The Dissertation Committee for Min Sik Kim
certifies that this is the approved version of the following dissertation:

Building and Maintaining Overlay Networks for Bandwidth-Demanding Applications

Committee:

Simon S. Lam, Supervisor

Michael D. Dahlin

Mohamed G. Gouda

Aloysious K. Mok

Scott Nettles

Building and Maintaining Overlay Networks for Bandwidth-Demanding Applications

by

Min Sik Kim, B.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2005

Acknowledgments

This work was supported by National Science Foundation grants CNS-0434515, ANI-0319168, and ANI-9977267, and Texas Advanced Research Program 003658-0439-2001.

Building and Maintaining Overlay Networks for Bandwidth-Demanding Applications

Publication No. ______________

Min Sik Kim, Ph.D.
The University of Texas at Austin, 2005

Supervisor: Simon S. Lam

The demands of Internet applications have grown significantly in terms of required resources and types of services. Overlay networks have emerged to accommodate such applications by implementing more services on top of IP (Internet Protocol). However, while overlay networks are successful in circumventing limitations of IP, the task of building and maintaining an overlay network is still challenging. In an overlay network, participating hosts are virtually fully-connected through the underlying Internet. However, since the quality of overlay connections varies, the performance of the overlay network is dependent on which connections are chosen to be utilized. Therefore, maintaining a "good" overlay network topology is crucial in achieving high performance.

To demonstrate how much performance gain can be achieved through topology changes, a distributed algorithm to build an overlay multicast tree

is proposed for streaming media distribution. The algorithm finds an optimal tree such that the average bandwidth of receivers is maximized under an abstract network model. However, increasing bandwidth does not necessarily lead to a better overlay topology; in overlay networks, interference between overlay connections should be taken into account. Since such interference occurs when different overlay connections pass through a congested link simultaneously, detecting congestion shared by multiple overlay connections is necessary to avoid bottlenecks.

For shared congestion detection, a novel technique called DCW (Delay Correlation with Wavelet denoising) is proposed. Previous techniques to detect shared congestion have limitations in applying to overlay networks; they assume a common source or destination node, drop-tail queueing, or a single point of congestion. However, DCW is applicable to any pair of paths on the Internet without such limitations. It employs a signal processing method, wavelet denoising, to separate queueing delay caused by network congestion from various other delay variations. The proposed technique is evaluated through both simulations and Internet experiments. They show that for paths with a common synchronization point, DCW provides faster convergence and higher accuracy while using fewer packets than previous techniques. Furthermore, DCW is robust and accurate without a synchronization point; more specifically, it can tolerate a synchronization offset of up to one second between two packet flows.

Because DCW is designed to detect shared congestion between a pair

of paths, there is a concern about scalability when it is used in a large-scale overlay network. To cluster N paths, a straightforward approach of using pairwise tests would require $O(N^2)$ time complexity. To address this issue, a scalable approach to cluster Internet paths using multidimensional indexing is presented. By storing per-path data in a multidimensional space indexed using a tree-like structure, the computational complexity of clustering is reducible to $O(N \log N)$. The indexing overhead can be further improved by reducing dimensionality of the space through the wavelet transform. Computation cost is kept low by using the same wavelet transform for both denoising in DCW and dimensionality reduction. The proposed approach is evaluated using simulations and found to be effective for large N. The tradeoff between indexing overhead and clustering accuracy is shown empirically.

As a case study, an algorithm that improves overlay multicast topology is designed. Because overlay multicast forwards data 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 overlay topology changes, a naïve approach may create bottlenecks in other parts of the network. The proposed algorithm removes all bottlenecks caused by the redundant data delivery of overlay multicast, detecting such bottlenecks using DCW. In a case where the source rate is constant and the available bandwidth of each link is not less than the rate, the algorithm guarantees that every node receives at the full source rate. Simulation results show that even in a network with a dense receiver population, the algorithm finds a tree that satisfies all the receiving nodes while other heuristic-based approaches often fail. A similar approach to finding bottlenecks and removing them through topology changes is applicable to other types of overlay networks. This research will enable bandwidth-demanding applications to build more efficient overlay networks to achieve higher throughput.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the Internet grows in scale, the demands of its applications also grow in terms of required resources and types of services. Traditional applications such as e-mail, FTP (File Transfer Protocol), and WWW (World Wide Web) consume more bandwidth than they used to, and recent multimedia applications have more stringent requirements in terms of transmission rate, packet loss, and delay. While a number of proposals including many RFCs (Requests for Comments) related to QoS (Quality of Service) and IP Multicast have been made to add support for these applications to the Internet, few of them have been actually deployed. Even those few have very limited availability, and it is unlikely to improve much in the future.

1.1 Overlay Networks

After years of unsuccessful efforts to deploy IP-level support for various application requirements, overlay networks have emerged as an alternative. In overlay networks, end hosts maintain connections among them, and implement services such as QoS and IP multicast on top of IP. While overlay networks are successful in circumventing limitations of IP, building and maintaining an overlay network is still challenging. In an overlay network, participating hosts are virtually fully-connected through the underlying Internet. However, since the quality of overlay connections varies, the performance of the overlay network is dependent on which connections are chosen to be utilized. Therefore, maintaining a “good” overlay network topology is crucial in achieving high performance.

Selecting overlay connections requires information on their characteristics provided by the underlying network. Those characteristics are estimated through network measurements. Hence, the goal of this dissertation is two fold: to develop network measurement and analysis techniques to obtain required information, and to design overlay network protocols exploiting the information to improve overlay topology for bandwidth-demanding applications.

1.2 Overlay Multicast

As a representative bandwidth-demanding applications, Chapter 2 selects multimedia streaming of Internet radio and television stations. In the past, they have been operated by companies with high-performance dedicated servers. However, the availability of broadband access and increasing computing performance of PCs have made it feasible for individuals to run their own radio stations. As a result, thousands of channels are serving multimedia on the Internet.[1]

---

[1] See Icecast (http://yp.icecast.org/) and SHOUTcast (http://www.shoutcast.com/).

These stations require one-way data transmission to a large number of receivers, to which an overlay network, more specifically overlay multicast, is an effective solution. In overlay multicast, participants form an overlay distribution tree in the application layer and perform multicasting among themselves. The main advantage is that it does not require multicast support from the underlying network. The overlay multicast tree can be constructed on top of any network that provides a unicast transport service. This dissertation takes overlay multicast as a representative example of bandwidth-demanding applications, and proposes algorithms and techniques to improve its performance.

A major challenge in overlay network design is to build an overlay topology that can provide high bandwidth.[2] In overlay multicast, data may be delivered multiple times over the same physical link because multicast forwarding is performed without support from routers. It may result in a bottleneck on the link, especially in applications demanding high bandwidth such as multimedia distribution. A straightforward way to avoid such a bottleneck is to measure the bandwidth of each overlay connection, and choose those with large bandwidth as tree edges. However, such heuristics looking for local optima may not lead to a global optimum. Chapter 2 will present a distributed algorithm that builds a tree in which the average bandwidth from the root node, computed over all receivers in the tree, is maximized [34]. The algorithm ensures that the multicast tree always heads toward a global optimum.

---

[2] For simplicity, we will use *bandwidth* and *available bandwidth* interchangeably.

Though each node behaves for its own good based on local information, the tree approaches a global optimal state as it evolves. Convergence of the tree to an optimal tree was proved using an abstract network model.

1.3 Finding Bottlenecks

Increasing bandwidth does not necessarily lead to a better overlay topology; in overlay networks, interference between overlay connections should be taken into account. Overlay networks usually consist of a large number of end hosts and unicast flows through overlay connections between them. These unicast flows have different source and destination nodes, and may interfere with each other by sharing one or more intermediate links. In overlay multicast, for example, such interference is very likely because data is often delivered multiple times by different flows over the same physical link. Then the link would become a bottleneck, which throttles the throughput of the entire subtree of downstream receivers. Locating such bottlenecks can improve overlay network topology significantly. If an overlay network is able to identify them by finding out which overlay connections are sharing them, it can change the overlay topology to avoid the bottlenecks and improve the overall throughput. There are many applications of overlay networks that would benefit, including overlay multicast, file download from multiple servers, overlay QoS routing, and exploiting path diversity.

The basic primitive required for bottleneck identification is to decide whether two paths are sharing a congested (bottleneck) link or not. There have been many techniques to detect shared congestion [24, 30, 45], but they require that the two tested paths share an endpoint, either at the source or at the sink. Thus, they cannot be used for general overlay networks. In Chapter 3, a novel technique, DCW (Delay Correlation with Wavelet denoising), without such a drawback will be presented. It is able to detect shared congestion between almost any pair of Internet paths by employing a signal processing technique—wavelet denoising [32].

1.4 Toward Internet-Wide Applications

DCW can be used to find "better" overlay connections in overlay networks. However, its requirement that the shared congestion detection should be performed for every congested pair of paths limits its application to large-scale systems; $O(N^2)$ tests are required to detect all shared congestion among N paths. There have been studies to reduce complexity by performing per-cluster tests instead of per-path tests [30, 54], but they still need $O(N^2)$ tests in the worst case, and also in the average case if each path shares links with a very small fraction of the other paths, which is common in large-scale overlay networks.

To reduce the computational complexity, Chapter 4 introduces an efficient clustering algorithm that groups paths sharing the same bottleneck into the same cluster [33]. The algorithm stores measurement data for each path into a multidimensional space, where data sets from paths sharing congestion are located closely to each other. Because the data sets are indexed using a tree-like structure, adding paths and searching neighbors in the space take sub-polynomial time. Hence, the algorithm can reduce the computational complexity of shared congestion detection for N paths from $O(N^2)$ to $O(N \log N)$, making the shared congestion detection technique more scalable.

1.5 Overlay Multicast Revisited

DCW, with multidimensional indexing, enables to identify bottlenecks in a scalable manner. Identified bottlenecks can be eliminated by replacing overlay connections passing through them with other, under-utilized connections. However, it should be done very carefully. Suppose a tree edge is cut in overlay multicast. Then 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 of the overlay topology. In the case of overlay multicast, it is possible to remove all bottlenecks shared by multiple overlay connections without incurring such oscillation. Chapter 5 will present an algorithm that always terminates, and on termination there remains no such bottleneck in the multicast tree [35]. Furthermore, since the algorithm is very careful not to increase depth of the tree unnecessarily with changing the tree topology, it maintains short relative delay, comparable to that of the tree built by a greedy algorithm that optimizes delay.

Chapter 2

Maximizing Bandwidth for Multimedia Distribution

Internet radio and television stations require significant bandwidth to support delivery of high quality audio and video streams to a large number of receivers. Overlay multicast enables them to reduce bandwidth consumption at the station's side.

Many overlay multicast systems have been proposed for different target applications. Each of them has its own way to create a distribution tree. Of the ones that try to perform tree optimization, they generally fall into one of two categories depending on which metric they emphasize in tree construction, i.e., reducing delay or increasing throughput.[1]

Consider a set of nodes (end systems) that form an overlay on the Internet. In systems with the goal of reducing delay [4, 8, 9, 41], a mesh consisting of all nodes and selected overlay connections is first constructed. Then the nodes measure Internet delays of the overlay connections, and run a routing algorithm, such as the distance vector algorithm, to find best paths from each node to others on the mesh.

---

[1] The throughput of a distribution tree is a notion we will make more precise later.

In one system with the goal of increasing throughput [28], overlay connections with high (available) bandwidth are first chosen as edges of the distribution tree. Then the system keeps trying to increase the bandwidth between each pair of nodes by modifying the tree topology. Unlike systems with the goal of reducing delay, for which the distance vector algorithm is proved to lead to an optimal state, the proposal in [28] lacks an algorithmic method to achieve an optimal solution. In another proposal [12], a centralized algorithm was presented to compute, for a given graph, a "maximal bottleneck" spanning tree rooted at a given vertex.

Since increasing throughput is more important than reducing delay in one-way multimedia delivery, it is desirable to have a distributed algorithm that finds a tree with "maximal throughput." However, this is not a straightforward task due to the difficulties described below.

The first is the result of a fundamental limitation of today's Internet, namely: there is no simple mechanism to measure the bandwidth available to a flow between two nodes. Generally, many packets need to be sent to detect the congestion status of a path as well as how much bandwidth a flow can use without adversely affecting other flows. In other words, bandwidth measurement requires a lot more traffic than delay measurement in the Internet. Therefore, in designing the distributed algorithm, we should avoid measuring the bandwidth of too many logical links. Thus, the first difficulty we encounter is how to choose logical links that need to be measured. If we choose too few, we may be unable to find an optimal tree due to insufficient information. On the other hand, if we choose too many, there would be substantial measurement overhead on the network.

Another difficulty is node failures. Because overlay multicast depends on participating nodes, which are user machines, rather than routers, it is likely that many nodes leave the multicast group during a session. Losing some nodes would definitely change the optimal tree; thus the algorithm should be designed to be adaptive, with the ability to re-compute a new optimal tree without too much additional overhead.

This chapter presents a distributed algorithm that builds a tree in which the average receiving rate, computed over all receivers in the tree, is maximized. Convergence of the tree to an optimal tree is proved under certain network model assumptions. Protocols that implement the proposed distributed algorithm are then designed to address the difficulties discussed above. In the protocols, each node measures bandwidth from at most $O(\log n)$ nodes, where n is the number of nodes in the tree. The distribution tree is continuously updated as it converges towards an optimal tree. When there is a node failure, the protocols will adapt and the distribution tree will start converging towards a new optimal tree.

The algorithm is evaluated experimentally by simulation. Simulation results show that significant bandwidth gain is obtained within a relatively short time duration. The optimal tree derived achieves an average receiving rate (per receiver) as much as 30 times that of a random tree depending on the network configuration. The simulation results also demonstrate how the average receiving rate increases as the distribution tree evolves. For a topology consisting of 51 end hosts and 100 routers, it takes about eighty seconds to get close to the maximum. Considering the usual playback time of audio and video streams, we believe this is reasonably fast.

2.1 Network Model

It is difficult to find a simple model capturing all aspects of the Internet. In building a streaming media distribution tree, however, our main concern is bandwidth. In other words, our goal is to find a tree that provides the largest (available) bandwidth we can utilize. Accordingly, the development of a network model focuses on this aspect.

Even when we limit our concern to bandwidth only, there are still many factors to be considered. Available bandwidth is determined by many parameters. In particular, the available bandwidth between two nodes is a function of the underlying Internet topology and existing traffic. Based upon the following observations, we abstract away detailed topology and traffic information in our network model.[2]

- Usually access links are bottlenecks causing congestion while backbone links are loss-free [51].
- An access link has incoming and outgoing bandwidths that do not affect

---

[2] This abstraction is needed by the theorems in Section 2.2, but not by the protocol implementation in Section 2.3.

each other.

An access link means a link that connects a host or its local area network to the network of its ISP. We use these observations to simplify our model. Since congestion occurs mainly on access links, we assume that the bandwidth available to a flow between two nodes is determined by the congestion status of the access links of the nodes. The links in between add delay, but do not limit the bandwidth of the flow. Based on these observations, we propose an abstract model.

2.1.1 Abstract model

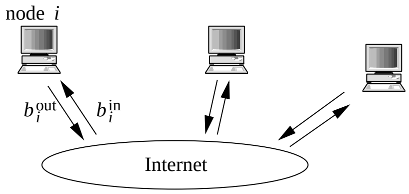


Figure 2.1: Abstract network model

A visual representation of our model is shown in Figure 2.1. A node is connected to the Internet through an access link, which has a pair of parameters: incoming and outgoing bandwidths. The incoming bandwidth of a node is the bandwidth from the ISP to the node, and the outgoing bandwidth is the bandwidth from the node to its ISP. In Figure 2.1, b_i^{in} represents the incoming bandwidth of the access link of node i, and b_i^{out} the outgoing bandwidth. A configuration of our network model is defined to be $M = (N, B)$, where N is a set of nodes and B is the set, $\{(b_i^{\text{in}}, b_i^{\text{out}}), i \in N\}$. N has $n + 1$ elements: a sender and n receivers. For convenience in presenting algorithms, we assume $N = \{0, 1, 2, \ldots, n\}$, where 0 represents the sender, and $\{1, 2, \ldots, n\}$ receivers.

Consider a distribution tree consisting of the nodes in N. The root of the tree is node 0, the sender. An intermediate node in the tree has one incoming connection from its parent and one or more outgoing connections to its children. We assume that the outgoing link bandwidth is allocated equally to its children. Let c_i denote the number of children of node i. We make the following assumption on $b_{i,j}$, the *edge bandwidth* from node i to a child node j, for every edge in the distribution tree.

Edge Bandwidth Assumption *Each node i is characterized by b_i^{in} and b_i^{out} such that if node j is a child of node i in the tree, then $b_{i,j} = \min\left(\frac{1}{c_i} b_i^{\text{out}}, b_j^{\text{in}}\right)$, where $i = 0, 1, \ldots, n$, $j = 1, 2, \ldots, n$, and $i \neq j$.*

If backbone links are not congested, then the bottleneck between two nodes should be one of the access links at either end. Therefore, we abstract away Internet topology and traffic by this assumption, and consider only access link bandwidths in our abstract model. (This abstraction is used by our theorems in section 2.2. In our protocol implementation, described in section 2.3, $b_{i,j}$ is obtained by measuring the available bandwidth from node i to node j.)

The three quantities defined above are determined by access link characteristics. We define two more quantities in the context of a distribution tree. The *incoming (receiving) rate* of node i is defined to be the minimum of edge bandwidths on the path from the root node to node i:

$$r_i^{\text{in}} = \min_{k=1,\ldots,l} b_{i_{k-1},i_k} \tag{2.1}$$

where $(0 = i_0, i_1, \ldots, i_l = i)$ is a path from the root node 0 to node i. The *outgoing (sending) rate* of node i is defined as follows.

$$r_i^{\text{out}} = \min\left(r_i^{\text{in}}, \frac{1}{c_i} b_i^{\text{out}}\right) \tag{2.2}$$

Table 2.1 summarizes the variables we have defined in this section.

Variable	Description
b_i^{in}	incoming access link bandwidth of node i
b_i^{out}	outgoing access link bandwidth of node i
$b_{i,j}$	edge bandwidth from node i to node j
r_i^{in}	incoming rate of node i
r_i^{out}	outgoing rate of node i
c_i	number of children of node i

Table 2.1: Variables

2.1.2 Fair Contribution Requirement

The centralized and distributed algorithms presented in section 2.2 are "greedy" algorithms. For these algorithms, in order for the distribution tree to converge to a global optimum, rather than a local optimum, the following condition is needed.[3]

---

[3] See proof of Theorem 2.2.1 in Section 2.2.1.

Fair Contribution Requirement *If* $b_i^{\text{in}} > b_j^{\text{in}}$*, then* $\frac{1}{c_i} b_i^{\text{out}} > \frac{1}{c_j} b_j^{\text{out}}$, for $i, j \in \{1, 2, \ldots, n\}$, $i \neq j$.

This requirement states that a node that receives more should provide more to each of its children. Suppose this requirement is not satisfied by a node that has a large incoming access link bandwidth and, relatively, a very small outgoing access link bandwidth. (This is typical of an ADSL access link.) If this node is placed high (closer to the root) in the distribution tree, selected by the greedy approach on the basis of its large incoming bandwidth without regard to its small outgoing bandwidth, then it is possible that the tree would fail to converge to the global optimum. Thus, before using one of the algorithms in section 2.2 to find a distribution tree, the values of b_i^{in} and b_i^{out}, for $i = 1, 2, \ldots, n$ should be chosen such that the Fair Contribution Requirement is satisfied.

In particular, for a node with an ADSL access link, the incoming bandwidth should be reduced to a value such that the node's incoming and outgoing bandwidth values conform to the Fair Contribution Requirement. On the other hand, if a node, say i, has a very large outgoing access link bandwidth relative to its incoming access link bandwidth, it would be desirable to choose a large value for c_i so long as the Fair Contribution Requirement is not violated.

We name this requirement "Fair Contribution" because, assuming that c_i is the same, for all i, the requirement states that a node that receives more from the system should provide more to the system. We consider this to be a basic fairness principle for peer-to-peer networks.

2.1.3 Tree evaluation

The incoming rate of each receiver is a good measure for evaluating a distribution tree, because it represents the amount of data that can be delivered from the root to the receiver per unit time. Given a network model $M = (N, B)$ and a tree consisting of the nodes in N, we can compute the incoming rate for every node except the root. A list of these rates is called a *rate vector*:

$$R = (r_1^{\text{in}}, r_2^{\text{in}}, \ldots, r_n^{\text{in}}) \,. \tag{2.3}$$

Note that each tree has an associated rate vector.

We can compare distribution trees by comparing their rate vectors. However, it is difficult to determine which vector is better. The best vector for one receiver is not necessarily the best for another. We can define a partial order as follows: For rate vectors, $R_1 = (r_1^1, r_2^1, \ldots, r_n^1)$ and $R_2 = (r_1^2, r_2^2, \ldots, r_n^2)$, $R_1 \geq R_2$ if and only if $r_i^1 \geq r_i^2$ for all i, $1 \leq i \leq n$. With the partial order, although we do not know in general which rate vector is "best," it should be clear that if there is a best vector, it must be a rate vector that is not less than any other rate vector. However, for a given network model M, there are usually more than one such "locally optimum" rate vectors. Trying to find one of these is too conservative a strategy. If we stop after finding a rate vector that is not less than any other, we may overlook another that increases a large amount of rate for one receiver by sacrificing a little for another. To take the overall rate increase into account, we will evaluate a distribution tree by its average incoming rate $\frac{1}{n}\sum_{i=1}^{n} r_i^{\text{in}}$. In the next section, we present a centralized algorithm and then a distributed algorithm to find a distribution tree that maximizes the average incoming rate of receivers.

2.2 Optimal Tree Algorithms

We define an optimal distribution tree to be a tree that maximizes the average incoming rate of a receiver. Given an abstract network model, $M = (N, B)$, we can find an optimal distribution tree by enumerating all trees. However, it is an infeasible approach even with a reasonable size N since there are exponentially many trees. We need more efficient algorithms to find an optimal tree.

In this section, we will first present a centralized algorithm and prove that it computes an optimal tree. Next we present a distributed version of the algorithm and prove that it converges to a tree that has the same rate vector as the optimal tree computed by the centralized algorithm. That is, the tree obtained by the distributed algorithm also maximizes the average incoming rate of a receiver.

2.2.1 Centralized algorithm

Figure 2.2 shows the centralized algorithm to find an optimal distribution tree. X is a set of nodes that can accommodate more children, and Y a set of nodes that are not added to the tree yet. Initially, only node 0,

```
CENTRALIZED-OPTIMAL-TREE
$T \leftarrow \varnothing$
$X \leftarrow \{0\}$
$Y \leftarrow N - \{0\}$
$r_0^{\text{in}} \leftarrow \infty$
while $Y \neq \varnothing$
    do $v \leftarrow$ a node in $X$ such that $r_v^{\text{out}} = \max_{i \in X} r_i^{\text{out}}$
       $w \leftarrow$ a node in $Y$ such that $b_w^{\text{in}} = \max_{i \in Y} b_i^{\text{in}}$
       $T \leftarrow T \cup \{(v, w)\}$
       $X \leftarrow X \cup \{w\}$
       $Y \leftarrow Y - \{w\}$
       if $|\{x | (v, x) \in T\}| = c_v$
          then $X \leftarrow X - \{v\}$
return $T$
```

Figure 2.2: Centralized algorithm

the root node, is in X, and all other nodes are in Y. In each iteration, the algorithm selects a node that can provide the highest outgoing rate in X, and a node that has the highest incoming access link bandwidth in Y. The edge connecting them is then added to the tree T. If the node selected in X cannot accept a child any more, it is deleted from X.

This algorithm is similar to the centralized algorithm in [12] in that both algorithms are based upon the greedy method [13]. However, both our abstract model and objective function for optimization are different from the ones in [12].

Theorem 2.2.1. *With Edge Bandwidth Assumption and Fair Contribution Requirement,*

CENTRALIZED-OPTIMAL-TREE *yields a tree T that maximizes the average* *incoming rate* $\frac{1}{n}\sum_{i=1}^{n} r_i^{\text{in}}$.

Proof Let T be the tree built with CENTRALIZED-OPTIMAL-TREE and $R = (r_1^{\text{in}}, r_2^{\text{in}}, \ldots, r_n^{\text{in}})$ its rate vector. Suppose that T^* is a tree that maximizes the average incoming rate and that its rate vector is $R^* = (r_1^{\text{in}^*}, r_2^{\text{in}^*}, \ldots, r_n^{\text{in}^*})$. Without loss of generality, we assume that $(1, 2, \ldots, n)$ is the order in which receiver nodes are added to the tree T by the algorithm. We will show that T^* can be transformed into T without changing the average incoming rate, which proves that T also maximizes the average incoming rate.

We use induction on the number of steps in transforming T^* into T. Let T_k denote the transformed tree after k steps. Then we prove that T_k has the following properties for all k, where $0 \leq k \leq n$.

P1. The subgraph consisting of nodes $0, 1, \ldots, k$ and edges between them in T_k is equal to the corresponding subgraph in T.

P2. The average incoming rate of T_k is equal to that of T^*.

The base case is trivial. After step 0, the transformed tree T_0 is T^* itself. Clearly, both P1 and P2 are satisfied by T_0.

Induction hypothesis: T_{k-1} satisfies both P1 and P2.

Given the hypothesis, we will show how to construct T_k that satisfies both P1 and P2.

Let the rate vector of T_{k-1} be $R' = (r_1^{\text{in}\prime}, r_2^{\text{in}\prime}, \ldots, r_n^{\text{in}\prime})$. By the induction hypothesis, $r_i^{\text{in}\prime} = r_i^{\text{in}}$ for all i, $1 \leq i \leq k-1$. The comparison of r_k^{in} and $r_k^{\text{in}\prime}$ gives two cases: (i) $r_k^{\text{in}} < r_k^{\text{in}\prime}$ and (ii) $r_k^{\text{in}} \geq r_k^{\text{in}\prime}$. We first show that (i) leads to a contradiction.

Assuming (i), let node j be the first node on the path from the root to node k in T_{k-1} that is not in $\{0, 1, 2, \ldots, k-1\}$. If $j = k$, k's parent in T_{k-1} must be in $\{0, \ldots, k-1\}$, and have an outgoing rate larger than k's parent in T to satisfy (i). This is impossible because k's parent should have the largest outgoing rate among the remaining nodes when it is selected by Line 6. If $j > k$, then $r_j^{\text{in}\prime} \geq r_k^{\text{in}\prime}$ because k is j's descendant. From this and (i), we conclude $r_j^{\text{in}\prime} > r_k^{\text{in}}$, which means j should have been chosen by Line 7 instead of k in building T. It contradicts the assumption that T is obtained by the algorithm.

Since (i) is impossible, (ii) must hold. Consider k's position in T.

(*Case 1*) If the same position in T_{k-1} is empty, then T_k satisfying P1 is obtained by moving k's subtree (a tree rooted at k) to that empty position in T_{k-1}. This move does not decrease any incoming rate for nodes in k's subtree because of (ii). Note that no tree can have a larger average incoming rate than that of T^* because T^* is an optimal tree. Since P2 in the induction hypothesis guarantees that the average incoming rates of T_{k-1} and T^* are equal, T_k cannot have a larger average incoming rate than that of T_{k-1}. Thus the average incoming rate of T_k must be equal to that of T_{k-1}, which proves that T_k satisfies P2.

(*Case 2*) If k's position in T is occupied by node l in T_{k-1}, there are two possibilities depending on whether k is l's descendant or not.

(*Case 2–1*) If k is l's descendant, T_k satisfying P1 is obtained by exchanging k and l in T_{k-1}. As we have shown in Case 1, the average incoming rate of T_k cannot exceed that of T_{k-1} due to the induction hypothesis (P2). Therefore, to prove that T_k satisfies P2, it suffices to show that the average incoming rate of T_k is larger than or equal to that of T_{k-1}.

By (ii), k's incoming rate does not decrease. Since k and its location have been selected in Lines 6 and 7 to maximize the incoming rate of the chosen node, we know the following inequality holds.

$$r_k^{\text{in}} \geq r_l^{\text{in}\prime} \tag{2.4}$$

Besides, since k is selected in Line 7, $b_k^{\text{in}} \geq b_l^{\text{in}}$ and accordingly $\frac{1}{c_k} b_k^{\text{out}} \geq \frac{1}{c_l} b_l^{\text{out}}$ by the Fair Contribution Requirement. This and Eq. 2.4 imply that $r_k^{\text{out}} \geq r_l^{\text{out}}$ by definition (Eq. 2.2). Therefore, the incoming rates of the nodes on the path from l to k in T_{k-1} do not decrease. There is also no change to the incoming rates of k's descendants in T_{k-1} because their ancestors remain same except the order. The only concern is node l.

To calculate l's new incoming rate, suppose that p and q are parents of l and k in T_{k-1}, respectively. The left tree in Figure 2.3 represents T_{k-1}, and the right T_k. The area surrounded with a dotted line is the common part of T and T_{k-1}, and contains nodes $1, 2, \ldots, k-1$.

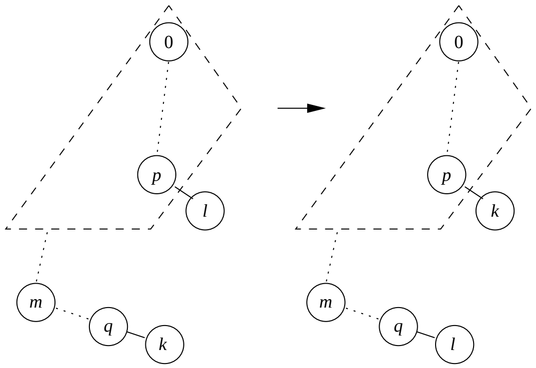


Figure 2.3: Converted trees

Then $r_p^{\text{out}} = r_p^{\text{out}\prime} \geq r_q^{\text{out}\prime}$ by the algorithm. Because the new incoming rate of l is $\min\left(r_q^{\text{out}\prime}, b_l^{\text{in}}\right)$ by the Edge Bandwidth Assumption, there are two cases depending on which value is the smaller. If $r_q^{\text{out}\prime} \geq b_l^{\text{in}}$, l's new incoming rate after exchange will be b_l^{in}, which is not less than the previous value because l cannot get more than its incoming bandwidth. If $r_q^{\text{out}\prime} < b_l^{\text{in}}$, l's new incoming rate will be $r_q^{\text{out}\prime}$, which is equal to $r_k^{\text{in}\prime}$, since $b_l^{\text{in}} \leq b_k^{\text{in}}$ by Line 7. In this case the net effect for k's and l's incoming rates is as follows.

$$\begin{aligned}
&(k\text{'s rate change}) + (l\text{'s rate change}) \\
&\quad = \left(r_k^{\text{in}} - r_k^{\text{in}\prime}\right) + \left(r_k^{\text{in}\prime} - r_l^{\text{in}\prime}\right) \geq 0 \quad \text{(by Eq. 2.4)}
\end{aligned}$$

Therefore T_k satisfies both P1 and P2 for Case 2–1.

(*Case 2–2*) If k is not l's descendant, T_k satisfying P1 is obtained by exchanging k's subtree and l's subtree in T_{k-1}. As in Case 2–1, we will prove that T_k satisfies P2 by showing that the average incoming rate of T_k is larger than or equal to that of T_{k-1}.

As we showed in Case 2–1, k's incoming rate does not decrease by exchange. Accordingly, the incoming rates of k's descendants do not decrease. We also showed that the sum of k's and l's incoming rates does not decrease. Thus, it suffices to show that the incoming rates of l's descendants do not decrease.

Before calculating the incoming rate changes of l's descendants, we claim

$$r_q^{\text{out}\prime} \geq \min\left(b_k^{\text{in}}, \frac{1}{c_k} b_k^{\text{out}}\right). \tag{2.5}$$

If not, there exists in T_{k-1} a bottleneck node m on the path from 0 to q such that $r_m^{\text{out}\prime}$ is equal to $r_q^{\text{out}\prime}$ and m's parent has a larger outgoing rate than $r_q^{\text{out}\prime}$. (m can be q itself.) It means we can achieve a higher average incoming rate by exchanging node m and node k, which contradicts that T_{k-1} maximizes the average incoming rate (P2).

We know $b_l^{\text{in}} \leq b_k^{\text{in}}$ by Line 7, and accordingly $\frac{1}{c_l} b_l^{\text{out}} \leq \frac{1}{c_k} b_k^{\text{out}}$ by the Fair Contribution Requirement. By this and Eq. 2.5, we get $r_q^{\text{out}\prime} \geq \min\left(b_l^{\text{in}}, \frac{1}{c_l} b_l^{\text{out}}\right)$. Since q provides a higher rate than l can forward to its children, the incoming rates of l's descendants do not decrease. Therefore, T_k satisfies both P1 and P2 for Case 2–2.

We have proved the inductive step for T_k. By induction, T_n has the same average incoming rate as T^*. Since $T_n = T$ by P1, T is a tree that maximizes the average bandwidth. □

```
DISTRIBUTED-OPTIMAL-TREE
  ▷ Code for node $x$ ($0 \le x \le n$).
periodic probe:
    choose a random ancestor $a \in A$
    if $\min(r_a^{\text{in}}, b_{a,x}) > r_x^{\text{in}}$
      then send ⟨probe; $x$, $r_x^{\text{in}}$, $b_x^{\text{in}}$, $b_x^{\text{out}}$, $c_x$⟩ to $a$

upon receiving ⟨probe; $y$, $r_y^{\text{in}}$, $b_y^{\text{in}}$, $b_y^{\text{out}}$, $c_y$⟩:
    if $y \notin C$ and $r_y^{\text{in}} < r_x^{\text{out}}$
      then if $|C| < c_x$ or $\min_{v \in C} r_v^{\text{out}} < \min\left(r_x^{\text{out}}, b_y^{\text{in}}, \frac{1}{c_y} b_y^{\text{out}}\right)$
              then $NewChild \leftarrow y$
              else if $\min_{v \in C} b_{x,v} > r_y^{\text{in}}$
                     then $m \leftarrow$ a random child
                          send ⟨probe; $y$, $r_y^{\text{in}}$, $b_y^{\text{in}}$, $b_y^{\text{out}}$, $c_y$⟩ to node $m$
                     else ignore the ⟨probe⟩ message
      else ignore the ⟨probe⟩ message

upon receiving ⟨child, $y$⟩ or $NewChild \neq$ NIL:
    if $NewChild \neq$ NIL
      then $y \leftarrow NewChild$
           $NewChild \leftarrow$ NIL
    $C \leftarrow C \cup \{y\}$
    if $|C| > c_x$
      then find $l$ such that $b_{x,l} = \min_{v \in C} b_{x,v}$
           $C \leftarrow C - \{l\}$
           if $y \neq l$
             then send ⟨accept; $x$⟩ to $y$
           find $i$ such that $b_{x,i} = \max_{v \in C} b_{x,v}$
           send ⟨child; $l$⟩ to node $i$
      else send ⟨accept; $x$⟩ to $y$

upon receiving ⟨accept; $y$⟩:
    send ⟨leave; $x$⟩ to node $p$
    $p \leftarrow y$

upon receiving ⟨leave; $y$⟩:
    $C \leftarrow C - \{y\}$
```

Figure 2.4: Distributed algorithm

2.2.2 Distributed algorithm

In a distributed version of our algorithm, each node maintains $O(\log n)$ states about its ancestors in the tree. The distributed algorithm is specified by the actions of each node, presented in Figure 2.4, where node x denotes some node in N. State variables maintained by node x are shown in Table 2.2. Protocol messages sent and received between nodes are shown in Table 2.3.

Variable	Description
p	parent
C	set of children
A	set of ancestors
b_x^{in}	incoming access link bandwidth of node x
b_x^{out}	outgoing access link bandwidth of node x
$b_{x,c}$	bandwidth from node x to a child c ($c \in C$)
c_x	maximum number of children
r_x^{in}	incoming rate of node x
r_x^{out}	outgoing rate of node x
$b_{a,x}$	bandwidth from an ancestor a to node x ($a \in A$)
r_a^{in}	incoming rate of an ancestor a ($a \in A$)

Table 2.2: State variables of node x

Message Sender	Meaning
$\langle \textit{probe; } i, r_i^{\text{in}}, b_i^{\text{in}}, b_i^{\text{out}}, c_y \rangle$ i or receiver's parent	The receiver is asked to be a new parent of node i.
$\langle \textit{child; } i \rangle$ receiver's parent	The receiver is asked to accept node i as a child.
$\langle \textit{accept; } i \rangle$ i	Node i has accepted the receiver as its child.
$\langle \textit{leave; } i \rangle$ i	Node i is no longer a child of the receiver.

Table 2.3: Messages of DISTRIBUTED-OPTIMAL-TREE ($0 \le i \le n$)

Initially, we assume that the state variables, p and C, in each node have been assigned values such that the nodes in N form a random tree rooted at node 0. The variables, p and C, are updated as shown in code for node x in Figure 2.4. In our abstract network model, b_i^{in}, b_i^{out}, and c_i, are known constants, for all $i \in N$, and they satisfy the Fair Contribution Requirement. Also, $b_{i,j}$, for all $i, j \in N$, are known constants, and they satisfy the Edge Bandwidth Assumption. (In our protocol implementation of the distributed algorithm, presented in Section 2.3, we describe several protocols that provide node x with up-to-date values of its variables.)

The code for node x in Figure 2.4 consists of five parts. In the first part (Lines 1–4), node x chooses an ancestor randomly. Random choice does not compromise algorithm correctness as long as the root node has nonzero probability to be chosen. It only affects how fast a tree converges to an optimal distribution tree. If the chosen ancestor can be a better parent than its current one, node x sends a $\langle$*probe*$\rangle$ message to the ancestor. The second part (Lines 5–13) describes the actions taken when a node receives a $\langle$*probe*$\rangle$ message. If the node cannot provide a higher rate than the current incoming rate of the probing node, the message is discarded. If it has room for a new child or the probing node is able to provide a higher rate to child nodes than one of the children of x, it accepts the probing node by setting *NewChild* to the probing node, which activates the third part of its code. Otherwise, the $\langle$*probe*$\rangle$ message is forwarded to a node chosen randomly among its children. The reception of a $\langle$*child*$\rangle$ message is handled in the third part (Lines 14–26). The new node is added to the children set, and the worst child (lowest edge bandwidth) is cut and forwarded to the best child. The fourth part (Lines 27–29) handles the reception of an $\langle$*accept*$\rangle$ message from a new parent, and the last part (Lines 30–31) handles the reception of a $\langle$*leave*$\rangle$ message from a child.

Theorem 2.2.2. *With Edge Bandwidth Assumption and Fair Contribution Requirement,*
DISTRIBUTED-OPTIMAL-TREE *makes the distribution tree converge to a tree that has the same rate vector as the one obtained with* CENTRALIZED-OPTIMAL-TREE*.*

Proof Let T be the tree constructed by CENTRALIZED-OPTIMAL-TREE. We first prove a stronger version of the theorem under the assumption that all incoming and outgoing bandwidths are distinct. The stronger version is that, with DISTRIBUTED-OPTIMAL-TREE, a tree rooted at node 0 converges to T. Without loss of generality, we assume that $(1, 2, \ldots, n)$ is the order in which receiver nodes are added to T by CENTRALIZED-OPTIMAL-TREE. We use induction on the node sequence $(0, 1, \ldots, n)$. The base case is trivial, because node 0 is the same for both DISTRIBUTED-OPTIMAL-TREE and CENTRALIZED-OPTIMAL-TREE.

Induction hypothesis: DISTRIBUTED-OPTIMAL-TREE has constructed a tree T' such that the tree (embedded in T') consisting of nodes $0, 1, \ldots, k-1$ and edges between them is the same as the corresponding tree embedded in T.

Given the hypothesis, we will show that T' evolves into a tree such that nodes $0, 1, \ldots, k$ satisfy the same condition as in the hypothesis. We note that none of nodes $0, 1, \ldots, k-1$ will move because their $\langle probe \rangle$ messages are discarded in Lines 6–7 of the distributed algorithm.

Consider node k in T'. If k is already at the same position in T' as it is in T, the induction step is done. Otherwise, k's incoming rate in T' must be lower than k's incoming rate in T because T is an optimal tree and all bandwidths are distinct (no tie). Eventually k sends a $\langle probe \rangle$ message to 0 because 0 clearly satisfies the condition in Line 3 if k is not at an optimal position. (Sending a $\langle probe \rangle$ message to a non-root ancestor can accelerate the convergence, without compromising this proof.) k cannot receive more in T' than it does in T because all such positions are filled out by nodes $1, 2, \ldots, k-1$. However, it keeps sending $\langle probe \rangle$ messages until it reaches k's parent in T. Since k is the best node among the remaining ones, k beats any other node in Line 7 and moves to its optimal position.

We have proved the inductive step. By induction on the node sequence $0, 1, \ldots, n$, each node moves into its optimal position, resulting in forming a tree equal to T.

If bandwidths are not distinct, we may encounter ties. Exchanging nodes with the same incoming and outgoing bandwidths, however, does not affect their incoming rates. Therefore, DISTRIBUTED-OPTIMAL-TREE makes an arbitrary tree converge to a tree with the same rate vector as T. □

2.3 Optimal Tree Protocol

We have proved that DISTRIBUTED-OPTIMAL-TREE finds an optimal tree for the abstract network model. To implement the algorithm, however, several protocols are needed to initialize state variables in each node and measure up-to-date values of these variables, namely: the Join protocol, the Edge Bandwidth Measurement protocol, the Bottleneck Discovery protocol, and the Ancestor Token protocol.

2.3.1 Joins

The distributed algorithm is assumed to begin with a tree consisting of all participating nodes, which is unrealistic. For implementation, we provide the Join protocol which specifies how a joining node finds an existing tree node to which it attaches as a child.

For streaming media distribution, we assume that each joining node knows the root (sender) address, which can be obtained through an out-of-band channel, such as WWW. When the root receives a join request from a node, say x, $x \neq 0$, the root accepts x as a child if the root has fewer children than c_0. Otherwise, the root replies to the request with the address of one of its children, say node i. Then the joining node sends a join request to i. The above procedure repeats until the joining node is accepted by some node in the tree. With this protocol, the processing overhead of a join is distributed over all nodes and the sender's load is much reduced. Note that this protocol allows a joining node to join the tree if it knows the address of any existing node in the tree. Therefore, the sender's load can be further reduced by simply announcing addresses of other tree nodes, in addition to the sender, over the out-of-band channel.

When the request of a joining node, say x, is accepted by a tree node, say y, y sends to x a range of sequence numbers indicating the part of the data stream currently available from y. Then x sends to y a chosen starting sequence number in the range, and y starts data transmission. After joining the tree, the state variables of x $(1 \leq x \leq n)$ are initialized as follows: $p = y$, $c_x = 2$, $C = \varnothing$, $A = \varnothing$, $b_x^{\mathrm{in}} = b_x^{\mathrm{out}} = \infty$, and $r_x^{\mathrm{in}} = r_x^{\mathrm{out}} = 0$. The root node has the same initial values except one: $r_0^{\mathrm{in}} = \infty$. After initialization, node x can begin executing the algorithm in Figure 2.4 to try to find its optimal position in the tree.

2.3.2 Tree information update

To run Distributed-Optimal-Tree, state variables in node x that were assumed to be up-to-date in the algorithm should be explicitly measured or calculated. We describe several more protocols and explain below how to estimate these variables.

Edge bandwidth $b_{x,c}$ The edge bandwidth from a node x to its child node c is measured with the Edge Bandwidth Measurement protocol. To avoid introducing extra traffic, this protocol measures bandwidth from actual data transmission. When the data stream is forwarded on the distribution tree from node x to node c, x transmits data packets using the congestion control mechanism of TCP.[4] In the data stream, there are marker packets, or *markers*, inserted by the root. In between two consecutive markers, 32 kB of data are transmitted. A marker has three fields: `seq_from`, `seq_to`, and `r_in`. The last field, `r_in`, is the incoming rate of the node who sends the marker; this field, updated at every node, is used by the Bottleneck Discovery protocol to be described below. `seq_from` and `seq_to` are set by the root and they do not change. They contain the sequence numbers of the data packet following this marker, and the data packet preceding the next marker.

When node c receives a marker, the time is recorded. Then node c tries to determine when it finishes receiving the 32 kB of data packets that follow this marker. The finishing time is detected either by the arrival of the next marker or a data packet whose sequence number is larger than or equal to `seq_to`. Node c calculates throughput from the amount of data received divided by the elapsed time from receiving the marker to receiving the last data packet. Node c then sends to x a protocol message containing the smaller of this throughput value and b_x^{in}. This smaller value is used as an estimate of $b_{x,c}$ at both nodes. Edge bandwidth measurements are carried out by node c for every data interval in between consecutive markers. (Note that in node x, until it has receive $b_{x,c}$ from c for the first time, c is excluded whenever node x compares its children to select one of them in the distributed algorithm.)

---

[4] Our data transport protocol does not use other features of TCP, such as reliability.

Throughput is a convenient metric for available bandwidth, used in some previous studies [22, 28]. Other available bandwidth estimation methods [19, 27] can also be used instead in our protocols. A disadvantage of using throughput to estimate bandwidth is that x should have received all of the data packets between two markers before it forwards the first marker to c. Otherwise, data transmission rate may be limited by the receiving rate of x, rather than the bandwidth between x and c. It certainly increases latency. Although we can avoid this latency by using dummy data to measure $b_{x,c}$, we let x wait to use the actual data stream because our protocols are designed for bandwidth-intensive applications.

Outgoing access link bandwidth b_x^{out} b_x^{out} is estimated as follows.

$$b_x^{\text{out}} = \begin{cases} \frac{c_x}{|C|} \sum_{c \in C} b_{x,c} & \text{if } C \neq \varnothing \\ \infty & \text{otherwise} \end{cases} \tag{2.6}$$

where C is x's set of children. When $|C| = c_x$, the above estimate is simply the total edge bandwidth and might be inaccurate if the outgoing access link is not saturated. At an intermediate node in a distribution tree, there is usually more outgoing traffic than incoming traffic because the node has more than one child. Besides, an access link with more outgoing bandwidth than incoming bandwidth is rare. Therefore outgoing links are likely to be congested and the total edge bandwidth would be a good estimate for the outgoing access link bandwidth. When $|C| < c_x$, the above formula tends to overestimate b_x^{out} and accordingly gives an advantage to x in finding its position in the tree. However, in the case that x is located higher in the tree than it should be, x

has a higher probability to get a new child. Eventually C of x becomes full and the inaccuracy is corrected.

Number of children c_x and incoming access link bandwidth b_x^{in} Initially c_x is set to 2. To satisfy Fair Contribution Requirement, b_x^{in} is assigned to be $\frac{1}{c_x} b_x^{\text{out}}$. Although this is a stronger condition than that in Fair Contribution Requirement, it is simple and easy to implement. In this case, if node x is willing to support more children without reducing its current incoming rate, it can increase c_x while not violating Fair Contribution Requirement so long as the following condition is satisfied.

$$b_x^{\text{in}} = \frac{1}{c_x} b_x^{\text{out}} > r_x^{\text{in}} \tag{2.7}$$

The reason is as follows. When x increases c_x, it should decrease b_x^{in} to the new value of $\frac{1}{c_x} b_x^{\text{out}}$ to satisfy Fair Contribution Requirement. The reduced b_x^{in} might cause the optimal position of x to be moved to another position by the algorithm. However, r_x^{in} remains unchanged, since any node y on the path from the root to x has a higher or equal incoming bandwidth, i.e. $b_y^{\text{in}} \geq b_x^{\text{in}}$. Because $\frac{1}{c_y} b_y^{\text{out}} = b_y^{\text{in}} \geq b_x^{\text{in}}$, the new incoming rate of x is limited only by its own incoming bandwidth, b_x^{in}, which by Eq. 2.7 is not smaller than the previous incoming rate of x.

Incoming rate r_x^{in} The incoming rate of node x is provided by our Bottleneck Discovery protocol as follows. As mentioned in the presentation of edge bandwidth measurements, the root node sends a marker packet periodically.

The last field of the marker, `r_in`, is set to “infinity” by the root. When a node, say i, receives the marker from its parent p, it compares `r_in` in the marker and $b_{p,i}$ measured by i. If $b_{p,i}$ is smaller, i overwrites `r_in` with $b_{p,i}$; otherwise, it is left unchanged. The updated marker is then forwarded by i to its child nodes. Thus, after the marker reaches node x and has been updated by x, `r_in` contains the minimum edge bandwidth on the path from the root to node x, which is r_x^{in}.

Outgoing rate r_x^{out} When node x has r_x^{in}, b_x^{out}, and c_x, r_x^{out} is obtained directly from Eq. 2.2.

Ancestor information A, r_a^{in}, $b_{a,x}$ State variables containing information about ancestors are used only in the first part (Lines 1–4) of DISTRIBUTED-OPTIMAL-TREE, where node x finds an ancestor, say a, to probe. The edge bandwidth $b_{a,x}$ should be known in Line 3 for x to decide if ancestor a can provide a higher rate. Since each node knows edge bandwidths from its parent to itself and from itself to its children only from the Edge Bandwidth Measurement protocol, $b_{a,x}$ needs to be measured separately. A concern is that measuring $b_{a,x}$ may overwhelm node a if many descendants ask a to perform measurement simultaneously. So, instead of letting x choose a arbitrarily, we design the Ancestor Token protocol which takes care of choosing an ancestor in Line 2 and measuring $b_{a,x}$ in Line 3 of the algorithm.

In the Ancestor Token protocol, node a sends out a token (packet) whenever a has one or more children. The token contains r_a^{in}. The token is passed to a's descendants as follows. When node a issues a token, it selects a child randomly, and passes the token to the child. When node x receives a token from a, it also passes the token to a randomly selected child if a is its parent. Otherwise, it either keeps the token with probability p, or forwards the token to a randomly selected child with probability $1 - p$. If x is a leaf node, it always keeps the token. Keeping the token means that x choose a in Line 2. While x has the token from a, it is entitled to measure $b_{a,x}$. Note that x retrieves r_a^{in} from the token, which is needed in Line 3.

The measurement procedure is similar to the one in the Edge Bandwidth Measurement protocol. Each node is expected to store in its buffer at least two consecutive marker packets and all data packets in between them. Node x sends a protocol message to a requesting measurement and data transmission to x. Then a transmits the first marker, data packets, and last marker. (Note that the markers carry r_a^{in} needed by x.) $b_{a,x}$ is estimated as in the Edge Bandwidth Measurement protocol. One difference is that the end of data transmission is detected by timeout in case the second marker is lost.

After $b_{a,x}$ has been measured or the token is lost (detected by timeout), a is ready to issue a new one. By adjusting how often tokens are issued, each node can control the amount of traffic used for bandwidth measurement from itself to descendants.

After getting r_a^{in} and $b_{a,x}$, x runs the remaining part (Lines 3–4) of the algorithm. Note that the Ancestor Token protocol removes the need for keeping information on ancestors. That is, A is no longer needed to run the algorithm, and r_a^{in} and $b_{a,x}$ are provided or measured when needed. Therefore the amount of information kept by each node is $O(c_x)$.

2.3.3 Node leaves and failures

In overlay multicast, we should pay more attention to node failures, because end systems are less reliable than routers in IP multicast. Therefore, it is critical to have address information about ancestor nodes. In our implementation, an important side effect when a node issues a token packet is propagating the node's address to descendant nodes. When a node has lost its parent, it is desirable for the node to contact its closest ancestor in the tree. We add a field called `distance` into the token packet to enable each node to construct a path from the root to itself. `distance` is initially set to 0 by the node issuing a token, and incremented by one by every node receiving it. Each node caches a list of ancestors containing their addresses and distances. Note that these are soft states to help recovery from node failures; with the Ancestor Token protocol, there is no longer any need for A in our algorithm implementation. If a node detects the loss of its parent by timeout, it sends a join message to nodes in its ancestor cache starting from the closest one. In the case of a voluntary leave, a leaving node sends its parent's address to all its children, so that they can send join messages to their grandparent.

2.3.4 Rate adaptation

In an optimal distribution tree, a node farther from the root has a lower incoming rate. Thus it may be necessary for a node to make the data stream forwarded to its child have a lower rate than the rate of the data stream it receives. A straightforward way to deal with this situation is to transcode the data stream whenever its rate should be lowered [36]. However, it may impose too much processing overhead on nodes. A better solution is to use hierarchical encoding.

Multimedia data are often encoded in layers, such that the sender provides a base layer and many enhancement layers. A receiver then subscribes to the base layer and upper layers to the extent allowed by its incoming rate. If a server makes as many layers as receivers, then every receiver can fully utilize its available bandwidth. On the other hand, with a small number of layers, a tree topology change might not lead to quality improvement if the new incoming rate of a node does not exceed the cumulative rate of the next layer. However, Yang et al. have shown that 80% of the average incoming rate can be utilized with a few (4 or 5) layers if the rates of layers are chosen carefully [52]. This indicates that available bandwidth increase is likely to improve quality for receivers when layered encoding is used.

2.4 Evaluation of the Optimal Tree Protocol

To evaluate our protocols, we run simulations using several distributions of access link bandwidths. There are various types of access links ranging from 56 kbps telephone lines to dedicated high-speed lines with bandwidth higher than 1 Mbps. Distribution of access link bandwidths also varies. In the simulations, we use the following distributions that include both slow ($< 56\,\text{kbps} \approx 0.05\,\text{Mbps}$) and fast ($\geq 5\,\text{Mbps}$) links. Similar distributions have been used in previous multicast studies [29, 52].

- A uniform distribution over the interval $[0.05, 5)$.
- A normal distribution with mean 2 and standard deviation 2.
- A bimodal distribution consisting of two normal distributions. The means are 0.05 and 2.5, and the standard deviations are 0.02 and 2, respectively. In our simulations, twenty percent of the receivers are selected from the first normal distribution.

2.4.1 How good is the optimal tree?

The first question to investigate is whether it is worthwhile to compute an optimal tree. Randomly-constructed trees are compared with optimal trees to show that an optimal tree actually increases the average incoming rate significantly.

A random tree is a tree built with a given number of nodes, whose access link bandwidths are drawn from one of the three distributions described above. An optimal tree with the same set of nodes is computed using CENTRALIZED-OPTIMAL-TREE. We plot the average incoming rates of both trees in Figure 2.5, with the number of nodes varied from 100 to 800. Each point in the

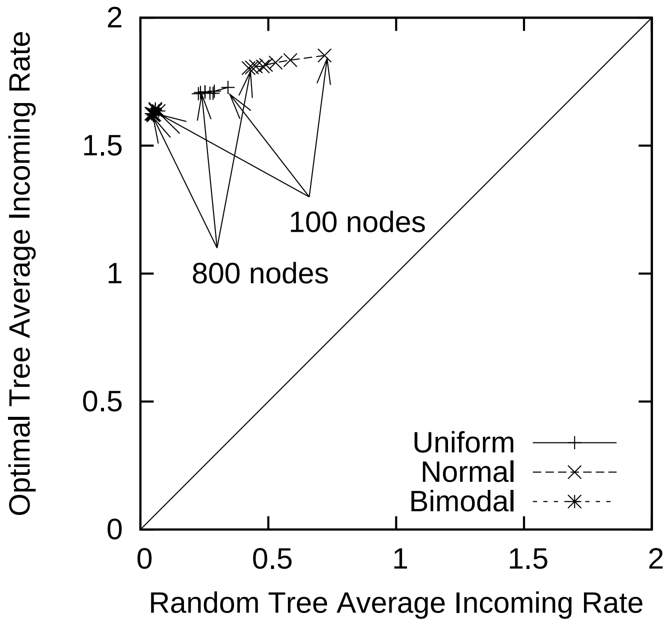


Figure 2.5: Optimal trees vs. random trees

figure represents the mean over ten simulations.

Though the actual values depend on distributions, an optimal tree has a much higher average incoming rate than a random tree. With the bimodal distribution, an optimal tree achieves a rate 30 times higher than the rate of a random tree. Note that random trees with the bimodal distribution have lower average incoming rates than those with the normal distribution, even though the mean of the bimodal distribution is larger than that of the normal distribution. The reason is that twenty percent of the nodes drawn from the bimodal distribution have very small bandwidths. It means that a small fraction of low bandwidth users can significantly slow down a large part of the tree. In this case, tree improvement is critical for the performance of bandwidth-intensive multicast applications.

Another thing to notice in Figure 2.5 is that the average incoming rate decreases (moves toward the origin) as the number of nodes increases. Such decrease is more noticeable for random trees. The corresponding decrease for optimal trees is, however, relatively small. Therefore, a tree with more nodes gets more benefit by computing an optimal tree.

2.4.2 Convergence speed

Even when the average incoming rate of an optimal tree is much higher than that of a random tree, how fast a random tree converges to an optimal tree is more important in practice. In this section, we investigate factors related to the convergence speed, especially the token keeping probability p and the number of nodes.

The convergence speed is heavily dependent on how tokens are distributed, because they give each node chances to relocate itself. Token distribution is governed by the Ancestor Token protocol with parameter p, the probability for a node to keep a token. Figure 2.6 shows how long it takes to achieve 80% of the maximum average incoming rate with different p values. Each point represents an average over ten runs. To measure elapsed time in the simulations, we use a *round* as a time unit. A round is the period during which each node issues a token once. We assume that every node issues tokens periodically. One round should be long enough for token propagation and edge bandwidth measurement. We also assume that edge bandwidth measurements are accurate in this section. The effect of inaccurate measurements will be discussed in Section 2.4.3.

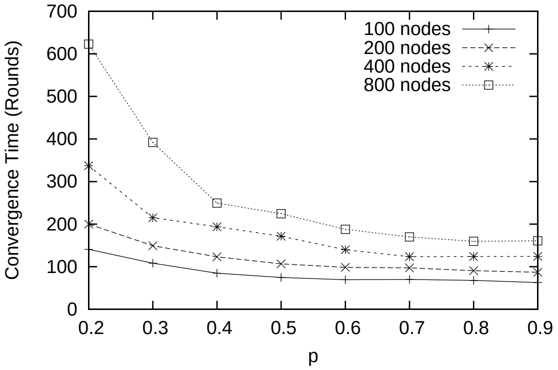


Figure 2.6: Convergence time vs. p

As shown in Figure 2.6, p should be large in order for fast increase of average incoming rate. With a small p, most tokens are used by leaf nodes, and the majority of the probe messages caused by those tokens are discarded in the second part (Lines 5–13) of DISTRIBUTED-OPTIMAL-TREE. In simulations with p larger than 0.9, the speed gain in achieving 80% of the maximum becomes negligible. So we use $p = 0.9$ in later simulations.

Figure 2.7 demonstrates how the average incoming rate changes over time when $p = 0.9$. A tree has 500 nodes, and the average incoming rate of the tree is normalized with respect to the maximum average incoming rate. The evolution of average incoming rate looks similar for all bandwidth distributions. Convergence to the maximum value takes hundreds of rounds. However, most benefits of the algorithm can be achieved within a short duration, about 50 rounds. To show that convergence time is not sensitive to the number of nodes,

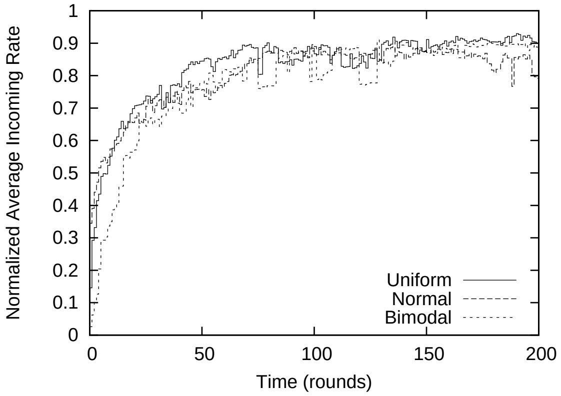


Figure 2.7: Evolution of average incoming rate

we plot the normalized average incoming rate both at the beginning and after 50 rounds in Figure 2.8. The normalized average incoming rate of each point

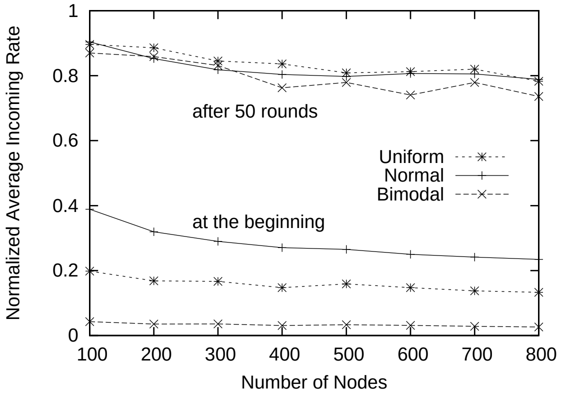


Figure 2.8: Average incoming rates at the beginning and after 50 rounds

is obtained by taking the average of 10 runs.

Again, all three trees with different bandwidth distributions show similar behaviors. Note that the average incoming rates after 50 rounds decreases as the number of nodes increases from 100 to 800. However, the decrease speed is slow. The average incoming rate for 800 nodes is 10% less than that for 100 nodes. Besides, the initial average incoming rates also decrease as the number of nodes increases; in fact, the amount of decrease is more than 30% from 100 nodes to 800 nodes. Therefore, the convergence speed is actually higher for a larger group.

These simulations show that the benefits of an optimal tree are significant and that most of them are achievable within a relatively short time.

2.4.3 Bandwidth measurement errors

We assumed that edge bandwidth measurements are accurate in the simulations presented in Section 2.4.2. In practice, however, edge bandwidth measurements may contain errors. These errors would adversely affect our protocols and lead to a sub-optimal tree.

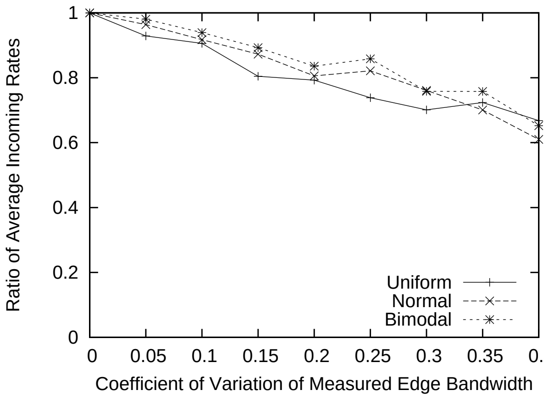


Figure 2.9: Measurement errors and achieved average incoming rate

In Figure 2.9, we investigate the impact of inaccurate bandwidth measurements on the average incoming rate. The tree has 500 nodes. Whenever a node measures an edge bandwidth, the value is drawn from the normal distribution with a mean value equal to the accurate edge bandwidth. We change the coefficient of variation (CoV) of the normal distribution to vary the degree of errors. The ratio of the average incoming rates (after 50 rounds) for trees with inaccurate and accurate measurement is plotted in Figure 2.9.

The ratio of the average incoming rates decreases linearly as CoV increases. In order to achieve a ratio higher than 0.8, CoV should not exceed 0.3. Some congestion control protocols designed to avoid sending rate fluctuations have sending rate CoV lower than 0.3 [53]; therefore, the throughput of one of these protocols would be suitable for edge bandwidth estimation in our algorithm implementation. Protocols with larger CoV like the AIMD (additive increase/multiplicative decrease) protocol of TCP can also be used by having sufficiently large measurement timescale to decrease CoV [18].

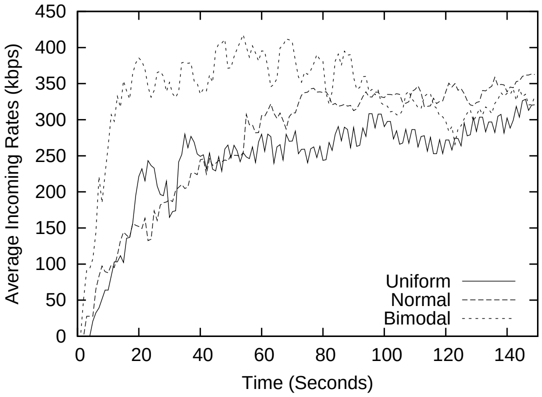


Figure 2.10: Tree improvement with measured bandwidth

Figure 2.10 shows the average incoming rate traces using AIMD throughput to estimate edge bandwidths. The simulations are run using the ns-2 simulator [17] for a topology generated with the Transit-Stub model of Georgia Tech Internetwork Topology Models (GT-ITM) [6]. The topology contains 100 routers: 75 stub routers and 25 transit routers. 51 nodes are added to the topology. One of them is the sender, and the other nodes are receivers. Access link bandwidths are drawn from the uniform, normal, and bimodal distributions described at the beginning of this section. Due to large variation in throughput measurements, the average incoming rate curves show large fluctuations. One thing to notice is that the average incoming rate is much lower than the average of the bandwidth distribution. The first reason is, as we have mentioned before, that measurement errors result in a low average incoming rate. The second is that throughput measurements with 32 kB blocks give a significantly lower value than the actual edge bandwidth, especially for those with high bandwidth; a 32 kB block may fail to saturate such a high bandwidth edge. Due to low link utilization, the measured edge bandwidth becomes much lower than the actual value, and the average incoming rate is also lower than it should be. However, the algorithm is still effective because all it needs is relative comparison among edge bandwidths.

Even with the inaccurate bandwidth measurements, the curves in Figure 2.10 look similar to those in Figure 2.7. In Figure 2.10, the average incoming rate increases for about eighty seconds and stays at a relatively stable level. Since the usual playback time of audio and video streams exceeds minutes and even hours, we believe this is acceptable.

2.5 Summary

Finding a good tree topology is critical for the performance of bandwidth-intensive multicast applications. In this chapter, a distributed algorithm has been proposed to build a tree in the application layer, and proved that it finds an optimal tree, which maximizes the average incoming rate of receivers under certain network model assumptions. Unlike other approaches using heuristics to find a local optimum, the proposed algorithm is always heading towards the global optimum. Protocols have been described to implement the algorithm on the Internet. Since a node does not keep any hard state in our implementation, it is resilient to membership changes and failures. Any node can take care of join requests in the same way as the root does, and can easily recover from leaves or failures of other nodes.

The protocol implementation has room for improvement, especially in bandwidth measurement. The AIMD throughput has large variations, caused in part by short-term unfairness of the protocol and in part by interference from other flows. The former is avoidable by adopting a more fair and smoother protocol such as TFRC [21] and TEAR [43]. Because a basic assumption of the proposed algorithm is that a node can measure the bandwidth between another node and itself, a more accurate and stable estimation technique will lead to better algorithm performance.

Chapter 3

Detecting Network Bottlenecks

In general overlay networks, maximizing bandwidth is often not feasible nor desirable. Each overlay connection needs a different amount of bandwidth, and providing more than that may result in wasting the extra bandwidth. Therefore, instead of assuming that access links are always bottlenecks, finding bottlenecks dynamically and adjusting topology will be a better, more general solution.

Knowledge of network bottlenecks can be used to improve the topology of overlay networks significantly. If an overlay network is able to determine which overlay connections are causing congestion by sharing the same link, it can change the topology to avoid such bottlenecks. Such network congestion of a link shared by multiple paths is called *shared congestion*.

Techniques for inferring shared congestion use two kinds of information from feedback: packet loss and delay. Techniques based on packet loss assume bursty packet loss [24, 45]. Thus, they work well with drop-tail queues and lossy links, but are slow and inaccurate with low loss rate or with other queueing disciplines, such as RED (Random Early Detection). Techniques based on delay [30, 45] show more robust behavior in such an environment. They are adequate for the case where two flows have a common source or a common destination. The major weakness of both kinds of techniques is that they require that the two tested paths share an endpoint, usually at the source. Thus, they cannot be used for general overlay networks.

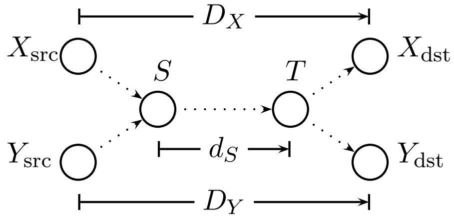
In this chapter, a novel technique (delay correlation with wavelet denoising or DCW) to detect shared congestion between two Internet paths is proposed. Like previous techniques, it is based on a simple observation: two paths sharing congested links have high correlation between their one-way delays. However, naïve correlation measurements may be inaccurate, due to random fluctuation of queueing delay and mild congestion on non-shared links. In our technique, these interfering delay variations are filtered out with *wavelet denoising*, a signal processing method for separating signal from noise.

The proposed technique is evaluated through extensive simulations and Internet experiments. When two paths have a common source, for which previous approaches can also detect shared congestion, our technique shows fast convergence with fewer packets. It takes at most 10 seconds to reach near 100% accuracy with both drop-tail and RED queues, while previous techniques often take longer or fail. We also show that our technique maintains its accuracy without a common endpoint; more specifically, it tolerates a synchronization offset between flows of up to one second, which is achievable on the Internet.

3.1 Basic Shared Congestion Detection Technique

A basic technique introduced in this section detects shared congestion using cross-correlation. This technique is effective when clocks of the nodes measuring delay are synchronized and there is only one point of congestion. With this as a basis, a general technique that tolerates a large synchronization offset and allows multiple points of congestion is developed in Section 3.2.

3.1.1 Two-path model



$$d_X = D_X - d_S, \;\; d_Y = D_Y - d_S$$

Figure 3.1: Two paths sharing links

Two paths sharing links on the Internet are illustrated in Figure 3.1. Paths X from X_{src} to X_{dst} and Y from Y_{src} to Y_{dst} are sharing links between S and T. Let the one-way delay of path X be D_X, and that of path Y be D_Y. Each of them has two components: d_S, the delay from S to T, and the remainder denoted by d_X or d_Y.

$$\begin{aligned} D_X &= d_S + d_X \\ D_Y &= d_S + d_Y \end{aligned} \tag{3.1}$$

A key observation is that the delay of a congested link has large fluctuations due to queueing delay changes, while the delay of a link with light load is relatively stable. A persistently congested link may have stable delay because its queue is persistently full. However, a measurement study shows that packet loss processes caused by congestion are better thought of as spikes rather than persistent congestion periods, and that loss runs of most spikes are shorter than 220 ms [55]. It confirms that a congested link shows large fluctuations in delay. In order to detect shared congestion, we need to determine whether such fluctuations occur between S and T.

3.1.2 Cross-correlation

Our basic technique is based on the observation that measured delays of two paths show strong correlation if the paths share one or more congested links, and little correlation if they do not share any congested links [45]. Suppose that paths X and Y in Figure 3.1 are sharing congested links between S and T, and that the other links are lightly loaded. Then D_X and D_Y will show strong similarity, since the only strongly varying component d_S is shared by both paths. On the other hand, if congestion occurs on links other than the links between S and T, D_X and D_Y become independent.

We use the cross-correlation coefficient to measure such similarity. Let $\{x_i\}$ and $\{y_i\}$ be one-way delay sequences of paths X and Y, respectively, assuming that each $\langle x_i, y_i \rangle$ pair was measured at the same time. Then their cross-correlation coefficient $XCOR_{\mathbf{xy}}$ is defined as follows.

$$XCOR_{\mathbf{xy}} = \frac{\sum_{i=1}^{m}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{m}(x_i - \bar{x})^2 \sum_{i=1}^{m}(y_i - \bar{y})^2}} \tag{3.2}$$

Note that $XCOR_{\mathbf{xy}} = 1$ if both d_X and d_Y are constant and d_S is not constant (shared congestion), and $XCOR_{\mathbf{xy}} = 0$ if d_S is constant and d_X or d_Y varies independently (no shared congestion). Of course, other network effects could make $XCOR_{\mathbf{xy}} = 1$ in the absence of shared congestion, or make $XCOR_{\mathbf{xy}} = 0$ in the presence of shared congestion. We follow earlier work by assuming that this rarely happens [45]; further Internet experimentation is required.

One of the properties of the cross-correlation coefficient is that its value is independent of any constant component of $\{x_i\}$ or $\{y_i\}$ and dominated by components with large fluctuations. It matches well with our purpose to determine if any of the shared links has large delay fluctuations. Also note that, due to this property, no clock synchronization between the source and destination nodes of paths X and Y is required in measuring one-way delay between them. However, clock skew may affect measurement. We will examine the effects of clock skew on shared congestion detection in Section 3.4.1.2. In this section, we assume that there is no clock skew.

3.1.3 Basic technique implementation

The basic technique consists of two stages: sampling and processing. In the sampling stage, X_{src} sends to X_{dst} a sequence of UDP packets with a timestamp, starting at time t_0 with its own clock. Each such UDP packet is called a *probe packet*. Probe packets are sent at a constant rate until $t_0 + T$, where T is the probe interval. On receiving a probe packet, X_{dst} calculates one-way delay and sends it, with the original timestamp, back to X_{src}. Then

X_{src} records the one-way delay together with the timestamp as a delay sample. Missing samples are linearly interpolated from neighboring samples, because if missing samples are discarded, X_i and Y_i are very likely out of synchronization from then on. The sampling stage ends when the last delay sample from X_{dst} is received (or upon timeout if the last probe or the reply to it is lost). Y_{src} and Y_{dst} also collect delay samples in the same way.

In the processing stage, the cross-correlation coefficient of two sequences of delay samples is computed as defined in Eq. 3.2. The actual procedure to gather delay sequences collected by different nodes is application-dependent. For example, in application-layer multicast, a common ancestor node of X_{src} and Y_{src} in the multicast tree can gather and process delay sequences.

3.1.4 Limitations

Applicability of the basic technique is limited because it makes two assumptions that generally do not hold for the Internet.

The first assumption is that *the two delay sequences are synchronized.* Ideally, the basic technique expects packets measuring x_i and y_i to pass through S at the same time. To achieve this, the endpoints would need precisely synchronized clocks, and to predict the delays from X_{src} and Y_{src} to S. However, one-way delays cannot be measured without network support, and network clock synchronization protocols are not accurate enough for our purposes, since they still allow errors up to half of the round-trip time between the nodes [39]. To quantify such synchronization errors, we define *synchronization offset* as the time difference between arrivals of two probe packets at S, one sent by X_{src} at time t with X_{src}'s clock and the other by Y_{src} at time t with Y_{src}'s clock. As the synchronization offset increases, the delay sequences collected by the two nodes show less and less correlation.

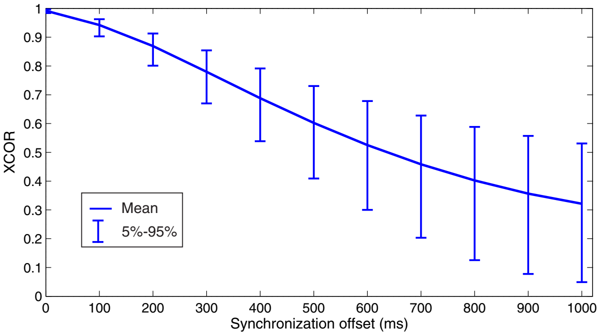


Figure 3.2: Cross-correlation coefficient between two delay sequences vs. synchronization offset

Figure 3.2 illustrates this; it plots the cross-correlation coefficient for two paths sharing a congested link as synchronization offset rises from 0 to 1 second. Each point is the mean coefficient over 300 simulations; the bars show 5th and 95th percentiles. In each simulation, two delay sample sequences were collected for 100 seconds on the topology shown in Figure 3.3 using ns-2 [17]. The bandwidth of every link was 1.5 Mb/s, and its propagation

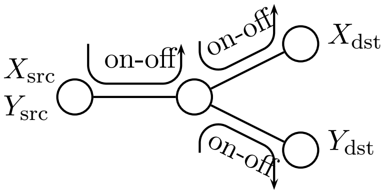


Figure 3.3: Simple topology with a common source

delay was chosen randomly between 20 ms and 30 ms for each simulation. The delay sequences represent one-way delays of two paths, from X_{src} to X_{dst} and from Y_{src} to Y_{dst}. Pareto ON-OFF CBR (constant bit rate) flows were used as background traffic, because then the congestion level could be controlled easily by changing the number of flows. The average ON and OFF times were selected uniformly between 0.2 and 3 seconds. The CBR rate was selected uniformly between 20 and 40 kb/s, and its Pareto shape parameter was 1.2. The loss rate of the shared link was about 10%; the other links did not have any loss. Without synchronization offset, the mean cross-correlation between the two delay sequences is about 0.99. However, the mean cross-correlation drops as synchronization offset increases so that a 600 ms synchronization offset results in half of the mean cross-correlation without offset.

The second assumption required by the basic technique is that *queueing delay variation on non-congested links is close to zero.* If such delay variation is not negligible, it confuses the basic technique and will give an obscure cross-correlation coefficient not close to zero or one. Then it is difficult to determine the threshold to differentiate shared congestion and independent congestion cases.

In Section 3.2, we propose wavelet denoising to enhance the basic technique. It effectively filters out delay variations in non-congested links and short-term fluctuations that confuse the basic technique, as well as negative effects of synchronization offset. With the combination of wavelet denoising and cross-correlation, our new technique can detect shared congestion for paths with a large synchronization offset and varying delays at non-congested links. It also determines quickly when there is no shared congestion.

3.1.5 Related work on shared congestion detection

Previous approaches to detect shared congestion using probe packet streams are also based on the assumption of strong correlation between packet delays or losses of two paths that share a bottleneck. Thus these approaches have the same limitation as our basic technique, i.e., two probe packet streams should be synchronized for such technique to be effective.

Rubenstein et al. proposed two techniques, one based on one-way delays and the other based on packet losses [45]. These techniques assume that the paths being probed share a common end point (either source or destination). The delay-based technique uses a Poisson process with an average interval of 40 ms to generate a sequence of delay samples. When two delay sequences are obtained for different paths, an auto-measure M_a is computed from the delays of pairs of adjacent packets in the first sequence. A cross-measure M_x is computed from a new delay sequence obtained by merging the two delay sequences. Only adjacent packet pairs with the first element in each pair from the first sequence and the second element in each pair from the second sequence are used in computing M_x. If $M_a < M_x$, it is inferred that the two paths are sharing a congested point. In their loss-based technique, M_a and M_x are conditional probabilities that a packet is lost when its following packet is lost. In their simulations, the delay-based technique was always more robust than the loss-based one.

Harfoush et al. [24, 25] proposed a loss-based technique that outperforms the loss-based technique of Rubenstein et al. In their technique, a common source sends a packet pair back-to-back at 15 Hz. The probability that only the second packet is lost is computed from packet losses. If the probability exceeds the threshold of 0.4, two it is inferred that the paths are sharing a congested point.

A different problem on detecting shared points of congestion was posed and investigated by Katabi et al [30]. They consider a large number of sources that send to a common destination. The paths form a tree rooted at the destination. Some of the tree nodes (routers) are bottlenecks such that every path goes through exactly one of the bottlenecks. They presented a passive measurement technique, based upon the entropy of packet interarrival times, to group sources into different clusters, one for each bottleneck along the way. Another approach to address a similar problem using the Markovian probing technique was presented by Younis and Fahmy [54].

3.2 Wavelet Denoising

To provide efficient solutions to network problems, various types of signal processing techniques have been employed for modeling [44] and analysis [1, 11, 26] of Internet traffic. However, they are mainly used to infer static or long-term network information from a large set of data collected over a long time span. In order to obtain dynamic information such as congestion status in a timely manner, techniques capable of on-line processing and fast response are required.

In this section, we first examine the time series of packet delay in a flow and its characteristics. Based on these characteristics, we introduce a signal processing technique—wavelet denoising [15]—that overcomes limitations of the basic cross-correlation technique in Section 3.1.4. Wavelet denoising takes the original delay time series, and generates another time series with reduced interfering fluctuations that might affect cross-correlation adversely. Finally, we discuss a procedure to find a wavelet basis that minimizes negative effects of synchronization offset.

3.2.1 Nature of delay data in time and frequency domain

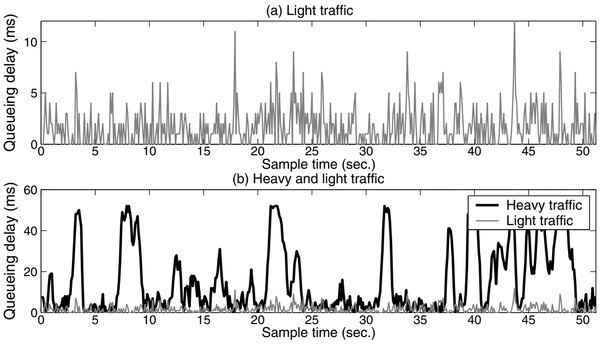


Figure 3.4: Time series of one-way delay of a single-hop path

Figure 3.4 demonstrates an example set of time series of packet delay for a link with two different congestion levels. The source and destination nodes were connected through a 1.5 Mb/s link on ns-2. The delay between them was measured using UDP packets as explained in Section 3.1.3. The time series in Figure 3.4(a) is the one-way delay under light traffic load (76 ON-OFF CBR flows, no packet loss) while the time series added in Figure 3.4(b) is the delay under heavy traffic load (92 ON-OFF CBR flows, loss rate between 2% and 10%). ON-OFF CBR flow parameter settings were identical to those described in Section 3.1.4. The 95th percentile of loss run length for heavy traffic load was about 180 ms, which is close to the Internet measurement result (220 ms) in [55].[1] Observe that the one-way delay with light traffic is a noise-like waveform with small amplitude, while the delay with heavy traffic shows an irregular pulse pattern with larger amplitude. Such pulses result from network congestion.

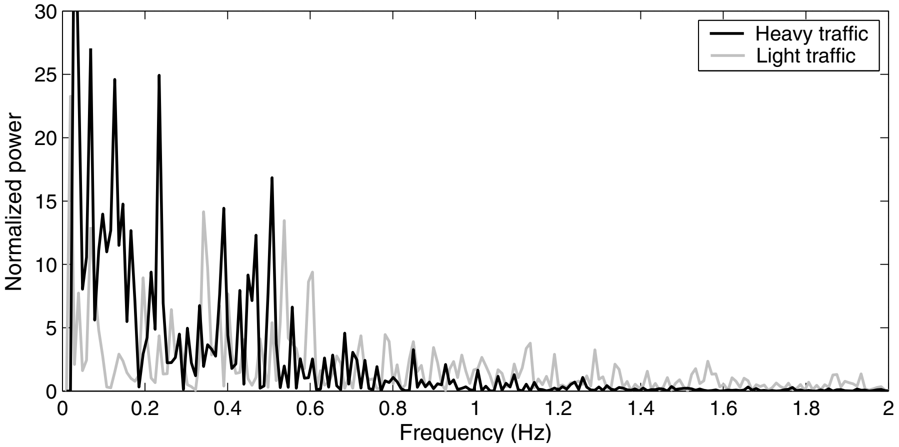


Figure 3.5: Power spectral densities of time series of delay data with light traffic and heavy traffic

The corresponding frequency domain power spectral densities of the

---

[1] Loss run lengths were measured using a Poisson packet stream with a rate of 50 Hz.

individual time series, normalized to unity area, are provided in Figure 3.5. In the frequency domain, the delay with heavy traffic shows larger amplitude at low frequencies than the delay with light traffic. Such large amplitude components at low frequencies correspond to the irregular pulses in Figure 3.4(b), caused by congestion, while others are introduced by the randomness of queue behavior, well-demonstrated in Figure 3.4(a). Therefore, for a proper assessment of network traffic under congestion via delay data, it is necessary to reduce the effects associated with random queue behavior which corrupts the traffic delays in both the time and frequency domains. In addition, if a synchronization offset is introduced in delay sampling, the measure of network traffic via delay will be less reliable.

If we are only interested in extracting the large amplitude components at low frequencies, a simple low-pass filter seems to be an intuitive solution. Low-pass filtering would smooth the delay signals, increasing cross-correlation when there is shared congestion. On the other hand, low-pass filtering may fail to diagnose non-shared congestion cases. Consider the extreme case that there is no congestion on either path. In such a case, near-zero cross-correlation is expected since the delay signals will be dominated by random queue behavior. However, simple low-pass filtering may over-smooth the signal, resulting in an inappropriately high value of cross-correlation. This is because the frequency spectrum in network delay data varies in a dynamic fashion due to the fact that network traffic changes in time. Therefore, any attempt to mitigate the interference effects should include an approach based on both time and frequency (or scale) analysis, e.g., the wavelet transform. Hence, we use wavelet denoising rather than simple filtering. We will show an empirical comparison between simple low-pass filtering and wavelet denoising in Section 3.4.2.3.

We will show that wavelet denoising is highly effective for the purpose of detecting shared congestion. A major advantage of wavelet denoising is that it preserves the dominant characteristics of one-way delay and filters out non-dominant ones in a time and scale localized manner, thus it can deal with the time-varying spectrum of network delay data. Therefore, even when there is no congestion, wavelet denoising preserves strong transients at high frequencies and thus maintains low cross-correlation between denoised signals.

There may exist other signal processing techniques that perform as well as or better than wavelet denoising. Much more investigation is needed to evaluate the large number of signal processing techniques in the literature.

3.2.2 Wavelet transform and denoising

The wavelet transform is a signal processing technique that represents a transient or non-stationary signal in terms of time and scale distribution. Due to its light computational complexity, the wavelet transform is an excellent tool for on-line data compression, analysis, and denoising.

Assume that a signal $f(t)$ is contaminated by an additive noise $n(t)$; then the measured data is $x(t) = f(t) + n(t)$. The measured time series $x(t)$ can be represented as an orthonormal expansion with wavelet basis $\psi_{i,j}(t) =$

$2^{-i/2}\psi(2^{-i}t - j)$ as follows [14]:

$$x(t) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} X_j^i \psi_{i,j}(t) \tag{3.3}$$

where the wavelet coefficients are calculated from

$$X_j^i = \int_{-\infty}^{\infty} x(t)\psi_{i,j}(t)\, dt\,. \tag{3.4}$$

Note that X_j^i is the discrete wavelet transform of $x(t)$ at scale i and at translation j, and represents how $x(t)$ is correlated with the i scaled and j translated basis function.

Two cases should be taken into account to achieve robust and reliable cross-correlation results. When there is congestion, the slowly varying congestion information (at high scale) should be extracted from the delay data, which are corrupted by synchronization offset and random queue behavior. Without congestion, strong random transients should be extracted to ensure a low correlation. Wavelet denoising is capable of selecting the desired signal while removing others in each case.

Wavelet denoising lets us build a nonlinear approximation of the signal $f(t)$ using the wavelet coefficients of the measured data $x(t)$. The wavelet coefficients for the measured data $x(t) = f(t) + n(t)$ become $X_j^i = F_j^i + N_j^i$, where $F_j^i = \int_{-\infty}^{\infty} f(t)\psi_{i,j}(t)\, dt$ and $N_j^i = \int_{-\infty}^{\infty} n(t)\psi_{i,j}(t)\, dt$. Then $\tilde{f}(t)$, an approximation of the signal $f(t)$, is obtained from the wavelet coefficients of the measured data $x(t)$ by suppressing noise with a nonlinear thresholding function, d_T. In this dissertation, we employ a soft thresholding operation on

d_T with the following definition [15]:

$$d_T(x) = \begin{cases} x - T & \text{if } x \geq T \\ x + T & \text{if } x \leq -T \\ 0 & \text{if } |x| < T\,. \end{cases} \tag{3.5}$$

The value of the threshold T is determined by the variance of the noise σ^2 [15] and the number of samples N using $T = \sigma\sqrt{2\log_e N}$, as proposed by Donoho [16]. Then the denoised signal $\tilde{f}(t)$ is obtained by applying the threshold to the wavelet coefficients X_j^i in Eq. 3.3.

$$\tilde{f}(t) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} d_T(X_j^i)\psi_{i,j}(t) \tag{3.6}$$

Soft thresholding plays a key role in the approximation of the traffic delay data under congestion. If there is shared congestion, the dominant low frequency term, which corresponds to the true traffic congestion information, will exhibit relatively large wavelet coefficient values at high scale (low frequency) so that true traffic information will remain after the thresholding operation. Meanwhile, the high frequency components, which can be assumed to be the effects of random queue behavior, will have relatively small wavelet coefficients at low scale (high frequency), and will be filtered by the thresholding operation. Soft thresholding also has the effect of smoothing the transient irregular peaks in the delay data. In the basic cross-correlation technique, randomly occurring peaks in the delay data could have a dominant deleterious effect on the cross-correlation value. Wavelet denoising smooths these irregular peaks, making the cross-correlation value more robust. On the other hand, when there is no congestion, delay variations caused by random queue behavior will have relatively large wavelet coefficient values, and thus will be preserved by soft thresholding.

3.2.3 Selection of wavelet basis

The wavelet transform provides a time and scale localized representation of a measured time series; however, the time and scale resolution of the representation depends on the selection of a wavelet basis. Hence, in order to get the most robust and reliable results from wavelet analysis including wavelet denoising, it is crucial to select the best basis function for wavelet decomposition [47]. In this dissertation, selection of a wavelet basis is confined to be within the Daubechies family of wavelets, which are widely used due to its simplicity of implementation. Other wavelets and their tradeoffs between performance and complexity need more investigation.

The correlation between a data signal and a wavelet basis is determined by time and frequency localized characteristics. Such characteristics of a data signal and wavelet basis can be represented by the time and frequency localized moments, which enable the approximation of the individual time-frequency signal elements as a Gabor logon [50]. Then the trace of the signal elements on the time-frequency plane is defined as an elliptic curve as shown in Figure 3.6. In this section, we define a metric, instantaneous SNR (signal-to-noise ratio), to indicate how closely a wavelet basis matches a data signal on the time-frequency plane. Then the metric is used to select a wavelet

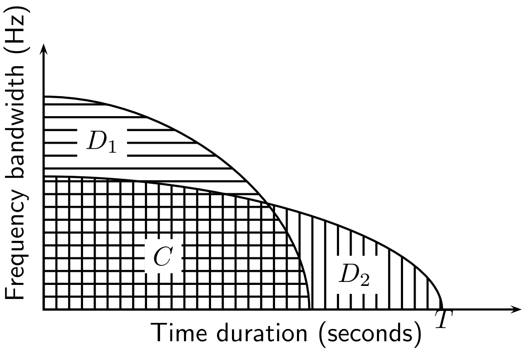


Figure 3.6: A schematic description of localized time-frequency characteristics for a data signal (horizontally hatched area) and a wavelet basis (vertically hatched area)

basis that minimizes the adverse effects of synchronization offset.

3.2.3.1 Instantaneous SNR

Figure 3.6 provides a schematic description of localized time and frequency characteristics for a data signal and wavelet basis. The quarter ellipse including C and D_1 represents the localized time-frequency characteristics of the data signal, and the quarter ellipse including C and D_2 represents those of the wavelet basis. For the two quarter ellipses to be well-matched, the size of the common area C should be large while the discrepancy $D = D_1 + D_2$ should be small. To quantify how closely the time-frequency characteristics of a data signal and wavelet basis match, we postulate a transient resolution index named "instantaneous SNR" whose dimension is dB/sec:

$$\mathrm{ISNR} = \frac{1}{T} 10 \log_{10} \frac{C}{D} \,. \tag{3.7}$$

T is the time duration of the wavelet basis [47] shown in Figure 3.6. ISNR provides a measure of similarity between the data signal and wavelet basis within the time frame of the wavelet basis function.

3.2.3.2 Minimizing adverse effects of synchronization offset

In our application, the measured data consists of two parts, slowly-varying congestion information and interference from random queue behavior and synchronization offset; such interference can be mitigated by employing a soft thresholding technique in wavelet denoising. We can further reduce the interference from synchronization offset by choosing a wavelet basis carefully.

Synchronization offset in the delay data can be interpreted as the difference of the time-shifted version of delay data and the original one. Therefore, the synchronization offset depends on the characteristics of the original data. Hence, the basis $\psi_{i,j}(t)$ should be chosen to maximize the ISNR of $f(t)$ and $\psi_{i,j}(t)$, and minimize the ISNR of $n(t)$ and $\psi_{i,j}(t)$, where $f(t)$ is the delay changes caused by network congestion and $n(t)$ is the interference caused by the synchronization offset. Therefore, it suffices to find the basis that maximizes the difference between the two ISNRs, which we call the *differential ISNR*. However, since the true $f(t)$ and $n(t)$ are not available directly, an approximation is required; we used the delay data of a congested path as $f(t)$, and the difference between the delay data and its shifted version as an approximation of $n(t) = f(t) - f(t - \Delta_{max})$, where Δ_{max} is the maximum possible synchronization error (1 second in this dissertation). More discussion on the maximum possible synchronization error is presented in Section 3.3.3.

In Figure 3.7, we plot the differential ISNR for Daubechies wavelets 2

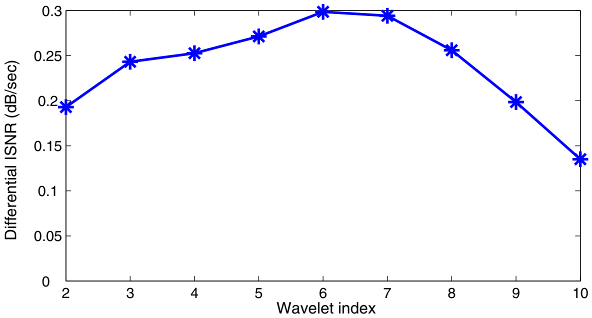


Figure 3.7: Differential ISNR between congestion signal and other noise for Daubechies wavelets

through 10. The delay sequences were obtained by repeating the simulation used to draw Figure 3.4(b) 120 times to approximate $f(t)$, and the interference $n(t)$ is directly computed from $f(t)$. Each point in Figure 3.7 is the mean value of the differential ISNR for the 120 sequences. As shown in the figure, Daubechies wavelet 6 has the highest differential ISNR, which implies that it is best matched with congestion information and least matched with the noise due to synchronization offset on the time-frequency plane. Therefore, the Daubechies wavelet 6 basis will be employed for wavelet denoising in this dissertation.

3.3 Implementation of DCW

The procedure of our wavelet-based technique is illustrated in Figure 3.8. The wavelet-based technique has the same sampling stage as described in Section 3.1.3. The sampling stage produces two sequences of delay samples,

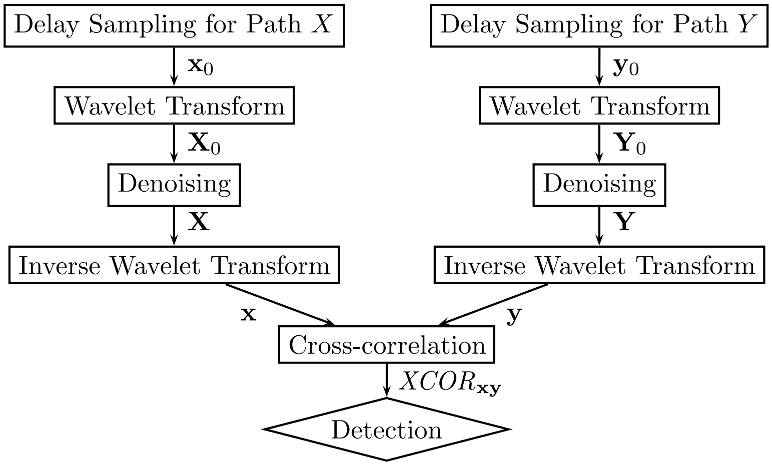


Figure 3.8: Shared congestion detection procedure

$\mathbf{x}_0$ and $\mathbf{y}_0$.[2] The processing stage uses wavelet denoising (wavelet transform, denoising, and inverse wavelet transform) to produce new, denoised sequences $\mathbf{x}$ and $\mathbf{y}$, as explained above. The cross-correlation coefficient $XCOR_{\mathbf{xy}}$ is computed from $\mathbf{x}$ and $\mathbf{y}$. (The computational overhead of these operations is very low. We found that when delay samples were collected at 10 Hz for 100 seconds for each of two paths, a machine with a 2.53 GHz Intel Pentium 4 CPU took only a few milliseconds to finish the operations.) As in the basic technique, the procedure to gather delay sequences for different paths is application-dependent and out of the scope of this dissertation.

There are three issues to discuss in implementing the wavelet-based technique: the delay sampling rate, synchronization offset between delay se-

---

[2] We use a lower-case bold letter to represent a delay sequence, and an upper-case bold letter to represent a sequence of wavelet coefficients.

quences, and the threshold for the binary decision.

3.3.1 Sampling rate

There is a trade-off in choosing the sampling rate of a delay sequence. High-rate sampling is more accurate but incurs a large overhead on the network. On the other hand, low-rate sampling has little overhead while being slow to converge. To investigate the effect of sampling rate on performance, we performed simulations with different sampling rates on the topology shown in Figure 3.3. The sequence of delay samples for each path was processed with our wavelet denoising method. To minimize effects from synchronization offset, we used a topology with a common source. The source nodes were co-located and their clocks were synchronized. A full evaluation involving synchronization offset will be presented in Section 3.4. Each link had a bandwidth of 1.5 Mb/s, and ON-OFF CBR flow parameter settings were identical to those in Section 3.1.4. To simulate shared congestion, we put 100 ON-OFF CBR flows on the shared link, and 60 on the other two links. With 60 flows, no packet loss was observed. The loss rate with 100 flows varied between 2% and 12%. For independent congestion, we put 60 ON-OFF CBR flows on the shared link, and 100 on the others.

Figure 3.9 plots the mean cross-correlation coefficient over 500 experiments for five different sampling rates. The behavior consistent over all sampling rates is that the coefficients converge either to one or to zero as more and more samples are collected. With all sampling rates except 1 Hz, the

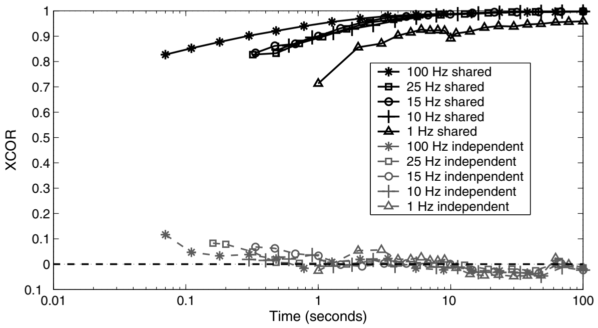


Figure 3.9: Effect of sampling rate

cross-correlation coefficient converges within 10 seconds. Their variance is also small; after 5 seconds, the interval between the 5th and 95th percentile values with shared congestion never overlaps with the corresponding interval with independent congestion for every rate but 1 Hz.

Since our technique is implemented in user space, the granularity of a timer in an operating system kernel should also be taken into account. Though recent operating systems provide clock rate of 100 Hz, older ones have only 10 Hz. From the figure, we conclude that a sampling rate of 10 Hz is fast enough in convergence and feasible to implement on most operating systems.

3.3.2 Limiting synchronization offset

There is a synchronization offset in the two sequences of delay samples collected. However, using simple techniques, the synchronization offset between any two paths on the Internet can usually be limited to 1 second. In Figure 3.1, the synchronization offset of two paths, from X_{src} to X_{dst} and from Y_{src} to Y_{dst}, is caused by (i) the difference of the delay from X_{src} to S

and the delay from Y_{src} to S, and (ii) the clock difference between X_{src} and Y_{src}. (i) is bounded by the maximum one-way delay on the network, and (ii) by half the round-trip time between X_{src} and Y_{src} since the clocks in these two nodes can be synchronized by exchanging packets. So the maximum offset is roughly the maximum round-trip time on the network. Measurement studies including one by CAIDA[3] confirm that round-trip time is less than 1 second for the vast majority of paths on the Internet.

3.3.3 Threshold for binary decision

Though cross-correlation itself is a reasonable measure of shared congestion, in situations where a binary answer is preferred, a threshold should be set. Since cross-correlation converges to one (or zero) for shared (or independent) congestion as in Figure 3.9, our technique is not sensitive to the threshold in such cases. However, because synchronization offset reduces correlation of paths sharing a congested link (as shown in Figure 3.2), it is still important to investigate an appropriate value for the threshold.

When cross-correlation coefficients of delay sample sequences with shared and independent congestion are close to each other, two types of errors may occur: false positives and false negatives. The former is the case where the technique reports shared congestion when there is no shared congested link, and the latter is the case where it reports non-shared congestion when there is one or more congested links shared by two paths. The error rate of each type

---

[3] Available at http://www.caida.org/tools/measurement/skitter/RSSAC/.

can be estimated from distributions of cross-correlation coefficients for shared and independent congestion. Then the threshold can be adjusted to minimize the total cost of errors using Bayesian testing. Our implementation assumes that the cost of false positive and the cost of false negative are equal, and minimizes the total error rate, which is the sum of the false positive ratio and the false negative ratio. Actual costs may differ from application to application.

To determine the best threshold value, we need an estimate of the synchronization offset for any two paths on the Internet. According to measurements by CAIDA, most paths from the F DNS root server to its customers have round-trip time less than 300 ms. Considering that customer hosts of a DNS root server are close to the server, we take 600 ms as the target synchronization offset to optimize the threshold for. More investigation is needed on the actual distribution of round-trip times, and the relationship between the target offset and the accuracy of the binary decision.

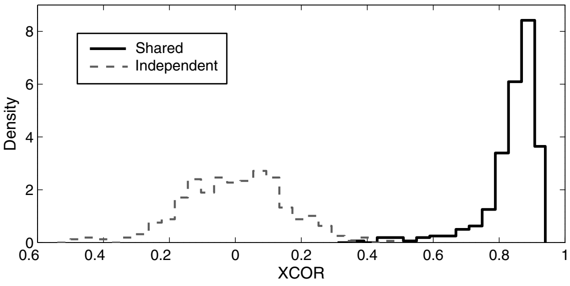


Figure 3.10: Cross-correlation coefficient distributions

Figure 3.10 shows the distributions of cross-correlation coefficients with 600 ms synchronization offset. The distributions were obtained from the same delay sequences used in Section 3.3.1. We used the delay samples collected during the first 10 seconds, with the sampling rate of 10 Hz. The left histogram represents the distribution for independent congestion, and the right one for shared congestion. If we approximate the histograms with normal distributions, they intersect when the cross-correlation coefficient (XCOR) is 0.512, which would be the threshold value that minimizes the total error rate. (The error rate is not sensitive to the choice of the threshold value as long as the threshold is between 0.3 and 0.6, because XCOR is rarely close to 0.512.) We use this value as the threshold in later experiments, unless stated otherwise. We will investigate the effect of the threshold on false positive and false negative ratio in Section 3.4.2.

3.4 Performance Evaluation

In simulations, we compare our technique against two representative techniques: a delay-based approach of Rubenstein et al. [45] and a loss-based one of Harfoush et al. [24, 25]. Below we refer to them respectively as MP (Markovian probing) and BP (Bayesian probing). See Section 3.1.5 for descriptions of both techniques.

We define *Positive Ratio* as a metric to represent the accuracy of each technique.

$$\text{Positive Ratio} = \frac{\text{\# of answers indicating shared congestion}}{\text{\# of experiments}} \tag{3.8}$$

If an experimental setup involves shared congestion, Positive Ratio should be close to one; otherwise, it should be close to zero.

We first compare our technique with MP and BP when paths share a common source node and have either shared congestion or independent congestion only. Then we investigate how they perform in more challenging environments involving paths not sharing a common source or destination and multiple points of congestion. Finally, we present initial results on the performance of our technique on the Internet.

3.4.1 Probing with a common source

Both MP and BP assume that there is a common source (or a common destination for MP). For such a topology, clocks for the two paths can be synchronized and two samples can be merged into one in chronological order. This is a critical requirement for both techniques. In fact, BP requires the stronger condition that two probe packets with different destinations must be sent back-to-back.

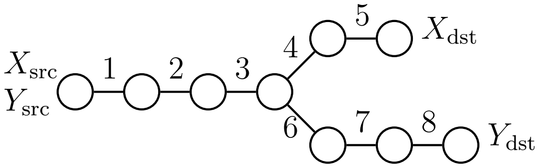


Figure 3.11: Topology with a common source

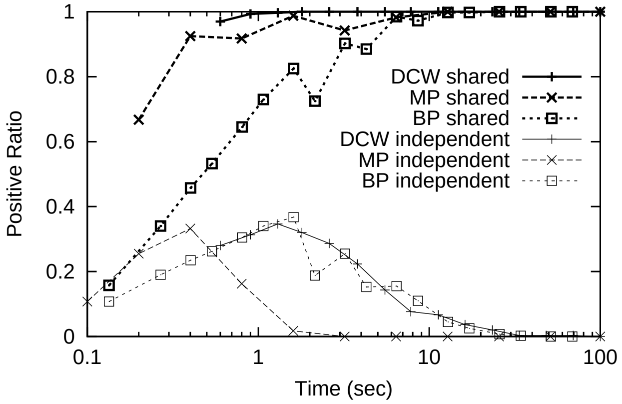
Figure 3.11 shows a network topology where two paths share a source node. Each link has a bandwidth of 1.5 Mb/s. A similar topology was used in simulations for MP [45]. We ran experiments for the following three scenarios depending on the type of background traffic.

Long-lived TCP flows A small number of long-lived TCP flows are used to cause congestion, and non-congested links are left idle. In shared congestion cases, a link is chosen from links 1 through 3, and 20 TCP flows are created to traverse the link. In independent congestion cases, links 1 through 3 are idle, and the other links have TCP flows, of which the number is chosen uniformly between 0 and 20.

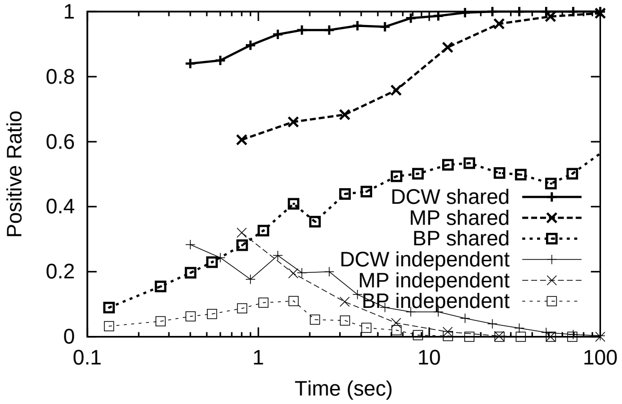
ON-OFF CBR flows A large number of ON-OFF CBR flows are used as background traffic. The congestion level is controlled with the number of such flows. For shared congestion, a link chosen from links 1 through 3 has 100 ON-OFF CBR flows. The number of ON-OFF CBR flows on the other links is chosen uniformly between 31 and 70. For independent congestion, links 1 through 3 have ON-OFF CBR flows between 31 and 70, and the other links between 61 to 100. The same parameter settings of ON-OFF CBR flows as in Section 3.1.4 are used.

Short-lived TCP flows A large number of short-lived TCP flows, created by ns-2's web traffic generator, are used as background traffic. The generated traffic consists of many "web sessions," in each of which a client node continually downloads from a server a web page containing multiple objects. For shared congestion, a link chosen from links 1 through 3 has 250 web sessions created by 25 web servers and 250 clients. The number of web sessions on the other links is chosen uniformly between 1 and 25. For independent congestion, links 1 through 3 have web sessions between 1 and 25, and the other links have web sessions between 151 and 250.

(a) Long-lived TCP background



(b) ON/OFF CBR background



(c) Short-lived TCP background

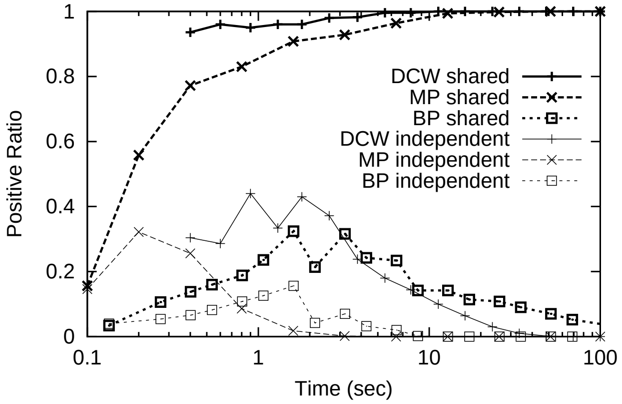


Figure 3.12: Convergence with a common source and drop-tail queues

3.4.1.1 Detection accuracy

Figure 3.12 plots Positive Ratio of each technique over 500 experiments as time progresses when links are using drop-tail queues. In the legend, DCW refers to our delay correlation technique with wavelet denoising. With long-lived TCP background, MP is fast in detecting both shared and independent congestion, while BP is relatively slow in both cases. DCW is slightly faster than MP for shared congestion, but as slow as BP for independent congestion. MP is the fastest in detecting independent congestion, in this case and many others below, because relatively small delay fluctuations on independent links can make the cross-measure M_x smaller than M_a. We will show this in Sections 3.4.2 and 3.4.3. Overall, every technique works well and reaches accuracy of over 90% within 10 seconds.

With ON-OFF CBR background traffic, however, all three techniques are slower in detecting shared congestion than with long-lived TCP background traffic. For DCW and MP, this is because non-congested links have small queueing delay fluctuations. For DCW, such fluctuations add noise to delay samples; for MP, they change the order in the merged samples and thus decrease M_x. Nevertheless, since DCW removes most noise through wavelet denoising, its degradation is not as severe as MP's. BP experiences the most notable degradation among the three; though it is the fastest for independent congestion, its Positive Ratio for shared congestion is still less than 0.6 after 100 seconds. This is because our ON-OFF CBR background flows include some with very short ON/OFF time, while all ON-OFF CBR flows in the simulations of [24] have relatively long ON time—2 seconds. BP requires the probability that both packets in a packet pair are lost to be high to detect shared congestion. A longer ON time means a queue remains full for a long time causing both packets in the pair to be dropped. However, it is less likely with short ON time. That leaves DCW to be the only technique that reaches 90% accuracy after 10 seconds with ON-OFF CBR background. Degradation of BP is even more pronounced with short-lived TCP background, because a loss period is even shorter in that scenario. As a result, BP fails to detect shared congestion. On the other hand, DCW and MP are not affected much.

In the same simulations with links using RED [32], DCW and MP showed similar performance as with drop-tail queues. However, BP did not work at all with RED queues. Its problem with RED was already pointed out using ON-OFF CBR flows [24], but the problem was more serious in our simulations because their simulation setup had a higher loss rate and smaller queues, which means that a RED queue's behavior was close to that of a drop-tail queue. Neither DCW nor MP had such a problem; they maintained performance as good as with drop-tail queues.

3.4.1.2 Effects of clock skew

The clock skew between two hosts measuring delay may affect the cross-correlation coefficient value, reducing the Positive Ratio for shared congestion. Figure 3.13 shows that the cross-correlation coefficient (XCOR) decreases as the maximum time skew during the measurement (100 seconds) increases,

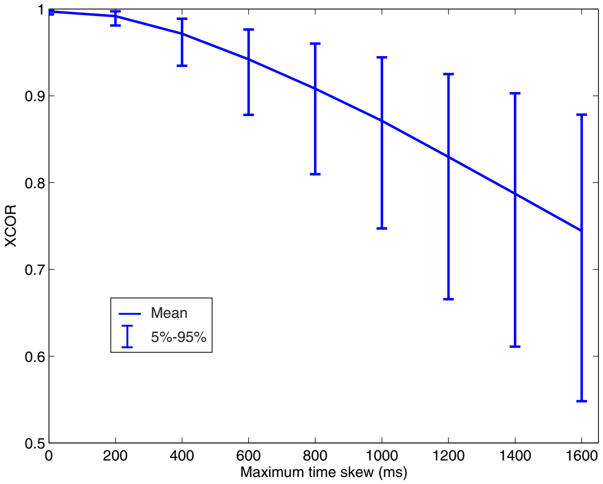


Figure 3.13: Effects of clock skew

using the simulation data for Figure 3.12(b). Nevertheless, XCOR is still close to 0.9 when the maximum time skew is 1 second. As this corresponds to gaining more than 14 minutes every day, clock skew is expected to be much less, and thus its effects on XCOR is negligible.

3.4.2 Probing with no common endpoint

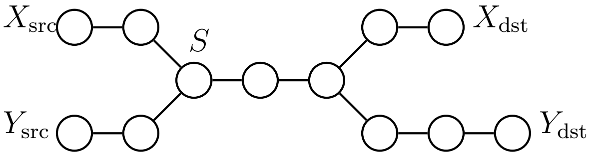


Figure 3.14: Topology with no common endpoint

The topology in Figure 3.14 is an extended version of that in Figure 3.11. The paths have different source and destination nodes. Delay samples collected at different nodes cannot be synchronized because of two reasons.

First, the clocks of node X_{src} and node Y_{src} are not synchronized. Second, the delay from X_{src} to S is different from the delay from Y_{src} to S.

3.4.2.1 Effects of synchronization offset

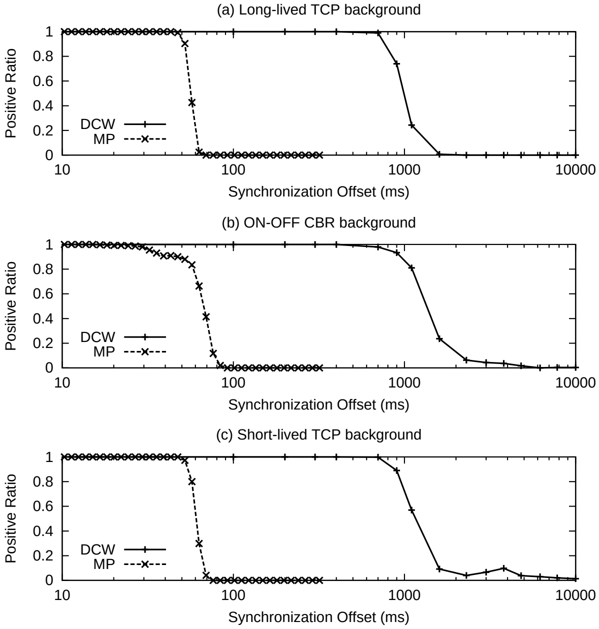


Figure 3.15: Effect of synchronization offset

To investigate the effect of synchronization offset between two paths, we plot, in Figure 3.15, the Positive Ratio for experiments with shared congestion as we increase the synchronization offset for all three types of background traffic. The original sets of delay samples were obtained from the two paths on the topology in Figure 3.11; the synchronization offset was added to the set of delay samples between Y_{src} and Y_{dst}. Only the overlapping portions were used. BP is excluded; its Positive Ratio with shared congestion is 0.2

or less even with 10 ms offset [24], due to its requirement that two packets (for different paths) be sent back-to-back. Because MP is slower than DCW in Positive Ratio convergence for shared congestion, MP may exhibit lower performance because of low accuracy if the number of delay samples is not large. Thus, detection used delay samples belonging to the first 100 seconds of the overlapping period to ensure that both MP and DCW had near-100% accuracy. The Positive Ratio drops to zero between 30 ms and 70 ms for MP, and between 1 sec and 2 sec for DCW. The sharp decrease of MP happens in the [30 ms, 70 ms] interval because the average probe rate in MP is 25 Hz, equivalent to 40 ms inter-departure time. Therefore, if the offset exceeds that value, most packets in a merged sequence are out of order, and the cross-measure M_x becomes low. Though we plot the results for drop-tail queues only, the results for RED queues are similar.

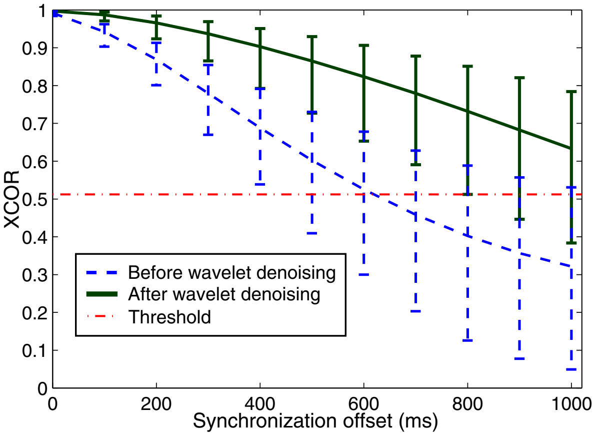


Figure 3.16: The effect of wavelet denoising on cross-correlation with synchronization offset

Next, we examine how wavelet denoising helps our technique in tolerating a large synchronization offset. The dotted curve and vertical bars crossing it in Figure 3.16 are copied from Figure 3.2, which shows the cross-correlation coefficients without wavelet denoising. We processed the data used in Figure 3.2 with our wavelet denoising, and plotted cross-correlation coefficient versus synchronization offset. The solid curve represents the mean cross-correlation coefficients, and the vertical bars indicate the 5th and 95th percentile values. Without wavelet denoising, the cross-correlation of the delay sequences decays very fast with increase of synchronization offset; with a 600 ms offset, the mean coefficient approaches the horizontal line representing the threshold (0.512). This means that the cross-correlation technique without denoising is only as good as a random decision at this point. However, the cross-correlation of the delay sequences after wavelet denoising is less sensitive to the synchronization offset, so that one can properly determine the state of congestion even with a fair amount of synchronization offset between the data. On the other hand, for independent congestion, the mean cross-correlation coefficients are not affected by wavelet denoising and are almost zero regardless of the synchronization offset.

Since synchronization offset may vary during delay measurements, we also performed an experiment with a randomized synchronization offset. For a given value of average synchronization offset m, the actual synchronization offset for a particular pair of packets in the two sequences of an experiment was chosen randomly over the interval $[0, 2m]$. The mean cross-correlation results were almost the same as those in Figure 3.16; the variances were larger due to the presence of randomized synchronization offsets.

3.4.2.2 Threshold value and false positive/negative

We use the receiver operating characteristic (ROC) curves to show the effect of the threshold value on false positive and false negative ratio in the presence of synchronization offset. ROC is a performance test methodology that measures the probability of detection P_D against the probability of false positive P_F [48]. In our application, they are defined as follows for a certain threshold value of cross-correlation T_{XCOR}.

$$\begin{aligned} P_D &= P(XCOR \geq T_{XCOR} \mid \text{shared congestion}) \\ P_F &= P(XCOR \geq T_{XCOR} \mid \text{independent congestion}) \end{aligned} \tag{3.9}$$

ROC performance can be graphically detected for all possible values of threshold T_{XCOR}; as we move along an ROC curve from the lower-left corner to the upper-right corner, the threshold varies from 1 to -1. The dashed straight line is the characteristics of the worst case, where the detection probability P_D equals the false positive probability P_F.

Figure 3.17 has two ROC curves drawn using the DCW simulation data for Figure 3.12. An offset of 600 ms was added to one of the delay sequences of each experiment. The dotted curve is an ROC curve before wavelet denoising, and the solid curve is after denoising. Since our technique converges in 10 seconds, delay samples for the first 10 seconds were used to compute the cross-correlation coefficient. With wavelet denoising, our technique shows an improved curve (higher detection probability P_D with the same false positive probability P_F) compared with the curve without denoising.

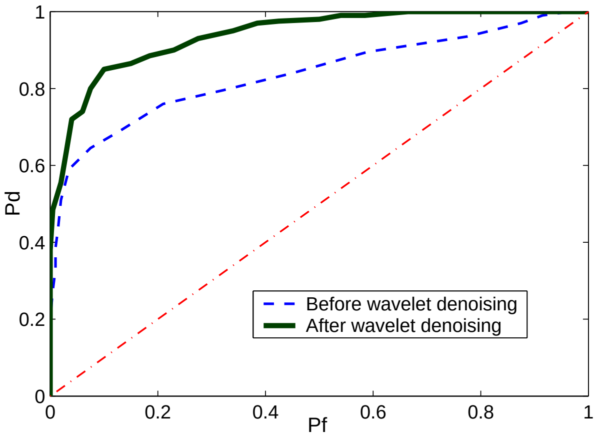


Figure 3.17: ROC with and without wavelet denoising

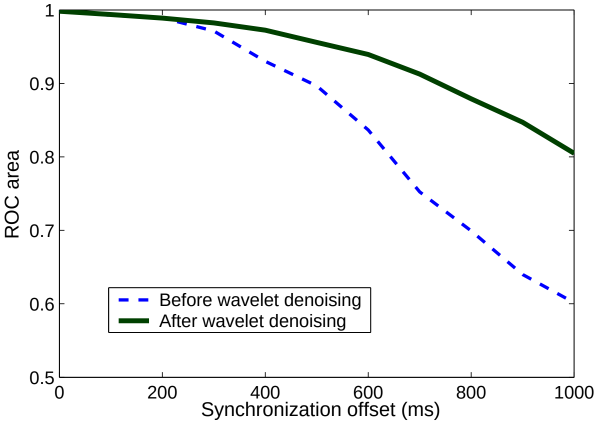


Figure 3.18: ROC performance versus synchronization offset with and without wavelet denoising

Note that the area under the curve, called the ROC area, provides a quantitative measure of performance for comparison of different curves; the area of an ideal curve is 1, while the area of a random decision maker is $\frac{1}{2}$. Figure 3.18 demonstrates the effect of wavelet denoising for different synchronization offsets using ROC area. Two curves show the ROC area with and without wavelet denoising as the synchronization offset increases. With tight synchronization, wavelet denoising makes little difference. As the offset increases, however, the basic technique curve drops to 0.6 at an offset of 1 second, becoming close to random decision. On the other hand, the technique with denoising degrades smoothly, maintaining 0.8 at the 1 second offset.

3.4.2.3 Comparison with low-pass filtering

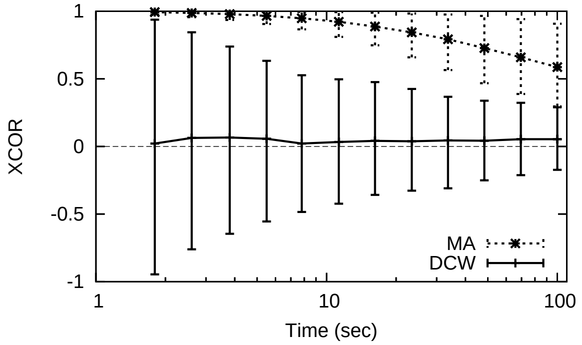


Figure 3.19: Convergence of low-pass filtering and wavelet denoising for independent congestion

When congestion occurs on shared links, wavelet denoising makes cross-correlation evaluation more robust by smoothing delay data curves. We tested a simpler mechanism to achieve this smoothing, namely a simple low-pass filter.

With suitable parameters, a moving average was able to provide similar improvement as wavelet denoising for cases with shared congestion. (We set the span of the moving average to 1.1 sec, which provides the same improvement as wavelet denoising for the experiments of Figure 3.16.) The problem with this filter appears in experiments with independent congestion. Figure 3.19 shows the convergence of the cross-correlation coefficient for the moving average (MA) and DCW when there is independent congestion in the experiment of Figure 3.12(b). Each point is the mean coefficient over 500 simulations; the bars show 5th and 95th percentiles. The mean coefficient of the moving average at 100 seconds is still 0.6, while that of DCW is almost zero from the beginning. That is, a simple low-pass filter may over-smooth transients at small scales, and thus require more delay samples to detect independent congestion. The ability of wavelet denoising to preserve strong transients at both small and large scales is critical for fast convergence in both shared and independent congestion scenarios.

3.4.3 Multiple points of congestion

So far, queueing delay variation on non-congested links was filtered out with wavelet denoising. However, if non-congested links have significant queueing delay variation, or there is more than one point of congestion, the delay variation on such links cannot be eliminated, and makes shared congestion detection more difficult. In fact, it is unclear what 'shared congestion' should mean under such conditions. Therefore, instead of deciding whether a technique detects shared congestion correctly, we investigate how the technique responds as the degree of shared congestion changes. One possible metric to represent the degree of shared congestion is how large the loss rate on shared links is compared with that on non-shared links. Hence, we define a new quantity called *shared loss rate ratio*. Let the loss rate of the shared portion of two paths be L_{shared}, and the loss rate of the non-shared portion of the first path to be L_1 and the second path L_2. Then the shared loss rate ratio is defined as follows.

$$L_s = \frac{L_{\text{shared}}}{L_{\text{shared}} + \max(L_1, L_2)} \tag{3.10}$$

If $L_{\text{shared}} > 0$ and $L_1 = L_2 = 0$, then L_s becomes 1; if $L_{\text{shared}} = 0$ and at least one of L_1 and L_2 is not zero, then L_s becomes 0. If there is no loss at all, then L_s is defined as 0, indicating no shared congestion.

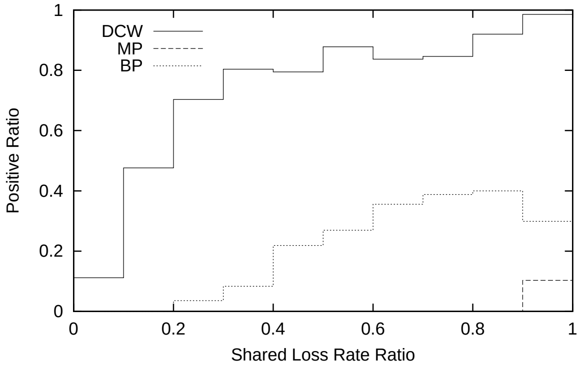


Figure 3.20: Positive Ratio with multiple points of congestion

In the following simulation, we used the topology in Figure 3.3. The number of ON-OFF CBR background flows on each link was chosen uniformly between 81 and 100, resulting in loss rate between 0 and 12%, and delay samples were collected for 100 seconds. L_s was computed from the actual loss rates of the links. 1000 experiments were classified into 10 groups depending on the interval their L_s belonged to. If L_s of an experiment is in $[0, 0.1)$ then it is in the first group, if in $[0.1, 0.2)$ then the second, and so on. If $L_s = 1$, the experiment is in the same group as those with L_s in $[0.9, 1)$. Positive Ratio (defined in Eq. 3.8) was calculated over all experiments in the same group. The results for DCW, MP, and BP are presented in Figure 3.20.

Positive Ratio of DCW is only about 0.1 when $L_s < 0.1$, but 0.8 or larger when $L_s \geq 0.3$. Thus, DCW has a cut-off at $L_s = 0.2$ differentiating shared and independent congestion. MP shows very different behavior. Positive Ratio is 0 for most intervals, and only 0.1 for the last one. Since we know that Positive Ratio of MP reaches 1 after 100 seconds if $L_s = 1$, this indicates that MP answers positively (meaning shared congestion) only when L_s is very close to 1. In other words, MP always gives a negative answer if there are multiple points of congestion, regardless of the degree of shared congestion. BP gives more and more positive answers as L_s increase, but does not have any sharp increase as such DCW has. Therefore, for those applications requiring a cut-off in shared congestion detection, DCW is preferred. However, the preferred cut-off value depends on the application. DCW can be customized for applications with different needs by adjusting its the threshold. Some applications need to determine whether two paths share *all* congested links [2], which corresponds to $L_s = 1$. In this case, MP would be a good choice.

3.4.4 Internet experiments

We applied our technique to a large-scale network, the Internet. Our preliminary Internet experiments involved six end hosts. Figure 3.21 shows their abstract topology. Note that each hop in the figure may consist of multiple physical hops. Three hosts, A_1, A_2, and A_3, are located in Austin, Texas, U.S.A. The other three hosts, K, T, and H, are located in Korea, Taiwan, and Hong Kong, respectively.

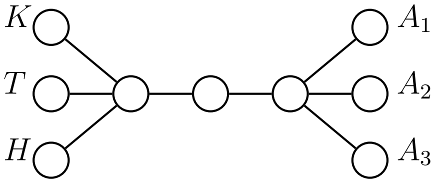


Figure 3.21: Experimental topology on the Internet

Delay samples were collected from the paths from A_1 to K and from A_2 to T between October 28 and November 2, 2003. We can reasonably conclude that there was no congested link because no probe packet was lost during measurement. In order to create a shared bottleneck, we opened 40 TCP sessions between H and A_3. The loss rate was about 5% while they were running. Since both paths experienced a similar loss rate, we conclude that the congestion occurred on one of the shared links.

The Positive Ratio for shared congestion and independent congestion (or no congestion in this case) is shown in Figure 3.22. The delay samples were collected for 15 seconds, and time was adjusted with measured clock difference

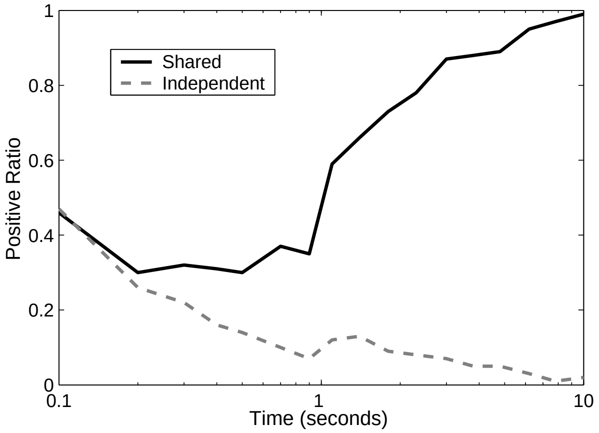


Figure 3.22: Convergence with Internet traces

between A_1 and A_2 by exchanging packets between them. Each experiment was repeated 100 times to calculate the Positive Ratio. The result resembles what we obtained through simulations. The accuracy of our technique exceeds 80% using the samples for the first 3 seconds, and reaches 98% after 8 seconds.

This experiment shows that our technique works well with real background traffic, and also that it diagnoses non-shared congestion correctly even when there is no congestion. However, the experiment was performed with limited settings, which included long-delay transpacific links and proximity of source nodes. Additional experiments are needed for more diverse environments.

3.5 Summary

Network resources are better utilized when multiple flows cooperate. However, such cooperation is feasible only when flows sharing a congested bottleneck can be identified. Previously proposed techniques had limitations, including a common endpoint and (sometimes) drop-tail routers. But they are not effective under other conditions, such as RED queueing, multiple points of congestion, or paths with different sources and destinations.

A robust shared congestion detection technique was proposed in this chapter, based on wavelet denoising and cross-correlation, namely DCW. The denoising process effectively removes noise and makes our technique more resilient to synchronization offset, which confuses other techniques. In simulations with shared congestion, DCW achieves faster convergence and broader application than previous techniques, while using fewer probe packets. Experiments on the Internet confirmed the simulation results. Many applications requiring topology construction in the application layer will benefit from the proposed delay correlation technique with wavelet denoising.

Chapter 4

Scalable Internet Path Clustering

The topology of an overlay network can be improved by identifying shared bottlenecks using DCW. Such bottlenecks in an overlay network are avoidable if the overlay network clusters connections that share the same bottleneck link, and replaces a subset of connections in each cluster with other connections not sharing the cluster's bottleneck.

However, DCW has been designed to detect shared congestion between two paths only. To cluster N paths, the straightforward approach of using pairwise tests would require $O(N^2)$ time complexity. There are other approaches proposed to reduce time complexity by performing per-cluster tests instead of per-path tests [30, 54]. In these approaches, one representative per cluster is maintained, and shared congestion detection is performed between a new path and each cluster representative to determine which cluster the new path should belong to. However, for reasons discussed in Section 4.1, these approaches are not applicable to large-scale overlay networks.

This chapter presents a scalable approach to cluster paths by shared congestion based on DCW. In this approach, measurement data are stored into a multidimensional space, where each data set collected from a network path is represented as a point. The most important characteristic of this space is that points are located closely if their corresponding network paths are sharing congestion. Due to this characteristic, finding all paths sharing congestion with a given path can be replaced with neighbor search in the space. Because points in the space are indexed using a tree-like structure, adding paths and searching neighbors takes sub-polynomial time. As a result, the computational complexity of clustering N paths can be improved to $O(N \log N)$. The indexing overhead can be further improved by reducing the dimensionality of the space through wavelet transform. Computation time is kept low because we can use the same wavelet transform for both wavelet denoising and dimensionality reduction. The tradeoff between dimensionality and clustering accuracy is investigated.

4.1 Related Work on Path Clustering

Among studies on identifying bottlenecks, FlowMate [54] and the entropy-based approach [30] have objectives that are most like the objective in this chapter.

FlowMate is based on the technique proposed by Rubenstein et al. [45] for shared congestion detection. Given two paths, a sequence of delay samples is obtained for each path. If correlation between successive packets in the first sequence is higher than correlation between the two sequences, it is inferred that the two paths are sharing a congested point. When clustering paths, FlowMate maintains a “representative” path in each cluster, and applies the shared congestion detection technique to a new path and the representative of each cluster, instead of every path in the cluster, to reduce computational complexity.

The entropy-based approach was designed to cluster flows from a large number of sources to a common destination. Thus the paths used by the flows form a tree rooted at the destination. It is assumed that each path contains exactly one bottleneck. For each path, inter-packet arrival times are measured at the destination. For each path, it calculates the average entropy for every cluster assuming the path is in that cluster. Then the path is moved to the cluster with the minimum average entropy.

Both approaches [30, 54] are inappropriate for large overlay networks for the following reasons. First, while overlay networks consist of a large number of paths with different sources and destinations, these approaches can only cluster paths that share a common end point, FlowMate at the source and the entropy-based approach at the destination. Moreover, the latter requires the amount of cross traffic to be low. More specifically, for the entropy-based approach to be robust, more than 20% of the traffic at the bottleneck should arrive at the common destination.

Second, *suppose* the N paths to be clustered share a common end point. The worst-case computational complexity of these two approaches is still $O(N^2)$ because both of them use a clustering algorithm similar to K-Means clustering [37] with a low complexity only when the number of clusters is small. In a large-scale overlay network, however, there exist many independent paths (each of which is a cluster) in addition to multi-path clusters. Therefore, the number of clusters is likely to be large.

4.2 Clustering Using a Multidimensional Space

In the proposed approach to path clustering, DCW [32] is used to detect shared congestion. In DCW, a sequence of one-way delay samples, called a *delay sequence*, is measured for each path. The DCW procedure for detecting shared congestion between two paths is shown in Figure 3.8.

As shown in Figure 3.8, the measured delay sequences, denoted by $\mathbf{x}_0$ and $\mathbf{y}_0$, are denoised using wavelet transform. Let the denoised delay sequences be $\mathbf{x}$ and $\mathbf{y}$. Then the cross-correlation coefficient $XCOR_{\mathbf{xy}}$ between them is computed. DCW decides that the two paths share congestion if $XCOR_{\mathbf{xy}}$ is larger than a specified threshold value, $XCOR_{\text{threshold}}$.

A major disadvantage of DCW when applied to a large number of paths is that the cross-correlation coefficient must be computed for every pair of paths, which does not scale well. To avoid pairwise computation, we make use of a data structure, where delay sequences are stored in such a way that given a path, all other paths sharing congestion with the path are found and retrieved easily. For this purpose, we use a multidimensional space.

Suppose that delay samples were collected from three different paths: X, Y, and Z. Then we denoise them to obtain $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$, respectively. According to the DCW procedure in Figure 3.8, we should compute $XCOR_{\mathbf{xy}}$,

$XCOR_{\mathbf{yz}}$, and $XCOR_{\mathbf{zx}}$. For better scalability, however, we instead map each denoised delay sequence to a point in a multidimensional space. A critical condition that the multidimensional space must satisfy is that points corresponding to strongly-correlated sequences should be located closely. For example,

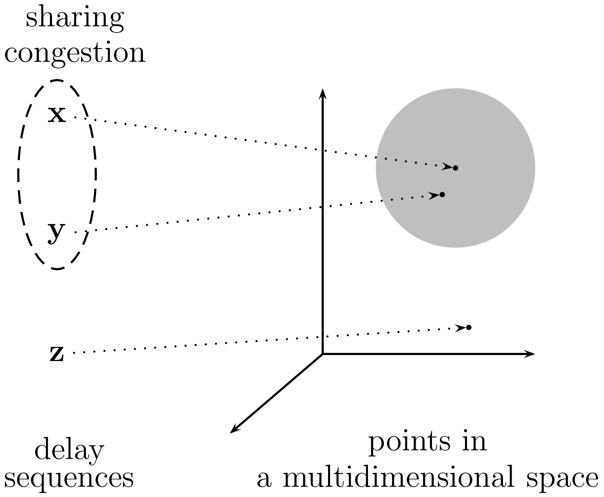


Figure 4.1: Mapping delay sequences into a multidimensional space

as shown in Figure 4.1, if $\mathbf{x}$ and $\mathbf{y}$ are strongly-correlated (because X and Y share congestion) and $\mathbf{z}$ is not, $\mathbf{x}$ and $\mathbf{y}$ should be mapped into points close to each other while $\mathbf{z}$ should be mapped to a point far from them. Then, in this space, all sequences that have strong correlation with a given sequence (in other words, all paths that share congestion with a given path) can be identified by searching neighbors of the point corresponding to the given sequence (or path). More specifically, we need a mapping such that the Euclidean distance between two points in the multidimensional space is a monotonically decreasing function of the cross-correlation coefficient between the denoised delay sequences mapped to those points. With such a mapping, all the points within the radius corresponding to $XCOR_{\text{threshold}}$ in the multidimensional space must represent delay sequences of the paths sharing congestion.

Challenges of this approach are to find a multidimensional space with the desired property and to support efficient insertion and neighbor search operations in that space.

4.2.1 Mapping delay sequences into a multidimensional space

Given two paths, X and Y, let their delay sequences after denoising be the following:

$$\begin{aligned}\mathbf{x} &= (x_1, x_2, \ldots, x_m)\\ \mathbf{y} &= (y_1, y_2, \ldots, y_m)\end{aligned}$$

Then the cross-correlation coefficient between them is computed using Eq. 3.2. The goal is to map the delay sequences $\mathbf{x}$ and $\mathbf{y}$ into two points $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$ in an m-dimensional Euclidean space so that the distance between $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$ is a monotonically decreasing function of $XCOR_{\mathbf{xy}}$. This is achieved with the following mapping.

$$\tilde{\mathbf{x}} = \frac{(x_1 - \bar{x}, x_2 - \bar{x}, \ldots, x_m - \bar{x})}{\sqrt{\sum_{i=1}^{m}(x_i - \bar{x})^2}} \tag{4.1}$$

$$\tilde{\mathbf{y}} = \frac{(y_1 - \bar{y}, y_2 - \bar{y}, \ldots, y_m - \bar{y})}{\sqrt{\sum_{i=1}^{m}(y_i - \bar{y})^2}} \tag{4.2}$$

Let $\tilde{\mathbf{x}} = (\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_m)$ and $\tilde{\mathbf{y}} = (\tilde{y}_1, \tilde{y}_2, \ldots, \tilde{y}_m)$. Then, by Eq. 4.1 and

4.2, the distance $D_{\tilde{\mathbf{x}}\tilde{\mathbf{y}}}$ between $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$ is derived as follows.

$$\begin{aligned} D_{\tilde{\mathbf{x}}\tilde{\mathbf{y}}} &= \sqrt{\sum_{i=1}^{m}(\tilde{x}_i - \tilde{y}_i)^2} \\ &= \sqrt{\sum_{i=1}^{m}\tilde{x}_i^2 - 2\sum_{i=1}^{m}\tilde{x}_i\tilde{y}_i + \sum_{i=1}^{m}\tilde{y}_i^2} \\ &= \sqrt{1 - \frac{2\sum_{i=1}^{m}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{m}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{m}(y_i - \bar{y})^2}} + 1} \end{aligned}$$

This is simplified using Eq. 3.2 to be

$$D_{\tilde{\mathbf{x}}\tilde{\mathbf{y}}} = \sqrt{2\left(1 - XCOR_{\mathbf{xy}}\right)}. \tag{4.3}$$

Therefore, given two delay sequences $\mathbf{x}$ and $\mathbf{y}$, the distance between their mappings $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$ is a monotonically decreasing function of the cross-correlation coefficient between $\mathbf{x}$ and $\mathbf{y}$.

The paths sharing congestion (or having the cross-correlation coefficient greater than $XCOR_{\text{threshold}}$) with a given path can be found by searching for neighbors of the path within the following radius.

$$D_{\text{threshold}} = \sqrt{2\left(1 - XCOR_{\text{threshold}}\right)} \tag{4.4}$$

The impact of this radius on clustering accuracy is investigated in Section 4.4.1.

4.2.2 Choice of an indexing scheme

By mapping delay sequences into a multidimensional space, pairwise computation of cross-correlation coefficients becomes unnecessary; inserting delay sequences into the multidimensional space and searching for neighbors within a radius replace the pairwise computations. This means that the complexity of those two operations, insertion and neighbor search, is critical to the overall performance. In this section, we introduce an index structure to facilitate them.

It is known that a well-designed multidimensional indexing scheme can insert N points in $O(N \log N)$ time and perform neighbor search within a sphere in $O(\log N)$ time [3]. Many indexing schemes have been proposed to store and manage multidimensional data, including the R-tree [20], R+-tree [46], R*-tree [5], SS-tree [49], and SR-tree [31]. As their names suggest, they are all based on a tree-like index structure with a similar insertion algorithm. However, each has a different search performance mainly because they employ different bounding shapes, which encompass the data in a subtree. The R-tree and its successors use rectangles as bounding shapes, and the SS-tree uses bounding spheres instead. The SR-tree integrates bounding rectangles and spheres to enhance the performance of neighbor search, especially for high-dimensional data. Since the SR-tree outperforms other schemes in neighbor search [31], it is used as the multidimensional indexing structure in the experiments presented in this dissertation.

Note that the clustering algorithm we propose does not depend on a specific indexing scheme; any multidimensional indexing scheme that efficiently performs insertion and neighbor search can be used.

4.2.3 Dimensionality reduction

To achieve high accuracy in detecting shared congestion, delay samples need to be collected for more than 10 seconds at a sampling rate of 10 Hz [32]. This means that the number of elements in a delay sequence is over 100, and so is the dimensionality of the multidimensional space. However, such high dimensionality increases the overhead of path clustering based on the multidimensional space, because the performance of multidimensional indexing deteriorates as the dimensionality of the data sets increases [31].

In our mapping between delay sequences and points in the multidimensional space, each delay sample corresponds to one coordinate of a point. This means that reducing dimensionality is equivalent to discarding delay samples, which immediately results in lower accuracy. Since all delay samples are considered to be "equally important," discarding any of them is equally harmful to accuracy. However, wavelet coefficients can break this symmetry. Using wavelet coefficients instead of time series enables efficient proximity search with lower dimensionality than that of using all delay samples. This is possible by utilizing wavelet coefficients at only large scales which bear information for slow-varying pattern of delay sequences [42].

Let the discrete wavelet transform[1] of $\tilde{\mathbf{x}}$ be

$$\tilde{\mathbf{X}} = (\tilde{X}_1, \tilde{X}_2, \ldots, \tilde{X}_m) = DWT(\tilde{\mathbf{x}})\,. \tag{4.5}$$

---

[1] Depending on the wavelet transform, the number of wavelet coefficients may be slightly different from the number of elements in $\tilde{\mathbf{x}}$.

One problem in mapping $\tilde{\mathbf{X}}$ instead of $\tilde{\mathbf{x}}$ to a multidimensional space is that the relationship between the distance in the multidimensional space and the corresponding cross-correlation coefficient shown in Eq. 4.3 may not hold any more. Fortunately, *if we choose DWT in Eq. 4.5 to be an orthonormal wavelet transform, the Euclidean distance between two time series is equal to the distance between their wavelet coefficients* [38].

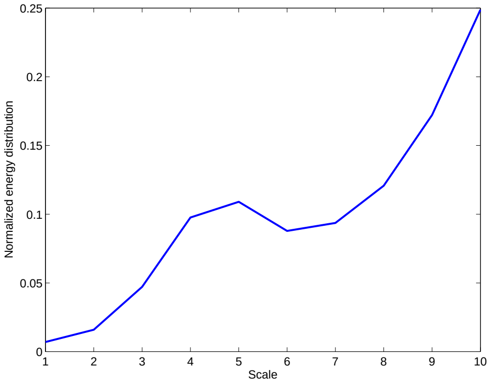


Figure 4.2: Energy distribution of wavelet coefficients

Figure 4.2 shows the energy (i.e. information) distribution of delay time series obtained from a congested link using ns-2 [17]. Most of the energy is concentrated on large-scale wavelet coefficients and any remaining energy is distributed sparsely over small-scale wavelet coefficients. Therefore, using wavelet coefficients only at large scales can achieve performance comparable to using all coefficients, while effectively reducing dimensionality. We will show empirically in Section 4.4.2 how many dimensions are needed to achieve good performance.

4.2.4 Reusing results of wavelet denoising

DCW uses discrete wavelet transform based on the Daubechies wavelet [32], which is orthonormal [14]. Therefore, we may use in $DWT(\tilde{\mathbf{x}})$ the same discrete wavelet transform that is used for denoising in Figure 3.8 to keep the computation cost low. In fact, $\tilde{\mathbf{X}} = DWT(\tilde{\mathbf{x}})$ can be obtained directly from $\mathbf{X} = DWT(\mathbf{x})$ without any need to compute $\tilde{\mathbf{x}}$.

Let $\overline{\mathbf{X}} = DWT(x_1 - \bar{x}, x_2 - \bar{x}, \ldots, x_m - \bar{x})$. Then

$$\begin{aligned}
\tilde{\mathbf{X}} &= DWT(\tilde{\mathbf{x}}) && (4.6)\\
&= DWT\left(\frac{(x_1 - \bar{x}, x_2 - \bar{x}, \ldots, x_m - \bar{x})}{\sqrt{\sum_{i=1}^{m}(x_i - \bar{x})^2}}\right) && (4.7)\\
&= \frac{DWT(x_1 - \bar{x}, x_2 - \bar{x}, \ldots, x_m - \bar{x})}{\sqrt{\sum_{i=1}^{m}(x_i - \bar{x})^2}} && (4.8)\\
&= \frac{\overline{\mathbf{X}}}{\|\overline{\mathbf{X}}\|}\,. && (4.9)
\end{aligned}$$

Since the discrete wavelet transform is a linear operation,

$$\begin{aligned}
\overline{\mathbf{X}} &= DWT(x_1 - \bar{x}, x_2 - \bar{x}, \ldots, x_m - \bar{x})\\
&= DWT(x_1, x_2, \ldots, x_m) - DWT(\bar{x}, \bar{x}, \ldots, \bar{x})\\
&= \mathbf{X} - \bar{x}DWT(\mathbf{I})
\end{aligned}$$

where $\mathbf{I} = (1, 1, \ldots, 1)$.

For indexing with wavelet coefficients at the K largest scales, we use only their corresponding coefficients from the above calculation. Thus, the final sequence to be stored in the multidimensional space is

$$\tilde{\mathbf{X}}' = (\tilde{X}_1, \tilde{X}_2, \ldots, \tilde{X}_k) \tag{4.10}$$

where k is the number of wavelet coefficients corresponding to the K largest scales.

4.3 Basic Implementation Steps

An actual implementation of path clustering consists of the following steps:

1. Select network paths to measure delay.
2. Measure delay samples to get $\mathbf{x}_0$ for each path.
3. Process $\mathbf{x}_0$ to obtain a wavelet coefficient vector with reduced dimensionality, $\tilde{\mathbf{X}}'$.
4. Collect $\tilde{\mathbf{X}}'$ for each selected path.
5. Cluster paths.

The first and fourth steps are application-dependent. For example, in the case of overlay multicast, delay is measured at every congested edge of a multicast tree, and each internal node of the tree collects data from its child nodes.

In this section, we will only describe the application-independent steps, i.e., what a node measuring delay should do (the second and third steps), and how a node collecting data performs clustering (the last step).

4.3.1 Measuring and processing delay samples

Either a source or destination of a path measures one-way delay with sampling frequency of 10 Hz as recommended by DCW [32]. Delay samples ($\mathbf{x}_0$ in Figure 3.8) are collected for 12.8 seconds to make the number of samples a power of 2 for calculation convenience. Then $\mathbf{x}_0$ is converted into $\tilde{\mathbf{X}}'$ by (i) using the wavelet transform, (ii) performing denoising, and (iii) applying Eq. 4.9–4.10. Only $\tilde{\mathbf{X}}'$ for the path is submitted to the node that clusters paths.

4.3.2 Path clustering

In general, a clustering problem is NP-hard [23]. However, since we know $D_{\text{threshold}}$, the maximum radius of a cluster defined in Eq. 4.4, we can design a simple and efficient algorithm for path clustering. The pseudo code is presented in Figure 4.3.

The algorithm begins with two sets: P, a set of $\tilde{\mathbf{X}}'$ for all paths, and S, initially empty and implemented with a multidimensional space indexed as described in Section 4.2. For notational simplicity, we use p to denote a member of P. We assume that the multidimensional indexing scheme being used (SR-tree [31] in our experiments) supports two operations: $\textsc{Insert}(S, p)$ which adds a point p to the space S, and $\textsc{Nearest-Neighbor-In-Sphere}(S, p)$ which searches in S for the nearest neighbor of p among points in the sphere centered at p with radius $D_{\text{threshold}}$. The latter returns one of them if there are multiple nearest neighbors, and **nil** if there is no neighbor in the sphere.

```
PATH-CLUSTERING($P$)
 1 ▷ $P$ is a set of $\tilde{\mathbf{X}}'$ for all paths.
 2 $S \leftarrow \varnothing$
 3 for each $p \in P$
 4     $s \leftarrow$ NEAREST-NEIGHBOR-IN-SPHERE($S, p$)
 5     if $s =$ nil
 6         INSERT($S, p$)
 7         $C_p \leftarrow \{p\}$
 8         $P \leftarrow P - \{p\}$
 9 for each $p \in P$
10     $s \leftarrow$ NEAREST-NEIGHBOR-IN-SPHERE($S, p$)
11     $C_s \leftarrow C_s \cup \{p\}$
12 return $\{C_s | s \in S\}$
```

Figure 4.3: Clustering algorithm

For each member p in P, the algorithm tests if any point stored in the multidimensional space is closer than the threshold from p. If none, the path represented by p does not share congestion with any of the paths corresponding to the previously inserted points, and thus it is added to S to create a new cluster. If there is a point closer than $D_{\text{threshold}}$, ignore p because p should belong to an existing cluster. In this way, after the first loop (Lines 3–8), S contains a set of points such that every point in P shares congestion with at lease one point in S while the points in S do not share congestion with one another. Each point inserted into S represents the center of a cluster. For each cluster, a set (C_p in Line 7) containing its center point is created to store points belonging to the cluster.

The second loop (Lines 9–11) identifies the members of each cluster. For each member p in P, the cluster of the closest center s in S is selected, and p is added to the selected cluster, C_s.

Finally, a set of all clusters is returned (Line 12).

For performance reason, our implementation always keeps the entire SR-tree in memory, although the original proposal for the SR-tree assumes that the tree is maintained on disk.

Note that the algorithm selects only one cluster for each path, whereas a path may belong to multiple clusters. If finding all clusters is more desirable, Lines 10 and 11 should be modified so that p is added to every cluster of which the center is in the sphere.

4.4 Performance Evaluation

In evaluating the proposed path clustering approach, the main focus is on the performance of clustering and various tradeoffs with different parameter values.

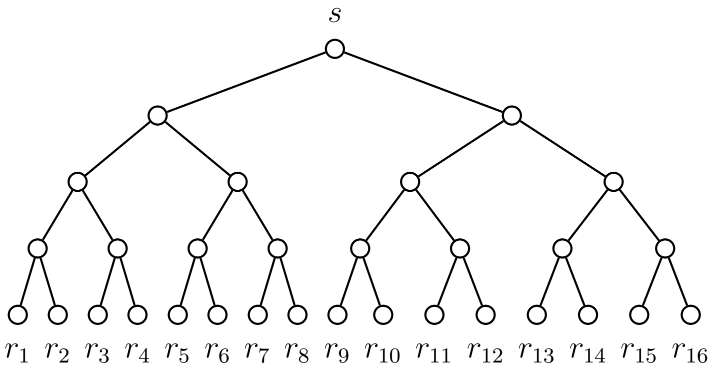


Figure 4.4: Network topology

We analyze the performance of the proposed approach using simulation data from ns-2 with the topology shown in Figure 4.4. The bandwidth of each link is 1.5 Mbps. To create background traffic, a different amount of short-lived TCP traffic is added to each link. TCP flows are created by ns-2's web traffic generator.

One-way delay samples are measured on 16 paths, from node s to node r_i for $1 \leq i \leq 16$. Along the path from node s to each r_i, at most one link is selected as a congested link, which is used by a large number of web sessions simultaneously, resulting in a loss rate between 5 and 10%. The number of web sessions is chosen uniformly between 180 and 250. The other links have less than 70 web sessions and no packet is lost. Every experiment was repeated 500 times to get an average.

Given N paths, we use the following as performance metrics for accuracy.

False positive rate Among $N(N-1)/2$ pairs of paths, the false positive rate is the fraction of path pairs such that the two paths in a pair do not share congestion with each other but belong to the same cluster.

False negative rate The fraction of path pairs such that the two paths in a pair share congestion with each other but belong to different clusters.

Clustering accuracy The fraction of path pairs that are neither false positives nor false negatives.

In this section, we use these metrics to study the impact of the threshold on neighbor search and the required dimensionality to maintain a reasonable accuracy. We also investigate the scalability of our clustering approach by comparing against clustering with pairwise operations.

4.4.1 Shared congestion threshold

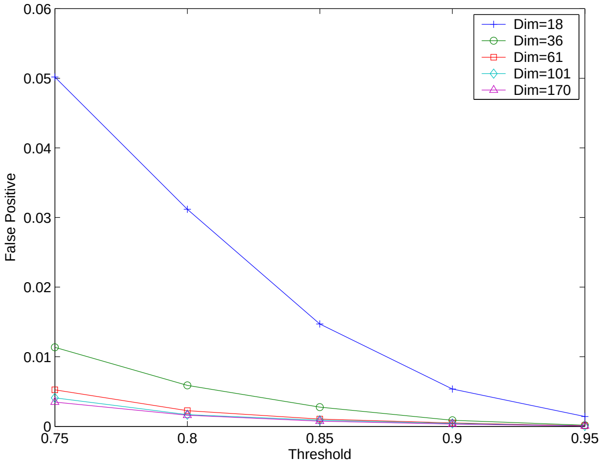


Figure 4.5: Impact of threshold on the false positive rate

The cross-correlation coefficient threshold ($XCOR_{\text{threshold}}$) affects both false positive and false negative rates directly, because the threshold determines the radius of neighbor search in clustering. A smaller radius (larger threshold) means more clusters with finer granularity, and accordingly it is less likely to get false positives. This observation is demonstrated in Figure 4.5, which shows the false positive rate versus threshold for a range of dimensionality between 18 and 170. The false positive rate decreases as the threshold increases for every dimensionality. It is especially prominent with the lowest dimensionality, 18.

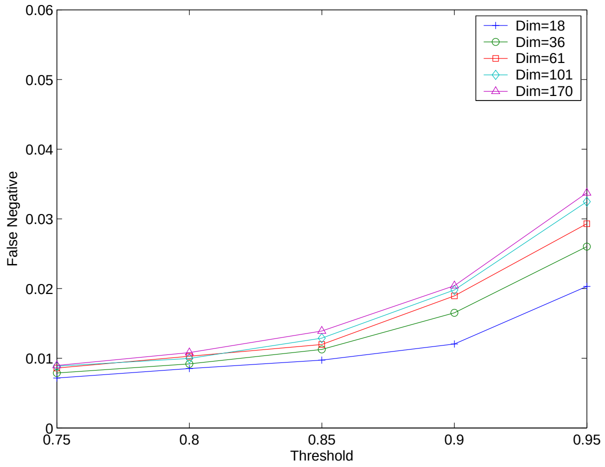


Figure 4.6: Impact of threshold on the false negative rate

Similarly, in Figure 4.6, the false negative rate increases as the threshold increases because a smaller radius leads to more clusters than needed. Therefore, there is clearly a tradeoff to be made between the false positive and false negative rates. The clustering accuracy is also affected by the threshold as shown in Figure 4.7. Depending on the dimensionality, a threshold between 0.75 and 0.9 maximizes the clustering accuracy.

Note that false positives are tolerable for some applications, but they may be completely intolerable for others [2]. So are false negatives. Therefore, an appropriate choice of threshold will vary from application to application.

4.4.2 Dimensionality

Dimensionality is another important parameter that affects performance. Because using fewer dimensions means that ignored dimensions cannot contribute to separating paths any more, the false positive rate increases as di-

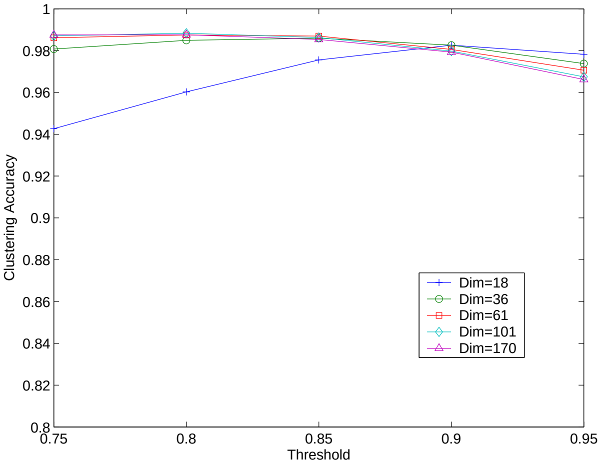


Figure 4.7: Impact of threshold on the clustering accuracy

mensionality decreases, while the false negative rate decreases.

Figures 4.5 and 4.6 shows that, with a low threshold (below 0.85), the decrease in the false positive rate as dimensionality increases is larger than the increase in the false negative rate. Therefore, the overall clustering accuracy is usually better with higher dimensionality as shown in Figure 4.8. With a high threshold (above 0.9), however, it is the opposite; the clustering accuracy gets worse with more dimensions. The reason is as follows. More dimensions often contribute to separating paths. However, if a threshold is high (meaning a small radius), the false positive rate is negligible, which means that the separations caused by additional dimensions are more likely to become false negatives than to correct false positives.

Even with a low threshold, increasing dimensionality is not the best choice. The overhead of maintaining the index structure must be taken into

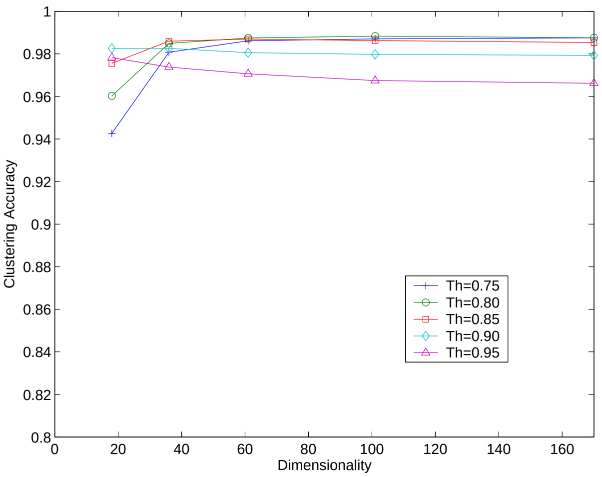


Figure 4.8: Tradeoff between accuracy and dimensionality

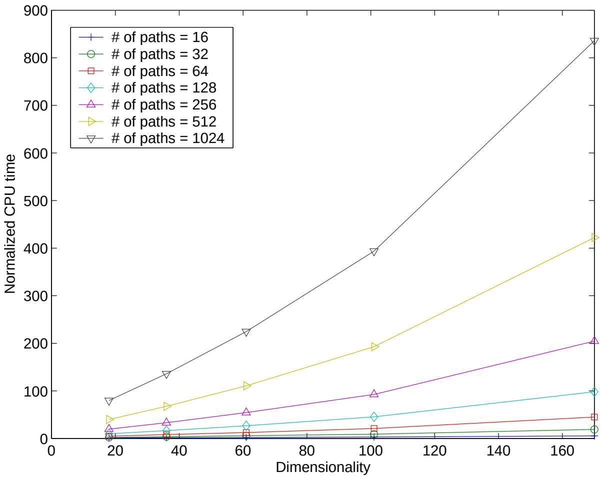


Figure 4.9: Overhead of high dimensionality

account; it is well-known that high dimensionality often incurs significant overhead in multidimensional indexing. To demonstrate this, we plot in Figure 4.9 the CPU time required for clustering as a function of dimensionality. Since the actual CPU time depends on many factors, we normalize it so that the CPU time of the fastest case (16 paths with 18 dimensions) is equal to one unit of time. With a C++ implementation on a Pentium 2GHz machine, one unit of time is about 5 milliseconds.

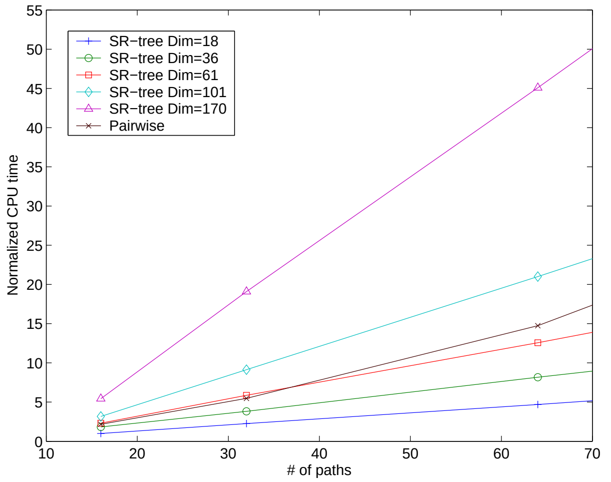
Figure 4.9 shows that the CPU time increases rapidly as the number of dimensions increases, especially with a large number of paths. Hence, the dimensionality should be kept minimal as long as the false positive and negative rates are acceptable. Considering the results in Figures 4.5, 4.6, and 4.8, we believe that 36 dimensions are more than sufficient in most applications, and that 18 dimensions are reasonable if used with a high threshold.

4.4.3 Scalability

The main goal of the proposed clustering approach is to achieve better scalability than the use of pairwise comparisons. While the multidimensional indexing improves the theoretical bound on time complexity, it would be more interesting to study when and how much the proposed approach outperforms the use of pairwise comparisons.

In Figure 4.10, we compare the proposed multidimensional indexing approach against the pairwise approach. Both approaches use DCW as a shared congestion detection technique. We plot the CPU time required for clustering

(a) Small-scale clustering



(b) Large-scale clustering

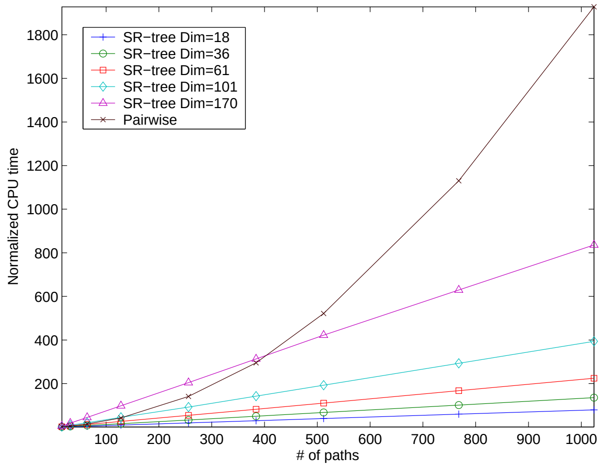


Figure 4.10: Clustering time

versus the number of paths for different dimensionalities. Two different scales, less than 70 paths in Figure 4.10(a) and up to 1024 paths in Figure 4.10(b), are considered. The CPU time is normalized so that the case with 18 dimensions and 16 paths takes 1 unit of time.

The comparison for a small number of paths presented in Figure 4.10(a) shows the overhead caused by multidimensional indexing. The multidimensional indexing approach takes nontrivial time to maintain a complicated data structure. Therefore, the pairwise approach is faster if 61 or more dimensions are used to cluster less than about 30 paths. However, due to its better time complexity, our approach exhibits better performance when the number of paths gets larger. Notice that the difference in slope between curves due to different time complexity.

This better CPU time performance is clearer when the curves are extended in Figure 4.10(b). Because of its $O(N^2)$ complexity, the pairwise approach curve diverges from the other curves as the number of paths increases. The CPU time increase with dimensionality is significant, and low dimensionalities incur a fairly small overhead. Since the difference between 18 and 36 dimensions in terms of accuracy is rather large as observed in Figure 4.7, 36 dimensions would be a reasonable choice in practice.

4.5 Summary

For large-scale distributed systems such as overlay networks, it is crucial to identify bottlenecks in the network so as to allocate network resources efficiently. However, previously proposed techniques to detect network bottlenecks shared by multiple paths do not scale well because they handle only two paths at a time. In this chapter, a novel approach to cluster paths sharing congestion was proposed. The proposed approach employs multidimensional indexing and wavelet transform for better scalability. It outperforms previous approaches when dealing with more than tens of paths. The granularity of clustering is controllable by adjusting the neighbor search radius. Tradeoffs between time-space complexity and accuracy with different dimensionalities were investigated. Many overlay networks will benefit from scalable path clustering by using the clustering information to improve its topology and, in turn, overall throughput.

Chapter 5

Eliminating Bottlenecks

Previous chapters presented how to identify bottlenecks in overlay networks in a scalable manner. In this chapter, the proposed approach is applied to overlay multicast to remove bottlenecks from a multicast tree, without relying on the requirements used in Chapter 2.

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 [7]. 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 overlay connections 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 overlay connections in the multicast tree that traverse those bottlenecks using DCW, and remove them by changing the multicast tree topology.

In this chapter, an algorithm that removes *all* bottlenecks shared by multiple overlay connections is presented. In the 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. The algorithm is implemented in a distributed fashion, and compared with other heuristically-built multicast trees using various performance metrics. Simulation results show that even in a network with a dense receiver population, the proposed algorithm finds a tree that satisfies all the receiving nodes while other heuristic-based approaches often fail.

5.1 Two-Layer Network Model

The network model in this chapter 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.

5.1.1 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 $P_L(u, v) = \{(u, n_1), (n_1, n_2), \ldots, (n_i, v)\}$ chosen to minimize the total cost of the links in the set. If there are more than one such path, we assume that the routing algorithm always selects the same path among them.

5.1.2 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 $P_T(u, v) \subset T$.

5.1.3 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 $\mathit{Src}(f)$, a sink node $\mathit{Snk}(f)$, and the rate of the flow $\mathit{Rate}(f)$.

Let $F(m, n)$ be a set of flows passing through the link $(m, n) \in L$. Formally, $F(m, n) = \{f | (m, n) \in P_L(\mathit{Src}(f), \mathit{Snk}(f))\}$. A link (m, n) is a *bottleneck* of the multicast session if and only if $B(m, n) < \sum_{f \in F(m,n)} \mathit{Rate}(f)$. 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.

5.2 Bottleneck Elimination 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 [32]. Before we describe the algorithm, we define notation to be used in explanation.

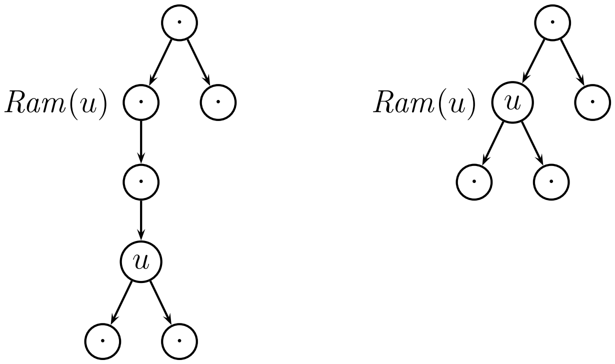


Figure 5.1: Ramification point

- $r \in H$ 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 Fig. 5.1.

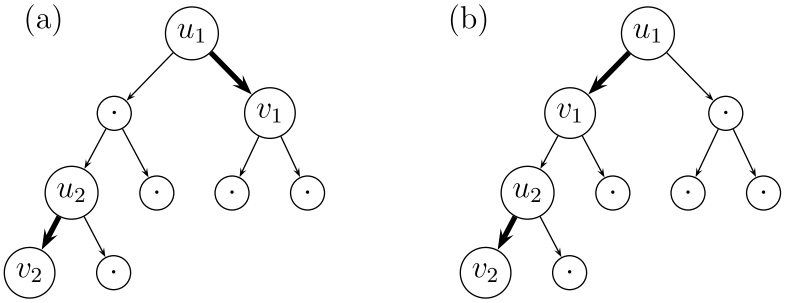


Figure 5.2: (a) Inter- and (b) intra-path shared bottlenecks

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 Fig. 5.2, 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 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 a finite number of 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.

5.2.1 Inter-path shared bottleneck

The algorithm to remove an inter-path shared bottleneck is shown in Fig. 5.3. See also Fig. 5.2(a) for illustration. When an inter-path shared bottleneck is detected between two edges (u_1, v_1) and (u_2, v_2), the edge farther from the root, (u_2, v_2), 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, then 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 do not 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.

```
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   $T \leftarrow T \cup \{SLeaf(v_1), v_2\} - \{(u_2, v_2)\}$
 4 else
 5   $t \leftarrow$ 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   $T \leftarrow T \cup \{(t, v_2)\} - \{(u_2, v_2)\}$
 7 if $\{(u_2, x) | (u_2, x) \in T\} = \varnothing$
 8   $u, v \leftarrow Parent(Ram(u_2)), Ram(u_2)$
 9   $c \leftarrow \arg\min_{i \in \{w | Parent(w) = u, w \neq v\}} d(i, SLeaf(i))$
10   $T \leftarrow T \cup \{(SLeaf(c), v)\} - \{(u, v)\}$
```

Figure 5.3: Removal of an inter-path shared bottleneck

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.

Lemma 5.2.1. *Let D and D' denote leaf distance vectors before and after* Remove-Inter-Path-Shared-Bottleneck *respectively. Then we have $D \prec D'$.*

Proof 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) does not 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 does not 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. $\square$

5.2.2 Intra-path shared bottleneck

Figure 5.4 presents the algorithm to remove an intra-path shared bottleneck. See also Fig. 5.2(b) for illustration. 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 5.2.1, 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.

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

Figure 5.4: Removal of an intra-path shared bottleneck

From the two cases above, we conclude the following lemma.

Lemma 5.2.2. *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 $C < C'$.*

Proof If $SLeaf(v_1) \neq SLeaf(v_2)$ in Line 2, $D \prec D'$ holds by Lemma 5.2.1. 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 does not 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 does not 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 $C < C'$. □

5.2.3 Shared bottleneck elimination

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

Theorem 5.2.3. *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 5.2.1 and Lemma 5.2.2. 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 an 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.

5.3 Bottleneck Elimination Protocol

Our algorithm is based on the 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.

5.3.1 Shared congestion removal

In real networks, a shared bottleneck is a congested link shared by multiple flows belonging to the same multicast session. DCW [32] 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 [32]. (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.

5.3.2 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.

5.3.3 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 does not 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.

5.4 Evaluation

For evaluation, the proposed protocol is compared against two heuristic-based schemes. The first one optimizes the multicast tree using bandwidth estimation as in Overcast [28]. 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 does not 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 [40], 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 [32]. 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.

5.4.1 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 [6]. There are 24 transit routers, 576 stub routers, and 1152 hosts participating in 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 [10].

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 tree to the unicast delay between the same nodes. Defined in [10].

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 does not change any more. However, the bandwidth heuristic may oscillate as shown in Overcast [28] 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 5.5 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

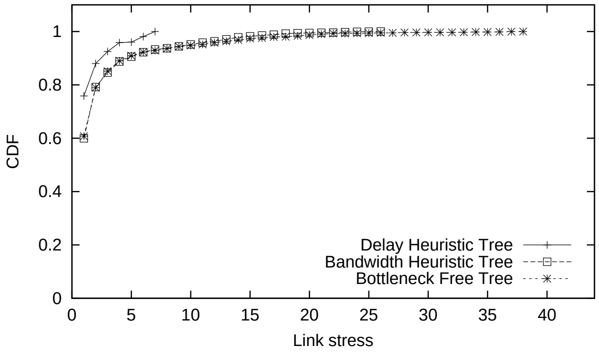


Figure 5.5: Link stress distribution

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.

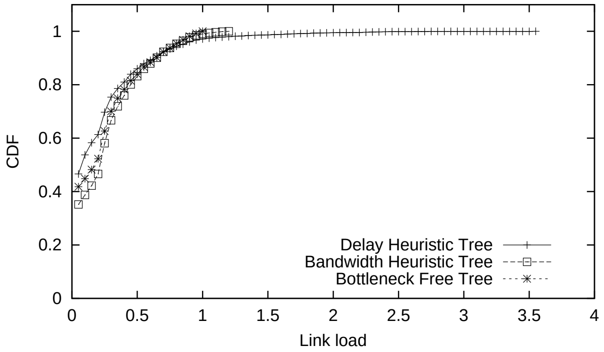


Figure 5.6: Link load distribution

Figure 5.6 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.

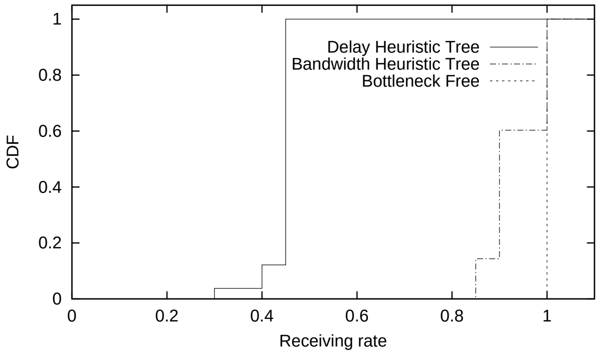


Figure 5.7: Receiving rate distribution

In Fig. 5.7, 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 Fig. 5.6; 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 Fig. 5.8. Because the bandwidth heuristic pays little attention to delay, its RDP is worse than the others.

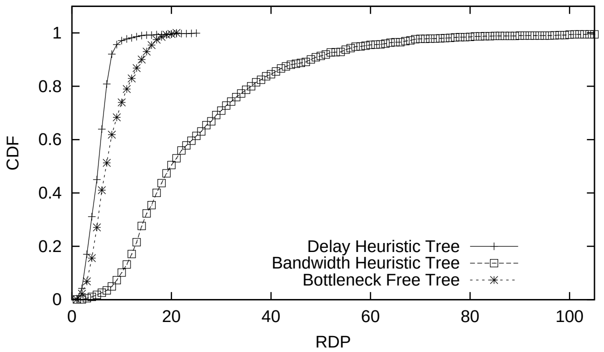


Figure 5.8: RDP distribution

We have to mention that this experiment regarding 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 does not have strong correlation with round-trip time [40], we do not expect significant improvement with TCP fair rate allocation.

5.4.2 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 number of packet transmissions required for the last case in removing an intra-path shared bottleneck, where edges are reversed along the path between edges sharing the bottleneck, is equal to the length of the path. Therefore the length of a round interval must be on the order of tens of seconds.

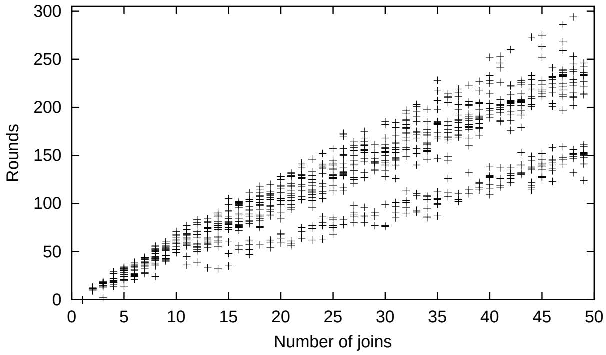


Figure 5.9: Convergence after nodes join

Fig. 5.9 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.

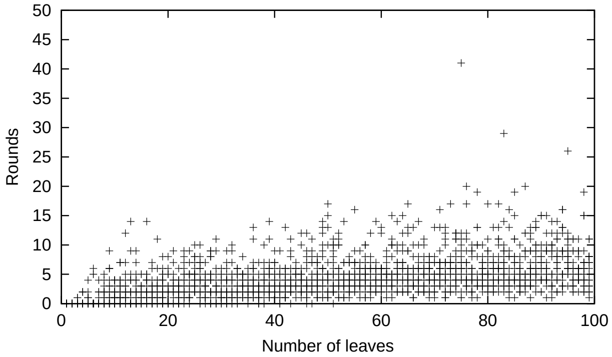


Figure 5.10: Convergence after nodes leave

Unlike joins, concurrent leaves can be handled relatively easily. In Fig. 5.10, 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. Therefore, it is critical to remove the bottleneck early.

We plot in Fig. 5.11 the time it takes to remove bottlenecks with different depths in the tree. A new shared bottleneck was created by reducing available bandwidth. As we expect, a bottleneck near a leaf (depth larger than 25) can be removed within a couple of rounds. On the other hand, a bottleneck

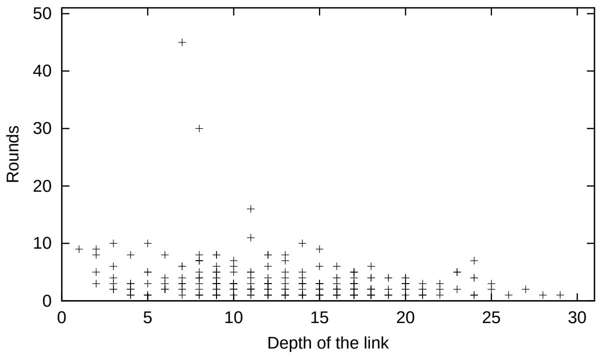


Figure 5.11: Convergence after available bandwidth change

close to the root takes longer—up to ten rounds with a few outliers.

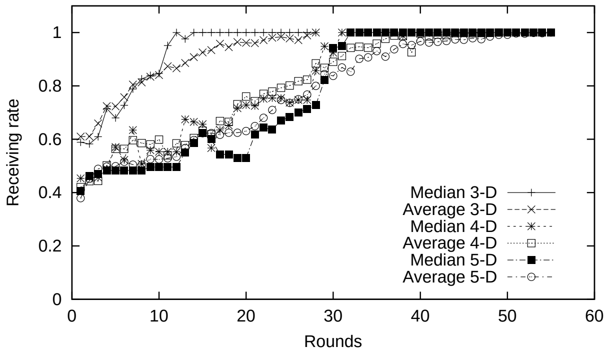


Figure 5.12: Receiving rate increase as a tree converges

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 5.12, 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 of the receiving rate increase is achieved within early half of the convergence time.

5.4.3 Effects of Measurement Errors

All the three schemes we evaluated in Section 5.4.1 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 5.13 shows the 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. This means that the poor performance of the bandwidth heuristic in earlier simulations resulted from the weakness of the heuristic itself, not from the 20% error we introduced.

For shared congestion detection, there are two types of errors: false

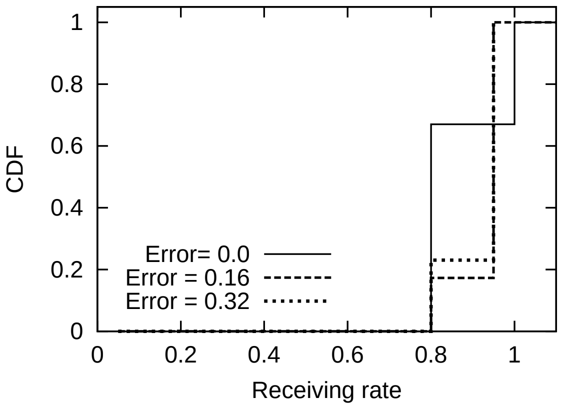


Figure 5.13: Effects of errors in bandwidth estimation

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 [32].

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 5.14. However, we notice only a very slight increase of the RDP as the error rate increases up to 6%.

The effect of false negatives is also negligible in the range from 0 to 6%, as shown in Figure 5.15; this type of errors make the convergence slower due to hidden shared bottlenecks, but only by a few rounds. Therefore, errors in

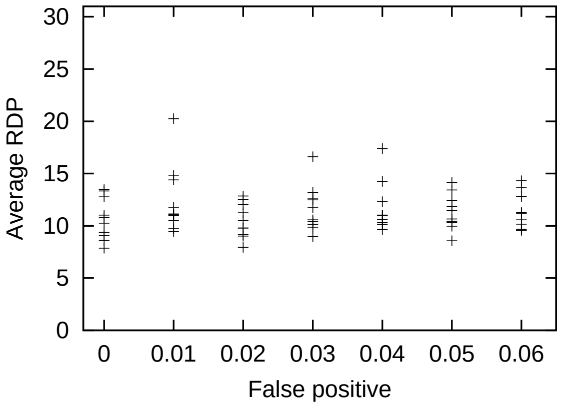


Figure 5.14: Effects of false positive in shared congestion

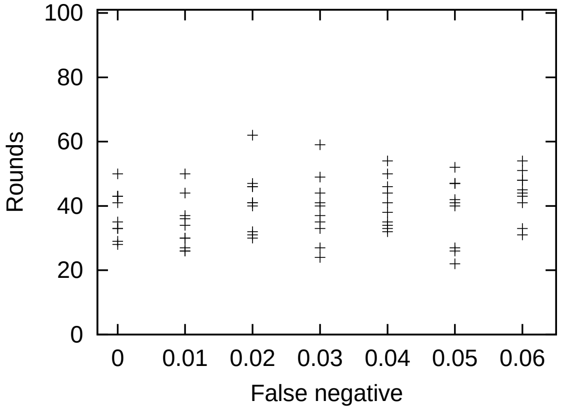


Figure 5.15: Effects of false negative in shared congestion

shared congestion detection less then 6%, which is achievable with DCW, do not affect much the performance of our protocol.

Also note that, unlike the bandwidth heuristic, which only looks for a 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.

5.5 Summary

In bandwidth-demanding multicast applications such as multimedia distribution, it is critical for a user to receive at the full source rate so as not to experience quality degradation. Though many heuristics to achieve high receiving rate have been proposed, they often fail to provide required receiving rate. In this chapter, a new tree construction algorithm that removes bottlenecks caused by the multicast session was proposed, and it was proved that the algorithm 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.

Chapter 6

Conclusion

Building an efficient topology is crucial in overlay networks. Because an overlay network consists of a large number of connections between participating end hosts, having a topology that minimizes interference between them is essential to achieve high throughput. This dissertation focuses on inferring network characteristics and improving overlay network topology using them. For bandwidth-demanding applications, one of the most critical network characteristics is available bandwidth. Chapter 2 showed that knowledge of available bandwidth could increase average available bandwidth of overlay multicast significantly. By replacing low-bandwidth overlay connections with high-bandwidth ones, the multicast tree built with the proposed algorithm achieved 30 times higher average bandwidth than randomly built trees.

Although it demonstrated very well that an overlay network benefits greatly from information on the underlying network characteristics, such an algorithm does not take interference between overlay connections into consideration, and therefore leads to suboptimal topology. Thus, how to identify and avoid such interference was also studied.

DCW was proposed to identify interfering overlay connections using a signal processing technique, wavelet denoising. It infers robustly whether two Internet paths are sharing the same congested link or not. Due to the denoising process making DCW tolerate synchronization offset up to one second, DCW became an ideal solution for overlay networks which usually have a large number of connections with different sources and destinations.

To apply DCW to more than two paths in a scalable manner, multidimensional indexing was introduced. It stores data collected from each path as a single point in a multidimensional space, and indexes it using a tree-like structure. In this space, a neighbor search for a path retrieves all paths that share congestion with it. The computational complexity of multidimensional indexing was reduced by using a subset of wavelet coefficients that were more relevant to shared congestion detection. For example, decreasing the number dimensions for indexing from 170 to 36 caused only a slight decrease of detection accuracy. Experiments showed that the multidimensional indexing outperformed the pairwise approach in clustering paths by shared congestion. After clustering paths, an overlay network can avoid bottlenecks in its topology by choosing at most one path in each cluster.

As a case study, an algorithm that improves overlay multicast topology was designed. The proposed algorithm finds bottlenecks shared by multiple overlay connections using DCW, and removes them from the multicast tree by relocating involved subtrees. It was proved that there remains no such bottleneck in the tree upon termination of the algorithm. The tree built with the algorithm also maintained low link load and shared delay penalty in simulations. A similar approach to finding bottlenecks and removing them through topology changes can be applied to other types of overlay networks. The techniques and algorithms proposed in this dissertation will serve as a foundation on which future applications can achieve higher throughput by building more efficient overlay networks.

Bibliography

[1] Patrice Abry, Richard Baraniuk, Patrick Flandlin, Rudolf Riedi, and Darryl Veitch. Multiscale nature of network traffic. *IEEE Signal Proessing Magazine*, 19(3):28–46, May 2002.

[2] Aditya Akella, Srinivasan Seshan, and Hari Balakrishnan. The impact of false sharing on shared congestion management. In *Proceedings of the 11th IEEE International Conference on Network Protocols*, November 2003.

[3] Sunil Arya, Theocharis Malamatos, and David M. Mount. Space-time tradeoffs for approximate spherical range counting. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 535–544, January 2005.

[4] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proceedings of ACM SIGCOMM 2002*, August 2002.

[5] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM SIGMOD '90*, pages 322–331, 1990.

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

[7] 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.

[8] Yatin Chawathe and Mukund Seshadri. Broadcast Federation: an application layer broadcast internetwork. In *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, May 2002.

[9] Yang-hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. Enabling conferencing applications on the Internet using an overlay multicast architecture. In *Proceedings of ACM SIGCOMM 2001*, August 2001.

[10] 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.

[11] Mark Coates, Alfred O. Hero III, Robert Nowak, and Bin Yu. Internet tomography. *IEEE Signal Proessing Magazine*, 19(3):47–65, May 2002.

[12] Reuven Cohen and Gideon Kaempfer. A unicast-based approach for streaming multicast. In *Proceedings of IEEE INFOCOM 2001*, April

2001.

[13] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 17. MIT Press, 1990.

[14] I. Daubechies. The wavelet transform, time-frequency localization and signal analysis. *IEEE Transactions on Information Theory*, 36(5):961–1005, September 1990.

[15] D. L. Donoho. De-noising by soft-thresholding. *IEEE Transactions on Information Theory*, 41(3):613–627, May 1995.

[16] D. L. Donoho and I. M. Johnstone. Ideal spatial adaptation by wavelet shrinkage. *Biometrika*, 81:425–455, 1994.

[17] Kevin Fall and Kannan Varadhan, editors. *The ns Manual*. The VINT Project, 2005.

[18] Sally Floyd, Mark Handley, Jitendra Padhye, and Jörg Widmer. Equation-based congestion control for unicast applications. In *Proceedings of ACM SIGCOMM 2000*, August 2000.

[19] Mukul Goyal, Roch Guerin, and Raju Rajan. Predicting TCP throughput from non-invasive network sampling. In *Proceedings of IEEE INFOCOM 2002*, June 2002.

[20] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD '84*, pages 47–57, 1984.

[21] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP friendly rate control (TFRC): Protocol specification. RFC 3448, The Internet Society, January 2003.

[22] Katrina M. Hanna, Nandini Natarajan, and Brian Neil Levine. Evaluation of a novel two-step server selection metric. In *Proceedings of the 9th IEEE International Conference on Network Protocols*, November 2001.

[23] Pierre Hansen and Brigitte Jaumard. Cluster analysis and mathematical programming. *Mathematical Programming*, 79(1-3):191–215, 1997.

[24] 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.

[25] Khaled Harfoush, Azer Bestavros, and John Byers. Robust identification of shared losses using end-to-end unicast probe. Technical Report BUCS–TR–2001–001, Computer Science Department, Boston University, Massachusetts, U.S.A., January 2001. Errata to the previous reference.

[26] Polly Huang, Anja Feldmann, and Walter Willinger. A non-instrusive, wavelet-based approach to detecting network performance problems. In *Proceedings of the First ACM SIGCOMM Internet Measurement Workshop*, pages 213–227, November 2001.

[27] Manish Jain and Constantionos Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP

throughput. In *Proceedings of ACM SIGCOMM 2002*, August 2002.

[28] 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.

[29] Tianji Jiang, Mostafa H. Ammar, and Ellen W. Zegura. Inter-receiver fairness: a novel performance measure for multicast ABR sessions. In *Proceedings of ACM SIGMETRICS '98*, June 1998.

[30] 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.

[31] Norio Katayama and Shin'ichi Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of ACM SIGMOD 1997*, May 1997.

[32] 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.

[33] Min Sik Kim, Taekhyun Kim, YongJune Shin, Simon S. Lam, and Edward J. Powers. Scalable clustering of Internet paths by shared conges-

tion. Technical Report TR–05–25, Department of Computer Sciences, The University of Texas at Austin, May 2005.

[34] Min Sik Kim, Simon S. Lam, and Dong-Young Lee. Optimal distribution tree for Internet streaming media. In *Proceedings of IEEE ICDCS 2003*, Providence, RI, May 2003.

[35] Min Sik Kim, Yi Li, and Simon S. Lam. Eliminating bottlenecks in overlay multicast. In *Proceedings of IFIP Networking 2005*, pages 893–905, May 2005.

[36] Zhijun Lei. Media transcoding for pervasive computing. In *Proceedings of the 9th ACM International Conference on Multimedia*, September 2001.

[37] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. University of California Press, 1967.

[38] Stéphane Mallat. *A Wavelet Tour of Signal Processing.* Academic Press, 2nd edition, 1999.

[39] David L. Mills. Network time protocol (version 3) specification, implementation and analysis. RFC 1305, March 1992.

[40] 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.

[41] Dimitrios Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel. ALMI: an application level multicast infrastructure. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, March 2001.

[42] Ivan Popivanov and Renee J. Miller. Similarity search over time-series data using wavelets. In *Proceedings of the 18th International Conference on Data Engineering*, February 2002.

[43] Injong Rhee, Volkan Ozdemir, and Yung Yi. TEAR: TCP emulation at receivers — flow control for multimedia streaming. Technical report, Department of Computer Science, North Carolina State University, April 2000.

[44] Rudolf H. Riedi, Matthew S. Crouse, Vinay J. Ribeiro, and Richard G. Baraniuk. A multifractal wavelet model with application to network traffic. *IEEE Transactions on Information Theory*, 45(3):992–1018, 1990.

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

[46] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings*

of the 13th International Conference on Very Large Data Bases, pages 507–518, 1987.

[47] YongJune Shin, Edward J. Powers, William M. Grady, and S. C. Bhatt. Optimal Daubechies' wavelet bases for detection of voltage sag in electric power distribution and transmission systems. In *Wavelet Applications in Signal and Image Processing VII, SPIE*, pages 873–883, July 1999.

[48] Harry L. Van Trees. *Detection, Estimation, and Modulation Theory.* John Wiley & Sons, December 1968.

[49] David A. White and Ramesh Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523. IEEE Computer Society, 1996.

[50] W. Williams. Uncertainty, information, and time-frequency distributions. In *Advanced Signal Processing Algorithms, Architectures and Implementations II, SPIE.*, pages 144–156, July 1991.

[51] Maya Yajnik, Jim Kurose, and Don Towsley. Packet loss correlation in the MBone multicast network. In *Proceedings of IEEE Global Internet 1996*, November 1996.

[52] Y. Richard Yang, Min Sik Kim, and Simon S. Lam. Optimal partitioning of multicast receivers. In *Proceedings of the 8th IEEE International Conference on Network Protocols*, Osaka, Japan, November 2000.

[53] Y. Richard Yang, Min Sik Kim, and Simon S. Lam. Transient behaviors of TCP-friendly congestion control protocols. *Computer Networks*, 41(2):193–210, February 2003.

[54] Ossama Younis and Sonia Fahmy. Flowmate: Scalable on-line flow clustering. *IEEE/ACM Transactions on Networking*, 13(2):288–301, April 2005.

[55] Yin Zhang, Nick Duffield, Vern Paxson, and Scott Shenker. On the constancy of Internet path properties. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, November 2001.

Index

Vita

Min Sik Kim was born in Seoul, Korea on September 25, 1974, the son of Youngsoon Shin and Taeshin Kim. After completing his work at Seoul Science High School, Seoul, Korea, in 1993, he entered Seoul National University in Seoul, Korea, and received the degree of Bachelor of Science in Engineering in February 1996. During the years 1996 through 1998, he served in the Republic of Korea Army. In August 1999, he entered the Graduate School of The University of Texas at Austin.

Permanent address: 55-10 Yeokchon 2-Dong Eunpyeong-Gu
Seoul 122-900, Korea

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.