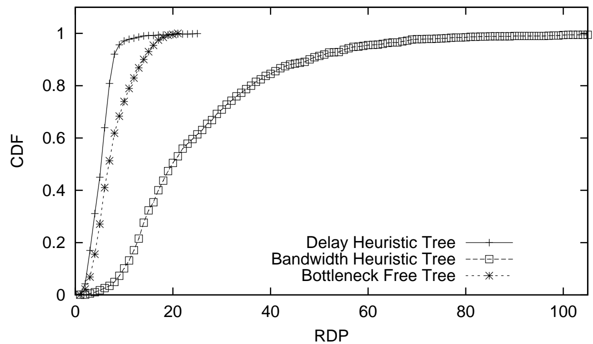

Figure 8: RDP distribution

RDP is somewhat unfair to the bandwidth heuristic; the relative delay of the bandwidth heuristic would be better if rate allocation was TCP fair rather than max-min fair. That is because TCP throughput is affected by round-trip time, and then by choosing a path with high throughput, a short path is very likely to be chosen. Nevertheless, since the 10 kB TCP throughput doesn't have strong correlation with round-trip time [8], we don't expect significant improvement with TCP fair rate allocation.

5.2 Convergence speed

The next aspect of our protocol to evaluate is its convergence speed. The protocol defines a series of actions performed during each round, and thus we use *round* as a unit to measure the convergence time. Because each round of our protocol involves shared congestion detection, which takes ten seconds to achieve high accuracy ($> 95\%$), a round should be longer than that. Also, the last case in removing an intra-path shared bottleneck, where edges are reversed along the path between edges sharing the bottleneck, requires packet transmissions as many as the length of the path. Therefore the length of a round interval must be on the order of tens of seconds.

Figure 9 shows how long it takes for a tree to stabilize when n nodes join. The convergence time increases linearly as the number of joining nodes increases, reaching 300 round when 50 hosts join. This presents an upper bound because in this scenario all the nodes first become children of the root node resulting in shared congestion on most links close to the root. The convergence time would be reduced if nodes are allowed to contact a non-root node directly to join or the root node forwards the new node to a random node, though the resulting tree might be taller.

Unlike joins, concurrent leaves can be handled relatively easily. In Figure 10, we plot the convergence time when n hosts leave the tree. Except for a few outliers, most cases take less than 20 rounds. This is because shared bottlenecks in different subtrees can be eliminated concurrently.

Another question is that how long it takes to remove a new bottleneck caused by external factors such as increased background traffic. Because of the tree structure, a bottleneck close to the root usually affects a large number of downstream nodes. There-

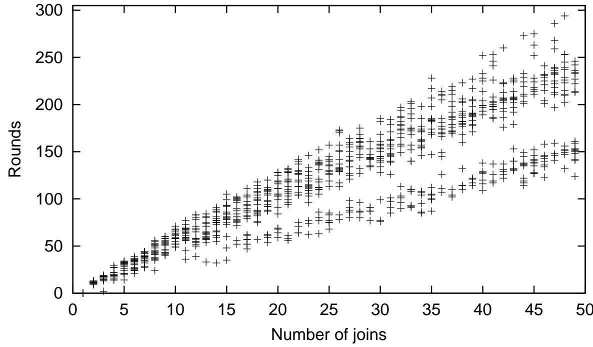

Figure 9: Convergence after nodes join

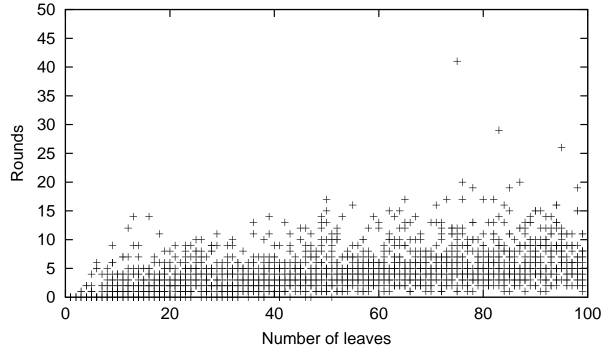

Figure 10: Convergence after nodes leave

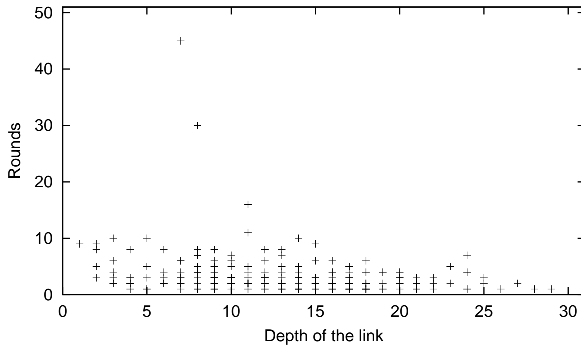

Figure 11: Convergence after available bandwidth change

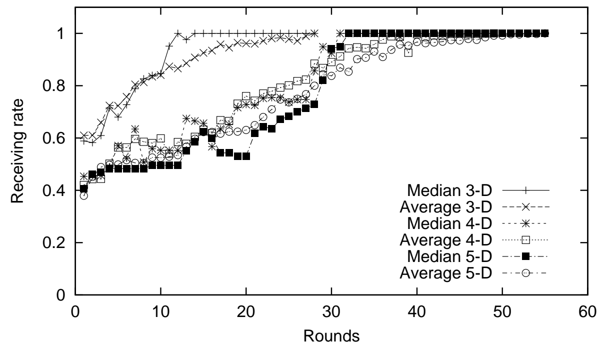

Figure 12: Receiving rate increase as a tree converges

fore, it is critical to remove the bottleneck early.

We plot in Figure 11 the time it takes to remove bottlenecks with different depth in the tree. A new shared bottleneck was created by reducing available bandwidth. As we can expect, a bottleneck near a leaf (depth larger than 25) can be removed within a couple of rounds. On the other hand, a bottleneck close to the root takes longer—up to ten rounds with a few outliers.

In real applications where available bandwidth changes dynamically and nodes leave and join at any time, the tree is more likely to keep evolving toward the moving target, rather than staying at the bottleneck-free state. Therefore, it is important to increase receiving rate in early rounds of evolution. In Figure 12, we demonstrate receiving rate changes as time elapses until the tree converges to the bottleneck-free state. The initial trees are built randomly with fixed degree. For each degree (3, 4, or 5), both median and average receiving rates are plotted. Although it takes tens of rounds to converge, most receiving rate increase is achieved within early half of the convergence time.

5.3 Effects of measurement errors

All the three schemes we evaluated in Section 5.1 depend on network measurements. In this section, we investigate the relationship between errors in measurements and tree performance in terms of receiving rate. We exclude the delay heuristic because delay measurement is relatively easy and accurate compared with bandwidth measurement and shared congestion detection. The network topology we use has 4 transit routers, 96 stub routers, and 192 end hosts. The link bandwidth is uniformly distributed in the intervals $[200, 100]$ between transit routers, $[15, 35]$ between a transit and a stub router, $[5, 10]$ between stub routers, and $[1, 3]$ between a stub router and an end host.

Figure 13 shows cumulative distributions of receiving rate for the bandwidth heuristic with different levels of errors when every receiving node joins through the root node. Although the receiving rates with higher errors are less than those with lower errors, the difference is not significant. It means that the poor performance of the bandwidth heuristic in earlier simulations resulted from the weakness of the

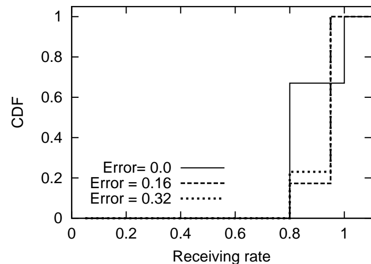

Figure 13: Effects of errors in bandwidth estimation

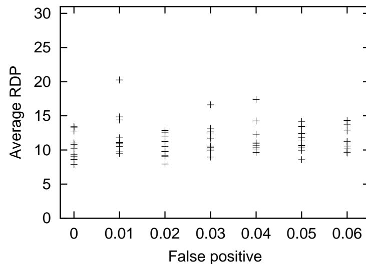

Figure 14: Effects of false positive in shared congestion

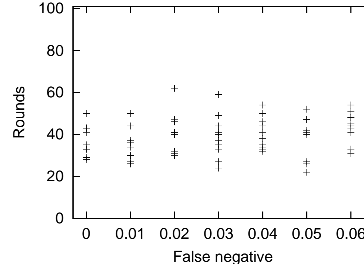

Figure 15: Effects of false negative in shared congestion

heuristic itself, not from the 20% error we introduced.

For shared congestion detection, there are two types of errors: false positive and false negative. False positive means that two paths are considered to be sharing congestion when they are not. Note that these errors are not as serious as bandwidth estimation errors because both error rates are very low in DCW if measurement period is longer than 10 seconds [7].

In the following simulations, we run the bottleneck-tree protocol with different false positive and false negative ratio, starting with a randomly built tree with degree of 3.

False positives may make the tree deeper because subtrees are moved under a deeper node even without shared congestion. The effects are demonstrated in Figure 14. However, we notice only a very slight increase of the RDP as the error rate increases up to 6%.

The effect of false negative is also negligible in the range from 0 to 6%, as shown in Figure 15; this type of errors make the convergence slower due to hidden shared bottlenecks, but only by a few rounds. Therefore, errors in shared congestion detection less then 6%, which is achievable with DCW, don't affect much the performance of our protocol.

Also note that, unlike the bandwidth heuristic, which only looks for local optimum and does not consider the source rate it should support, our protocol eventually reaches the state where all receiving rates are as high as the source rate, with or without errors.

6 Conclusion

In bandwidth-demanding multicast applications such as multimedia distribution, it is critical for a user to receive at the full source rate not to experience quality degradation. Though many heuristics to achieve high receiving rate have been proposed, they often fail to provide required receiving rate. We proposed a new tree construction algorithm that removes bottlenecks caused by the multicast session, and proved that it removes every such bottleneck. If the available bandwidth of each link is larger than the source rate, the algorithm guarantees that all receiving nodes receive at the full source rate. Simulation results show that our protocol maintains low link load and short delay penalty while providing the maximum receiving rate.

References

[1] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.

[2] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):100–110, October 2002.

[3] Yang-hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communications*, 20(8), October 2002.

[4] Khaled Harfoush, Azer Bestavros, and John Byers. Robust identification of shared losses using end-to-end unicast probe. In *Proceedings of the 8th IEEE International Conference on Network Protocols*, November 2000.

[5] 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 4th Symposium on Operating Systems Design and Implementation*, pages 197–212, October 2000.

[6] Dina Katabi, Issam Bazzi, and Xiaowei Yang. A passive approach for detecting shared bottlenecks. In *Proceedings of the 10th IEEE International Conference on Computer Communications and Networks*, October 2001.

[7] Min Sik Kim, Taekhyun Kim, YongJune Shin, Simon S. Lam, and Edward J. Powers. A wavelet-based approach to detect shared congestion. In *Proceedings of ACM SIGCOMM 2004*, August 2004.

[8] T. S. Eugene Ng, Yang-hua Chu, Sanjay G. Rao, Kunwadee Sripanidkulchai, and Hui Zhang. Measurement-based optimization techniques for bandwidth-demanding peer-to-peer systems. In *Proceedings of IEEE INFOCOM 2003*, April 2003.

[9] Dan Rubenstein, Jim Kurose, and Don Towsley. Detecting shared congestion of flows via end-to-end measurement. *Transactions on Networking*, 10(3):381–395, June 2002.

A Proof of Lemma 1

There are two cases depending on $d(r, SLeaf(v_1))$ and $d(r, v_2)$.

Case 1 $d(r, SLeaf(v_1)) \leq d(r, v_2)$

If u_2 has more than one child, the number of leaf nodes decreases by one in Line 3 . Otherwise, the condition in Line 7 is satisfied, and hence it decreases in Line 10. Note that removing (u, v) doesn't increase the number of leaf nodes because $u = Parent(Ram(u_2))$ has more than one children by the definition of *Ram*. Therefore, $D \prec D'$ holds.

Case 2 $d(r, SLeaf(v_1)) > d(r, v_2)$

In this case, t in Line 5 is not a leaf node; thus Line 6 doesn't decrease the number of leaf nodes. Since $d(r, t) = d(r, u2) + 1$, the depth of every leaf node in the subtree of v_2 increases by one when the subtree is moved by Line 6. If u_2 becomes a leaf node in Line 6, the path from $Ram(u_2)$ to u_2 is relocated under another leaf node in Line 10. Note that the depth of every node between $Ram(u_2)$ and u_2 increases after relocation. As a result, $|D|$ is equal to $|D'|$, but some elements in D are replaced with larger values, resulting in $D \prec D'$.

Therefore, $D \prec D'$ in both cases. □

B Proof of Lemma 2

If $SLeaf(v_1) \neq SLeaf(v_2)$ in Line 2, $D \prec D'$ holds by Lemma 1. Otherwise, there are the following two cases.

Case 1 $Ram(v_1) \neq Ram(v_2)$

When the condition in Line 7 holds, Line 8 doesn't increase the number of leaf nodes. If u_2 becomes a new leaf node by Line 8, then the number of leaf nodes is decreased again in Line 13. Hence the net effect is always negative. If the condition in Line 7 doesn't hold, Line 11 either maintains the same number of leaf nodes or increases by one. In the case of increase, it is decreased back in Line 13. Thus the number of leaf nodes always remain same. However, the depth of every node in the subtree rooted at v_2 increased by 1 due to the way t is chosen. Therefore, $D \prec D'$ always holds.

Case 2 $Ram(v_1) = Ram(v_2)$

The condition means that every node between v_1 and u_2, inclusively, has only one child. In other words, the path from v_1 to u_2 is just a list. Then we reverse the order of the list, and connect it upside down. Since leaf nodes are not affected by this change, we know $D = D'$. Consider C and C'. Note that (u_1, v_1) and (u_2, v_2) are sharing a bottleneck link. Suppose the bottleneck link is (α, β). Then $P_L(u_1, v_1) = P_L(u_1, \alpha) \cup \{(\alpha, \beta)\} \cup P_L(\beta, v_1)$ and $P_L(u_2, v_2) = P_L(u_2, \alpha) \cup \{(\alpha, \beta)\} \cup P_L(\beta, v_2)$. Hence, the total cost of these two edges is $\sum_{(m,n)\in P_L(u_1,\alpha)} c(m, n) + c(\alpha, \beta) + \sum_{(m,n)\in P_L(\beta,v_1)} c(m, n) + \sum_{(m,n)\in P_L(u_2,\alpha)} c(m, n) + c(\alpha, \beta) + \sum_{(m,n)\in P_L(\beta,v_2)} c(m, n)$. After Remove-Intra-Path-Shared-Bottleneck, the edges (u_1, v_1) and (u_2, v_2) are removed, and (u_1, u_2) and (v_1, v_2) are added. The cost for other links remains same because it is symmetric. Since $P_L(u_1, u_2)$ is the shortest path between the two nodes, the cost along the path is not larger than the cost of the path going through α. In other words, $\sum_{(m,n)\in P_L(u_1,\alpha)} c(m, n) + \sum_{(m,n)\in P_L(u_2,\alpha)} c(m, n) \geq \sum_{(m,n)\in P_L(u_1,u_2)} c(m, n)$. Similarly, $\sum_{(m,n)\in P_L(\beta,v_1)} c(m, n) + \sum_{(m,n)\in P_L(\beta,v_2)} c(m, n) \geq \sum_{(m,n)\in P_L(v_1,v_2)} c(m, n)$. Because $c(\alpha, \beta) > 0$, the cost after Remove-Intra-Path-Shared-Bottleneck is strictly less than the cost before.

Therefore, $D \prec D'$, or $D = D'$ and $C < C'$. □