WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DESIGN ISSUES OF RESERVED DELIVERY SUBNETWORKS

by

Ruibiao Qiu

Prepared under the direction of Professor Jonathan S. Turner

A dissertation presented to the Henry Edwin Sever Graduate School of
Washington University in partial fulfillment of the
requirements for the degree of

DOCTOR OF SCIENCE

May 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

DESIGN ISSUES OF RESERVED DELIVERY SUBNETWORKS

by

Ruibiao Qiu

ADVISOR: Professor Jonathan S. Turner

May 2006

Saint Louis, Missouri

The lack of per-flow bandwidth reservation in today's Internet limits the quality of service that an information service provider can provide. This dissertation introduces the reserved delivery subnetwork (RDS), a mechanism that provides consistent quality of service by implementing aggregate bandwidth reservation. A number of design and deployment issues of RDSs are studied.

First, the configuration problem of a single-server RDS is formulated as a minimum concave cost network flow problem, which properly reflects the economy of bandwidth aggregation, but is also an NP-hard problem. To make the RDS configuration problem tractable, an efficient approximation heuristic, largest demands first (LDF), is presented and studied. In addition, performance improvements with local search heuristic is investigated. A traditional negative cycle reduction and a new negative bicycle reduction algorithms are applied and evaluated.

The study of RDS configuration problems is then extended to multi-server RDSs. The configuration problem can be similarly formulated as the single-server RDS configuration problem; however, the major challenge of multi-server RDS configuration is the optimal server locations. A number of server placement algorithms are evaluated using simulations. The simulation results show that a class of greedy algorithms provide the best solutions. In addition to configuration problem, the dynamic load redistribution mechanism is studied to improve the tolerance to server failures. A configuration algorithm to build redistribution subnetworks is proposed and evaluated to deal with single server failures in a group of servers.

Besides the exclusive bandwidth access, there are potentials to further improve end-to-end performance in an RDS because end hosts can utilize the knowledge about the underlying networks to achieve better performance than in the ordinary Internet. These improvements are illustrated with a source traffic regulation technique to resolve the unbalanced bandwidth utilization problem in an RDS. A per-connection and an aggregated regulation algorithm for single-server and multi-server RDSs are presented and studied.

To Hung-Jen, Audrey, Emma, and my parents

Contents

List of Figures

Acknowledgments

First of all, I would like to thank my advisor, Professor Jonathan S. Turner for his inspiration, guidance, care, and the opportunity to realize my goal to the fullest. During my time as a graduate student and as graduate research assistant at ARL, I have learned a great deal from Dr. Turner, not only his brilliant insights and broad knowledge in the networking area, but also his passion for research and his care for students.

I would like to thank the other members of my thesis committee: Dr. Roger Chamberlain, Dr. Sergey Gorinsky, Dr. Weixiong Zhang, and Dr. Norman Katz. Their invaluable feedback and suggestions widened my views of the problem, and helped me improve the dissertation tremendously in many ways. Some of their feedbacks and suggestions have become important parts of this dissertation.

I would like to thank all staff members and fellow students at ARL and the Computer Science department for their stimulating ideas and great help.

Thanks to my parents, Quangxiang Qiu and Peihe Yan, my brother, Ruidi, and my sister, Juanli, for their love and support.

Finally, I would thank my wife, Hung-Jen for her support. Without her, this dissertation would not be possible. Her encouragement and love helped me through the difficult times. Especially, she took good care of our lovely daughters, Audrey and Emma, with whom I should have spent more time.

Ruibiao Qiu

Washington University in Saint Louis
May 2006

Chapter 1

Introduction

1.1 Motivations

The Internet has become an information infrastructure that we increasingly depend on in our daily life. However, the Internet in its current state is not capable of meeting the needs of mission critical applications. One key deficiency of the Internet is the lack of effective per-flow bandwidth reservation mechanisms. As a result, the majority of today's Internet traffic is best-effort traffic, and guaranteed services are not readily available. Because of the best effort nature of today's Internet, there is no way to distinguish between transaction-oriented mission critical data traffic and traffic from causal web browsing. All different traffic sources have to compete equally for the bandwidth resource, making it it difficult to provide guarantees for mission critical applications. In addition, the Internet is vulnerable to malicious attacks, such as denial of service (DoS) and worm attacks. For example, in January of 2003, the Internet "slammer" worm attack left thousands of bank customers without ATM access, and dozens of flights grounded [16]. The data communication between the servers at the bank and airline companies headquarters and the terminals on ATMs and in the airports was severely affected when the Internet got heavily congested with traffic generated by the worms. Clearly, the current Internet is an insufficient information infrastructure, and needs great improvements to provide consistent and stable services comparable with traditional information infrastructure, such as telephone networks.

In order to make the Internet a better information infrastructure, various per-flow bandwidth reservation techniques have been proposed to improve services of the Internet. However, they are not widely deployed as expected. The major hurdles are the concerns about the

costs of upgrading and maintaining the per-flow reservation mechanisms from the network service providers, because it is widely believed that the per-flow reservation is too expensive to be practical, especially in the core networks.

On the other hand, the idea of aggregate bandwidth reservation is widely accepted. Instead of reserving bandwidth for individual flows, aggregate bandwidth reservation reserves bandwidth for an aggregate of flows. Aggregate bandwidth reservation can be easily implemented in the existing network service providers' backbone networks as long as the backbone routers are capable of efficient packet classification and support different queues for different flows. As these two functions are quite standard in today's routers, it makes aggregate bandwidth reservation a more viable option than per-flow reservation.

In this dissertation, we introduce a new aggregate bandwidth reservation based network service called *Reserved Delivery Subnetwork* (RDS) as an alternative solution for providing more consistent quality of service in today's Internet. An RDS is provided by a network service provider (such as a telecommunication carrier), and is designed for information service providers who need to delivery consistent quality of service to their customers even under very extreme network conditions, such as worm attacks. An RDS provides a subnetwork for an information service provider to connect from a central location to the access routers at different locations where customers of the information service are found. The links in an RDS are carefully provisioned with sufficient bandwidth so that traffic from the source node can flow through to the sinks without contention from other traffic sources, improving quality of service. Although it is difficult to provide quality of service for individual flows in the current Internet, RDSs give service providers a way to address the quality of service issue on an aggregate basis. In addition, bandwidth limits on reverse paths provide a protection mechanism against malicious attacks.

An ideal reserved delivery subnetwork must be configured to achieve two main goals: first, it must satisfy the demand of all customers at different locations; second, the network resource must be utilized efficiently so that more services can be provided. In addition, the end hosts in an RDS should be able to leverage the RDS infrastructure to achieve better performance than in the ordinary Internet. In this dissertation, a number of issues in the configuration and operation of reserved delivery subnetworks are studied. Specifically, the configuration problem for a single-server RDS is first studied. The results are then extended

to the configuration problem for larger RDSs with multiple servers. The fault tolerance issues in a multi-server RDS is also investigated, and an algorithm to configure redistribution subnetworks for server failures and overloading is presented. For the operation of an RDS, the end-to-end performance improvements inside an RDS are studied, and a source traffic regulation technique is introduced to leverage the underlying RDS.

1.2 Applications

A number of network services and applications can benefit from the deployment of reserved delivery subnetworks.

One of the most direct applications is web content delivery. A web site or an Internet content provider (ICP), such as CNN, can purchase such a service from the physical network service provider, such as SBC. An RDS can be set up that is rooted at the access router where the ICP servers reside, and connect to all locations where the majority of user demands are found. The ICP can deliver consistent service to end users with some degree of bandwidth guarantee, even under extreme network conditions.

Another RDS application can be found in enterprise virtual private networks (VPNs). For example, in a bank or an airline company that depends heavily on the time-critical delivery of transaction-oriented data, the company headquarters can subscribe to a customized reserved delivery subnetwork such that data communication will not be interrupted even when the network is under attack. It is possible that a service provider and the end users are located in different network domains run by different physical network service providers. Instead of negotiating a multilateral service level agreement with each individual network provider, a special type of service provider can be involved. We can call such a service provider as a *Reserved Delivery Subnetwork Provider* or an *RDSP.* An RDSP provides reserved delivery service to a customer by constructing an RDS from the customer to their end users. The subnetwork may span multiple network domains. According to the customer requirements, the RDSP purchases reserved bandwidth on subnetwork links from each individual network provider, and gains service revenues from the customers that subscribe to the service from it.

RDSs could extend their applications to grid computing [43]. In traditional studies of grid computing problems, the focus has been mostly put on the resource management on the nodes of a computational grid. Less attention was paid to the bandwidth resource management of the interconnecting networks. However, this is an important aspect of grid computing because inefficient use of the bandwidth resource may limit the performance of a computational grid application. Traffic flows inside a grid can also benefit from the economy of bandwidth aggregation, thus, an RDS can help manage bandwidth resource in a grid efficiently and effectively its performance.

Multimedia traffic flows have greater demands for consistent bandwidth availability to make the playout smooth. An RDS can improve multimedia streaming services by reserving aggregate bandwidth for a streaming server so that the streaming traffic is not affected by other best effort traffic. The burstiness of certain types of multimedia streams is generally bounded by their encoding standards, and can be measured and represented with standard methods. Therefore, an RDS provisioned for a multimedia streaming server could effectively achieve higher bandwidth efficiency than an RDS for general data traffic.

1.3 Contributions

The main contributions of the work presented in this dissertation are:

- This dissertation proposes an alternative solution to per-flow bandwidth reservation using aggregate bandwidth reservation to provide more consistent quality of service and circumvent the deployment hurdles in today's Internet. This new network service can be easily implemented with existing facilities in the backbone networks of the network service providers without drastic changes.
- This dissertation formulates the RDS configuration problem as a minimum concave cost network flow problem. The edge cost function in our problem formulation is a concave function that reflects the bandwidth economy of aggregation more accurately than a linear edge cost function.
- This dissertation presents an efficient approximation algorithm for the NP-hard RDS configuration problem. It produces solutions closer to an estimated lower bound with

much less time complexity than exhaustive search, and is suitable for large networks with hundreds of nodes.

- This dissertation evaluates our approximation algorithm using local search heuristics based on negative cost cycle and bi-cycle reduction. In addition to the traditional negative cycle reduction, the special subgraph structures of multi-cycles are discovered in a network with concave edge cost function. This discovery leads to the negative bicycle (extensible to multi-cycle) reduction algorithm.

- This dissertation formulates the configuration problem for multi-server RDSs similarly as a minimum cost network flow problem, and identifies that the key to the configuration problem is the server placement problem. A number of server placement are evaluated using simulations. A class of greedy server placement algorithms are found to produce the best solutions.

- This dissertation develops a configuration algorithm for redistribution subnetworks in a multi-server RDS, which improves fault tolerance by dynamically redirecting traffic flows from a faulty server to other servers.

- This dissertation demonstrates the potential end-to-end performance improvements in an RDS by proposing a source traffic regulation technique to resolve the unbalance bandwidth utilization problem. It shows that by leveraging the information of the underlying RDS, better end-to-end performance can be achieved. A number of regulation algorithms that can be implemented in various environments and platforms are proposed and evaluated.

1.4 Organization

The rest of the dissertation is organized as follows: the reserved delivery subnetwork (RDS) architecture is formally introduced in Chapter 2. In Chapter 3, the configuration problem for single server RDS is described. The problem is formulated as a minimum concave cost network flow problem, and an efficient approximation algorithm is presented and evaluated. In addition, the local search heuristics based on negative cost cycle and bicycle reduction is also investigated, and the results from simulation studies are presented. In Chapter 4,

the study is extended to deal with RDSs with multiple servers. The configuration problem is similarly formulated as a minimum concave cost network flow problem, however, the unknown server locations make the configuration for a multi-server RDS more complicated than the configuration for a single-server RDS. An study is described to evaluate a number of server placement algorithms in order to identify an good solution for multi-server RDS configuration. The improvements to fault tolerance to server failures in a multi-server RDS are also investigated, and a configuration algorithm for the redistribution subnetworks to redirect traffic for the faulty servers is presented. In Chapter 5, the source traffic regulation technique is presented to improve end-to-end performance by leveraging the underlying RDS. Chapter 6 concludes this dissertation with an outline of some future work that can expend from our work presented in this dissertation.

Chapter 2

Reserved Delivery Subnetworks (RDS)

This chapter formally introduces the reserved delivery subnetwork (RDS) as a new network service to provide more consistent service in today's Internet. We also outline a number of design issues related with the configuration and operation of reserved delivery subnetworks.

2.1 Formal Definition

A reserved delivery subnetwork (RDS) is a semi-private network infrastructure used by an information service provider to allow it to deliver more consistent performance to its customers. The *endpoints* of an RDS include a *source node* and a potentially large number of *sink nodes* distributed within a fixed network infrastructure. Sink nodes are typically routers within metropolitan areas where customers of the information service are found. A network provider selects a set of links within the network and dimensions bandwidth reservations on those links in order to accommodate expected traffic flows from the server to the various sink nodes. This allows traffic from the source node to flow through to the sinks without contention from other traffic sources, improving quality of service.

An example RDS is illustrated in Figure 2.1. In the backbone network of an information service provider, there is a server and a large number of users of the server information at different locations in the backbone network. An RDS (shown as the highlighted subnetwork) is set up to connect the server to these sink locations with user demands. For each location with user demands, there must be an RDS path from the source, and the reserved bandwidth on that path must be sufficient to satisfy the average total user demands of that location. The RDS must be set up in such a way that it can meet the demands of the

customers of an information service provider, and it utilizes the reserved link bandwidth efficiently such that more such services can be accommodated in the backbone network to maximize the service revenue of the network service provider.

2.2 RDS Configuration

There are two major tasks when we configure an RDS. The first task is to pick the links in the subnetwork such that there is a path from the server to all sink locations. The second task is to determine how much bandwidth should be reserved on all selected links. The provisioning of reserved bandwidth in an RDS is crucial its success. If insufficient bandwidth is reserved on a path from the server to a sink location, user demands at the location will not be fulfilled completely. On the other hand, if too much bandwidth is reserved on links, the bandwidth resource of the backbone network will not be efficiently utilized, driving up costs. Therefore, an optimal solution must reserve bandwidth in the most efficient way such that the sink overloading probability is minimized, and the network service provider's revenue is maximized for providing more RDS services.

Because we reserve aggregate bandwidth for a large number of bursty flows for a large number of users in an RDS, we must be able to handle traffic variance gracefully. To allow for variability in the traffic volume at sink nodes, reservations are dimensioned based on the mean and variance of the expected traffic. The mean and variance of sink traffic can be derived from long term traffic measurement and appropriate traffic modeling [19, 9, 54, 89, 10, 88, 68].

Links that carry large traffic volumes are generally more efficient than links that carry small traffic volumes, since the amount of bandwidth that must be reserved to accommodate traffic variability becomes a smaller fraction of the total as traffic volume grows. For example, if we take a closer look at the intermediate routers of the example RDS in Figure 2.1, we can notice that as traffic flows diverge to reach different sinks, the total reserved bandwidth on the "downstream links" will generally be larger than the reserved bandwidth on the upstream link (or links). This economy of aggregate bandwidth effect can be illustrated in the following example and analysis.

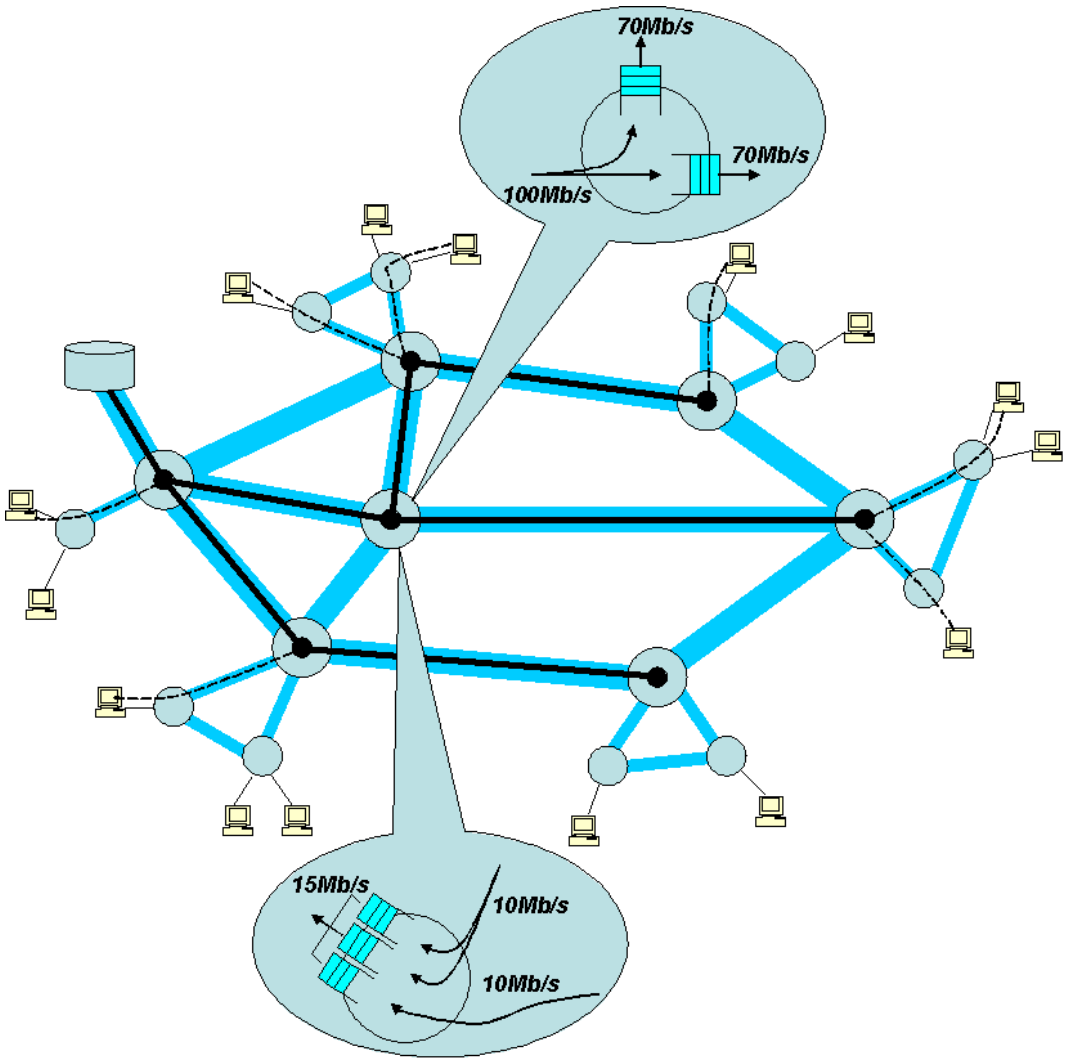


Figure 2.1: Reserved Delivery Subnetwork.

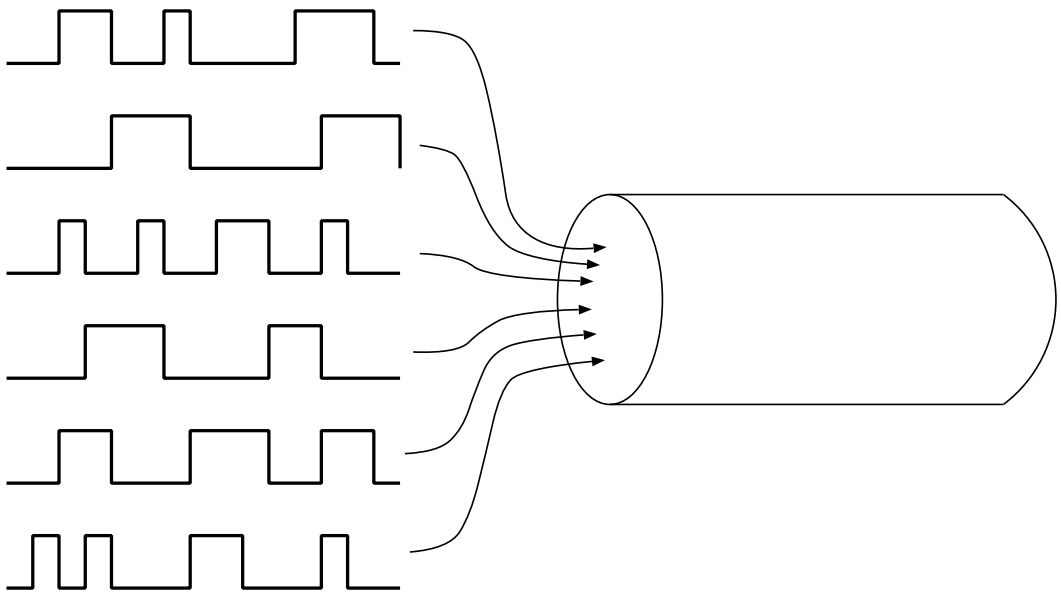


Figure 2.2: Aggregation of bursty flows.

Assume there is a large number of independent bursty on/off flows as shown in Figure 2.2, and each bursty flow source has a peak to average ratio of 25:1. We want to dimension the reserved bandwidth for these bursty flows on a link such that the overloading probability of the aggregate traffic is below 1%. We show the overloading probability of the aggregate flows of different sizes in Figure 2.3. When there are 100 such independent bursty flows, we must reserve at least 2.2 times the total average flow bandwidth to keep the overloading probability below 1%; while when there are $10,000$ independent flows, we only need reserve 1.14 times the total aggregate bandwidth to get the same overloading probability. It shows that we need almost twice as much as the reserved bandwidth per flow for the small flow aggregate (100 flows) as the larger flow aggregate ($10,000$ flows).

When there is a large number of statistically similar flows in an aggregate flow, we can treat the bandwidth of each individual flow as a independent random variable X_i and the bandwidth of the aggregate flow as another random variable X. $X = \sum_i X_i$. The mean μ and the standard deviation σ^2 of the aggregate flow grows linearly to the sum of the mean ($\sum_i \mu_i$) and the sum of the standard deviation ($\sum_i \sigma_i^2$) of the individual flows; thus, the variance of the aggregate flow σ grows as the square root to the sum of the standard deviation, $\sigma = (\sum_i \sigma_i^2)^{1/2}$. Because the majority of the aggregate bandwidth are within the range of the sum of the total average and some multiple (say, 3) of the variance, or $\mu + 3\sigma$,

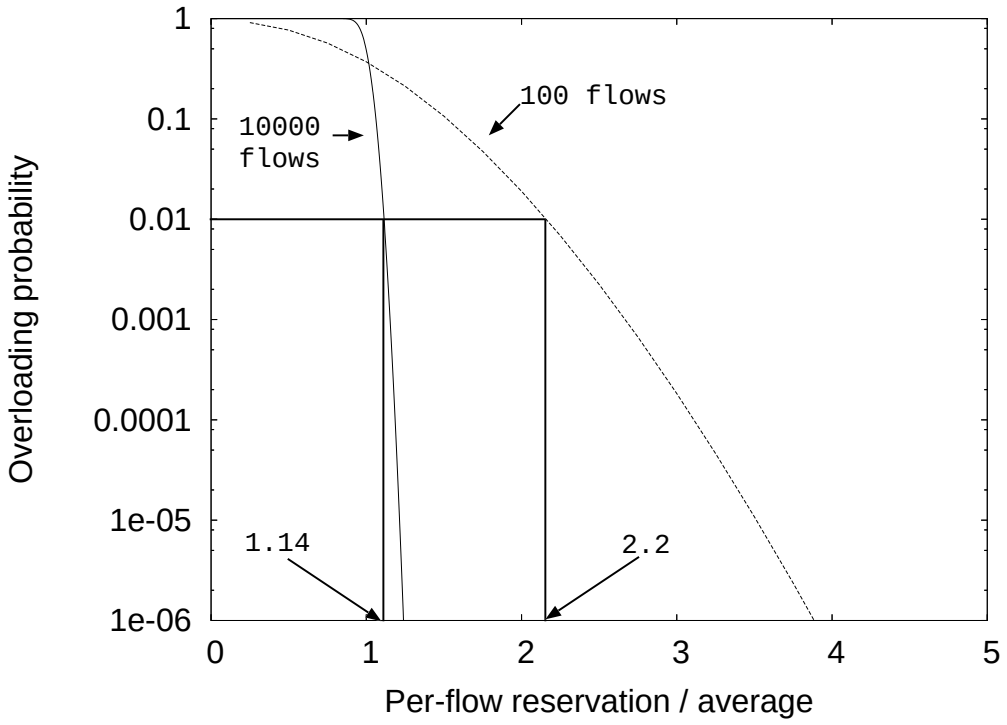


Figure 2.3: Bandwidth economy of aggregation.

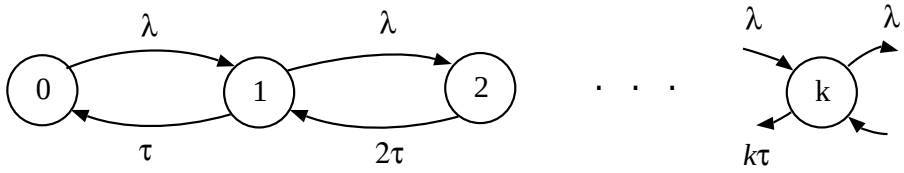


Figure 2.4: State transition diagram for the number of active flows on a link.

a reserved aggregated bandwidth of $\mu+3\sigma$ is sufficient to maintain a low overloading probability for the aggregate flow. Therefore, the reserved aggregate bandwidth that maintains a fixed overloading probability grows more slowly as the number of flows increases. This makes the necessary reserved aggregate bandwidth a concave function[1] of the number of flows.

If a link is viewed as carrying a large number of individual active data sessions, the dimensioning of the reserved bandwidth for the aggregate flow can be made based on random number of active sessions. Assume all active sessions arrive with exponential inter-arrival time with mean interval of $1/\lambda$, and sessions have exponential duration with mean duration of $1/\tau$, as illustrated in Figure 2.4. Assuming a standard $M/M/\infty$ queueing system, then the probability of k active sessions on a link is $p_k = (\rho^k/k!)e^{-\rho}$, where the average number of active sessions is $\rho = \lambda/\tau$. The variance $\delta^2 = (\sum_{k\geq 1} k^2 p_k) - \rho^2$. Substituting and expanding p_k, then

$$\begin{aligned}
\sum_{k\geq 1} k^2 p_k &= \sum_{k\geq 1} k^2(\rho^k/k!)e^{-\rho} \\
&= e^{-\rho}[\rho + \rho\sum_{k\geq 2} k\rho^{k-1}/(k-1)!] \\
&= \rho e^{-\rho}[1 + \sum_{k\geq 2}(k-1)\rho^{k-1}/(k-1)! + \sum_{k\geq 2}\rho^{k-1}/(k-1)!] \\
&= \rho[e^{-\rho} + \sum_{k\geq 1} k(\rho^k/k!)e^{-\rho} + \sum_{k\geq 1}(\rho^k/k!)e^{-\rho}] \\
&= \rho(1+\rho) \\
&= \rho + \rho^2
\end{aligned}$$

[1]The unit incremental value of a concave function $f(x)$ grows smaller as the value x increases. For $x_0 \leq x_1 \leq x_2$, $(f(x_1) - f(x_0))/(x_1 - x_0) \geq (f(x_2) - f(x_0))/(x_2 - x_0)$.

So, $\delta^2 = \rho$, and the standard deviation $\delta = \rho^{1/2}$. This result is consistent with the approximate standard deviation used for the aggregate flows. This result is from a specific source model, but it can apply to more generic source models such as pareto sources.

Because of this effect of bandwidth economy of aggregation, it is beneficial to group together flows going from the source to sinks that are close to one another as long as possible, even though this may cause traffic to follow a longer route than the shortest path.

The bandwidth economy of aggregation depends only on the independence of traffic in different flows. The effect of aggregation should not be confused with the issue of self-similarity and long range dependence of network traffic as described in [59, 48, 49]. The self-similarity characteristics of network traffic refers to the phenomenon that network traffic exhibits similar burstiness patterns over many different *time scales*, thus can not be properly modeled by a Poisson process. The self-similarity characteristics is not related to the bandwidth economy of aggregation, which shows the that the variance of aggregate traffic differs with *different sizes* of flow aggregates.

2.3 RDS Scalability

For large information service providers, such as CNN, their large number of users may be distributed in many vastly separated geographical areas. A single-server RDS may not be sufficient to serve such a user because of the cost of maintaining many high-bandwidth long-haul links and the reduced performance caused by the longer latency. To meet the demands of large number of customers in distributed locations, the service providers are motivated to install multiple servers at separate locations to reduce the cost and improve the end user quality of service. In this case, the information service provider must scale an ordinary RDS with a centralized server to one that has multiple servers. From the customers' perspective, a multi-server RDS reduces the transmission latency and hence increases the perceived quality of service. From the information provider's point of view, the additional server replicas eliminate the single point of failure in the RDS, and release the bandwidth tied up on the long haul connections from a single server to various remote locations. These benefits of improved quality of service and bandwidth efficiency can offset the cost of deploying the server replicas.

The configuration of a multi-server RDS is more complicated than the configuration of a single-server RDS. The complexity comes from two additional subproblems that are unique in multi-server RDS, namely, the server placement and sink partitioning. Server placement determines where should the replicated servers be placed, and sink partitioning decides which sinks should connect to which servers. Both server placement and sink partitioning are critical to the optimal configuration of a multi-server RDS.

2.4 RDS Fault Tolerance and Recovery

A multi-server RDS provides better quality of service with shorter latency and improved fault tolerance to single point failure than a single-server RDS. However, when a server fails or becomes overloaded, users with demands served by the affected server will still suffer from reduced service quality. To handle such a situation, a redirection subnetwork can be set up that allows other unaffected servers to take over the load on the affected server. The redirection subnetwork should be flexible to handle server overload on any server with minimum bandwidth overhead. In addition, the redistribution subnetwork should incur minimum communication overhead. The configuration problem of the redirection subnetworks for dynamic load redistribution in a multi-server RDS is another important issue for an information service provider.

2.5 RDS End-to-end Performance

An RDS can provide more consistent quality of service to users with exclusive access to reserved aggregate bandwidth for a large number of users. Besides the benefit of exclusive aggregate bandwidth access, there are other potentials to further improve the end-to-end performance in an RDS because the end hosts can utilize the knowledge about the underlying networks to achieve better performance than in the ordinary Internet. As Savage et al. [70] pointed out, the transport protocols in today's Internet are highly conservative, because they have to deal with the underlying network as a black box, and effectively probe the network repeatedly in order to determine a safe operating point. On the other hand, if some information about the underlying network is available, the end-to-end performance

can improve tremendously. We should be able to leverage such advantages to further improve the end-to-end performance in an RDS by enabling some forms of informed transport functionalities.

Chapter 3

Configuration of Single-Server Reserved Delivery Subnetworks

This chapter discusses the configuration of a basic single-server RDS. The RDS configuration problem is covered in three sections. Section 3.1 formulates the RDS configuration problem as a minimum cost maximum flow problem in a network with concave link costs, which reflects the bandwidth economy of aggregation in real network operations. Because the minimum concave cost network flow problem is an NP-hard problem, and the existing search-based exact algorithms are impractical for networks with hundreds of nodes, an efficient approximation algorithm with reasonably good solution quality is proposed in Section 3.2. The Largest Demands First (LDF) algorithm is described in this section, and its performance is studied using simulation. To further improve the solution quality and study the optimality of an algorithm for the RDS configuration problem, the application of local search heuristics is studied in Section 3.3. The traditional negative cost cycle reduction as well as a new negative bicycle reduction are used to improve the solutions obtained from LDF as well as other algorithms, and the improvements from the local search heuristics are studied. Section 3.4 summarizes this chapter.

3.1 Problem Formulation

In order to formulate the configuration problem for a single server RDS, we start with an elementary observation. If the traffic on a link consists of a large number of independent and statistically similar streams, the mean and the variance of the aggregate traffic scales

directly with the number of flows. So, we let $\sigma(\mu) = \alpha\mu^{1/2}$ denote the standard deviation of an aggregate traffic flow with mean μ, where α is a parameter. Note that when $\mu = \alpha^2$, $\sigma(\mu) = \mu$. That is, α^2 is the mean traffic rate for which the mean and standard deviation are the same. Given a traffic flow with mean μ and standard deviation $\sigma(\mu)$, a suitable choice for the reserved bandwidth is $\mu + k\sigma(\mu) = \mu + k\alpha\mu^{1/2}$, where k is a small constant (say 3). With these preliminaries, we can now proceed with a formal statement of the RDS configuration problem.

We are given a directed graph (or network) $G = (V, E)$ and two real-valued functions $l(\cdot)$ and $b(\cdot)$ defined on E. We refer to $l(e)$ as the *length* of edge e and $b(e)$ as its *bandwidth*. We also define a real-valued *edge capacity* $c(e)$, which represents the mean rate of the largest reservation that can be carried by edge e. The edge capacity satisfies the equation $c(e) + k\alpha c^{1/2}(e) = b(e)$ and is equal to $\left(-k\alpha + \sqrt{k^2\alpha^2 + 4b(e)}\right)^2/4$.

We are also given a *source node* $r \in V$ and a set of *sink nodes* $S \subseteq V$, with each sink node s having a mean demand $\mu(s)$. The minimum cost RDS that satisfies the mean demands, while respecting the capacity limits on the network links can be found by solving a minimum cost flow problem, in which the flow into each sink is given by its mean demand, and the total flow on each link e is bounded by $c(e)$. For an average aggregated flow of x, the cost of x on an edge e is defined to be $l(e)(x + k\alpha x^{1/2})$. The second factor in this expression corresponds to the amount of bandwidth that must be reserved to accommodate a flow of magnitude x. Note that the cost function is concave. Given a minimum cost flow that satisfies the demand, the optimal RDS is the subgraph of G defined by the edges with non-zero flows. The cost of the subnetwork is the sum of the costs of the flows on its edges.

In the minimum cost maximum flow problem, we seek a flow function f on the edges of the given network. For any node that is not a source or a sink, the sum of the flows on the incoming edges must equal the sum of the flows on the outgoing edges. The flow must satisfy the given capacity constraints on the edges and must satisfy the given demands required by the sinks. Among all such flows, we seek one of minimum cost. For each edge (u, v) in the original graph, the residual graph has an edge (u, v) if $f(u, v)$ is less than the capacity of (u, v) and it has edge (v, u) if $f(u, v)$ is greater than zero. The *residual capacity* of the edge (u, v) is the difference between the capacity and the current flow. The residual capacity of (v, u) equals $f(u, v)$. An augmenting path is just any path in the residual graph from the source to a sink on which more flow can be added. For any edge e in the original

graph, the cost of carrying x units of flow on e is $l(e)(x + k\alpha x^{1/2})$. We let $\delta_f(e, \Delta)$ be the change in cost caused by adding Δ units of flow on the edge e in the residual graph, assuming that Δ is no larger than the residual capacity of e. If Δ is larger than the residual capacity, $\delta_f(e, \Delta)$ is defined to be infinite. We refer to $\delta_f(e, \Delta)$ as the *incremental cost* of the edge e, with respect to the increment Δ. The incremental cost of a path, with respect to an increment Δ, is defined as the sum of the incremental costs of its edges. For any flow and increment Δ, we can define a tree $T_f(\Delta)$, which is a shortest path tree rooted at the source in the subgraph of the residual graph defined by the edges with residual capacity no smaller than Δ. The path costs in T are defined with respect to the incremental costs, $\delta_f(e, \Delta)$. As Δ is increased from zero, we get a finite sequence of trees $T_0, T_1, \ldots, T_m$. For each tree T_i in this sequence, there is a corresponding range R_i of values of Δ. The *incremental cost per unit flow* of an augmenting path p is $\delta_f(p, \Delta)/\Delta$, where Δ is the amount of flow needed to saturate p.

Note that when there are no limits on edge capacities, the best RDS is always a tree. We expect that in practice, network link capacities will often not be a limiting factor, so that the best RDS may typically be a tree. Even when link capacities are limited, we may wish to constrain the form of the solution so that all traffic going to a single sink is constrained to use the same path, in order to simplify the routing of the traffic (note that in this case, the RDS need not be a tree).

3.2 Largest Demand First (LDF) Algorithm

3.2.1 Algorithm Design Issues

As we noted previously, the edge cost function is a concave function of the currently carried amount of flow. Thus, when we aggregate more flows on a link, the over-provisioned bandwidth, that is necessary to accommodate traffic variations, decreases, resulting in more cost efficient networks. So, we prefer a configuration algorithm that rewards flow aggregation. However, it is possible that if we favor aggregation too strongly, longer paths may be selected while shorter and cheaper routes exist. Thus, we need also to restrict the path selection within a reasonable region.

When we select a path from the root to a sink, we can either keep all traffic to the sink on a single path, or split it among a number of paths leading to the sink, some of which may not have enough capacity for the sink by themselves. The concave edge cost function suggests that keeping the traffic flows together is more cost efficient than splitting them. However, such a strategy is not always able to satisfy all sinks in networks with limited link capacities, which leads to higher demand blocking ratio (the ratio of unmet demands to the total demands) than an algorithm that splits flows. Therefore, when we design an RDS configuration algorithm, we need to consider the tradeoff of flow splitting and aggregation, and try to reduce the cost while minimizing the possibility of sink blocking.

3.2.2 Algorithm Description

One of the classical methods for solving minimum cost flow problems is the minimum cost augmenting path method. This method iteratively selects a *minimum cost augmenting path* from the source to a sink that has unmet demand and adds flow along that path until either the demand has been satisfied or the capacity limit of some edge on the path has been reached. While this method can find an optimal flow when the cost per unit flow on each edge is constant, it cannot be directly applied to the RDS configuration problem, since the relative costs of two different paths can change depending on the magnitude of the flows added to those paths. That is, it may cost less to add x units of flow to a path p than to an alternative path q, but it may cost more to add $2x$ units of flow to p than to q.

Although we cannot use the minimum cost augmentation algorithm directly in the RDS configuration problem, we can apply similar ideas to construct an approximation algorithm that does not require an enumerative search of the problem space. In the minimum cost augmenting path algorithm, at each step we choose an augmenting path from the source to the sink in the *residual graph* for the current flow. It is well known [2] that when the cost per unit flow is constant, we can construct a minimum cost flow by finding a succession of minimum cost augmenting paths and *saturating* each one in turn (that is adding as much flow to the path as allowed by the capacity constraints, or the unmet demand at the sink, whichever is smaller). To apply the minimum cost augmentation strategy to the RDS problem, we seek an augmenting path from the source to a sink that has the smallest *incremental cost per unit flow* among all augmenting paths. In principle, this can be done

by constructing each of the distinct shortest path trees and selecting the best augmenting path found in all the trees. A computationally simpler alternative is to choose a small set of increments, construct the tree corresponding to each increment, and find the best augmenting path from among this smaller set of trees. While this only "samples" the set of trees, and hence will not always find the best path, it does at least approximate the minimum cost augmentation strategy. There are various strategies to select the set of increments. Because our goal is to schedule flows to the sinks, we should select increments related to the sink demands. In order to make such a selection, we can order the sinks in a specific order, and use the remaining unmet demand as the increment values (Δ). If a path is found within $T_f(\Delta)$, we can then augment the flow along the path. Note that the flow augmentation can also be implemented with various strategies, resulting in RDSs with different costs. The following pseudo code shows the generic framework of our algorithm. Depending on the sink sorting and path augmentation strategies, different algorithms can be obtained.

Order the sinks $s_1, \cdots, s_m$ according to a certain sorting strategy
for $i \in [1, m]$
    Augment flow to satisfy demand to s_i with a certain augmentation strategy
end

Each iteration of the algorithm requires the computation of a shortest path tree and possibly a *bottleneck shortest path tree*. Both of these computations can be implemented to run in $O(m + n \log n)$ time, where m is the number of edges and n the number of nodes.

The *Largest Demand First* (LDF) algorithm orders the sinks by their demands such that for sink $s_i \in \{s_1, s_2, \cdots, s_m\}$, $\mu_i \geq \mu_{i+1}$. LDF establishes paths to the sinks with the largest demands first. Therefore, the flows on existing paths are large, and the cost benefits of sharing a path to the root by subsequent sinks are high. In networks with ample link capacity, each iteration fully satisfies the demand at some sink, so the number of iterations equals the number of sinks. However, in networks with limited capacity, it is possible that some sink demands will not be satisfied after the same number of iterations.

The Single Flow Augmentation (SFA) algorithm always tries to augment a flow to a sink in a single path. If no such path can be found while there is still unmet demand, then the algorithm fails. In networks with ample link capacity, each iteration fully satisfies the demand at some sink, so the number of iterations equals the number of sinks. This leads

to an overall running time of $O(s(m + n \log n))$, in the case of ample link capacities. For arbitrary link capacities, the number of iterations still equals the number of sinks, but there are sinks whose demands cannot be satisfied, resulting in blocking situations. In addition, SFA results in lower cost network when the link capacity is not a limiting factor because it avoid the "penalty" of splitting flows. The obvious drawback is blocking in more congested networks.

3.2.3 Evaluation

To evaluate the LDF algorithm we compared the cost of the solution produced to that of an easily computed lower bound. The lower bound is computed by sorting the sinks in increasing order of their distance from the root and then assuming that each sink is reached by a path of this minimum length, and that the path can be shared with all sinks at greater distances from the root. We evaluated the algorithm on two networks. The first is a 15×15 torus (each node is connected to four neighbors forming a rectangular grid with "wrap-around edges" linking the top and bottom rows and the leftmost and rightmost columns). Link lengths were uniformly distributed, with the longest links being ten times longer than the shortest. The demands for the sinks were uniformly distributed, all with the same mean demand.

The second network, shown in Figure 3.13, includes a node at each of the fifty largest metropolitan areas in the United States; the link lengths were chosen to be equal to the geographic distances between the locations, and the demands were chosen to be proportional to the populations of the metropolitan ares. The locations of sources and sinks were selected randomly, with every node having the same probability of selection. For the results reported here, unbounded link capacities were used in both networks. An example RDS computed by the LDF algorithm is shown in Figure 3.1. The source for this example is in Chicago and there are ten sinks at various locations around the country (the sinks are designated by small squares on the map). The cost of this solution is about 1.34 times the cost of the lower bound.

Figure 3.2 shows how LDF performs on the torus. The first chart shows the ratio of the cost of the solution produced by LDF to the lower bound, as the number of cities increases from 1 to 50, while α^2 is fixed so that $\sigma(D) = D$, where D is the average demand per sink. Each

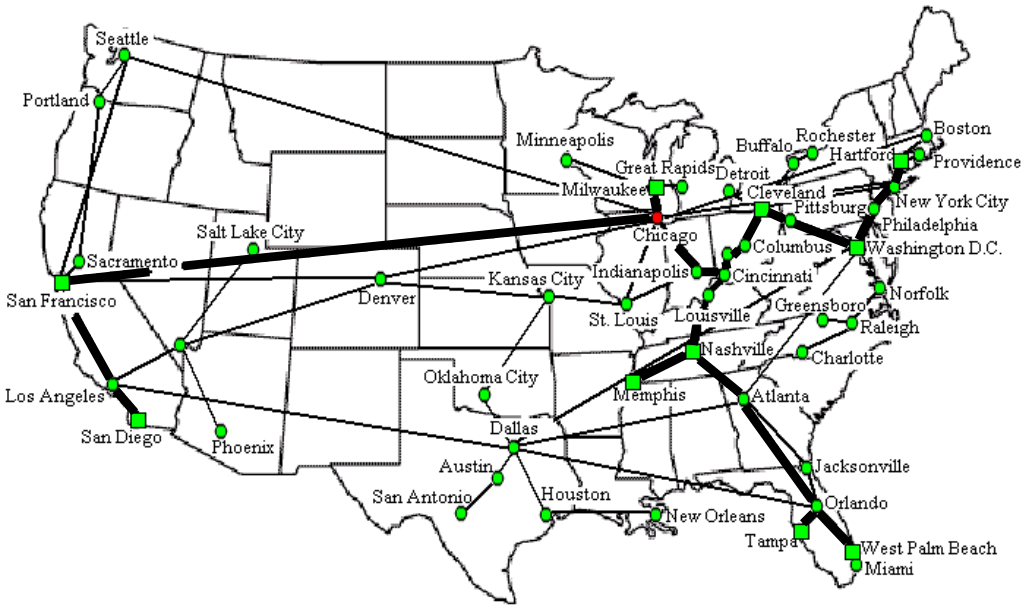


Figure 3.1: Example RDS computed by the LDF algorithm

data point represents the average of results from 100 independent problem instances. For large numbers of cities, the LDF algorithm produces solutions costing no more than about 1.6 times the lower bound. The curves labeled LB*(2), LB*(3) and LB*(4) are related to the lower bound and provide evidence (although no proof) that for larger numbers of cities the lower bound is fairly loose. LB*(2) is computed by first dividing the sinks into two sets, those to the “left” of the source and those to the ”right” of the source. Each of these subsets is then sorted by distance from the source and each node is assumed to share its path to the source with all nodes in the same subset that are at greater distance from the source. LB*(3) (and LB*(4)) is computed similarly, by first dividing the sinks into three (respectively four) sets of nodes defined by “pie-shaped” regions centered on the root, then sorting the subsets by distance from the root and assuming the maximum possible sharing of paths among nodes in the same set. For larger numbers of randomly distributed cities, it’s reasonable to expect LB*(2), LB*(3) and LB*(4) to be no larger than the cost of an optimal solution, although they do not constitute true lower bounds. Note that for 50 sinks, LDF produces solutions that average about 1.3 times LB*(3).

The second chart in Figure 3.2 shows how the performance of LDF varies in comparison to the lower bound as α^2 is varied so that $\sigma(D)/D$ varies from .2 to 5, while the number

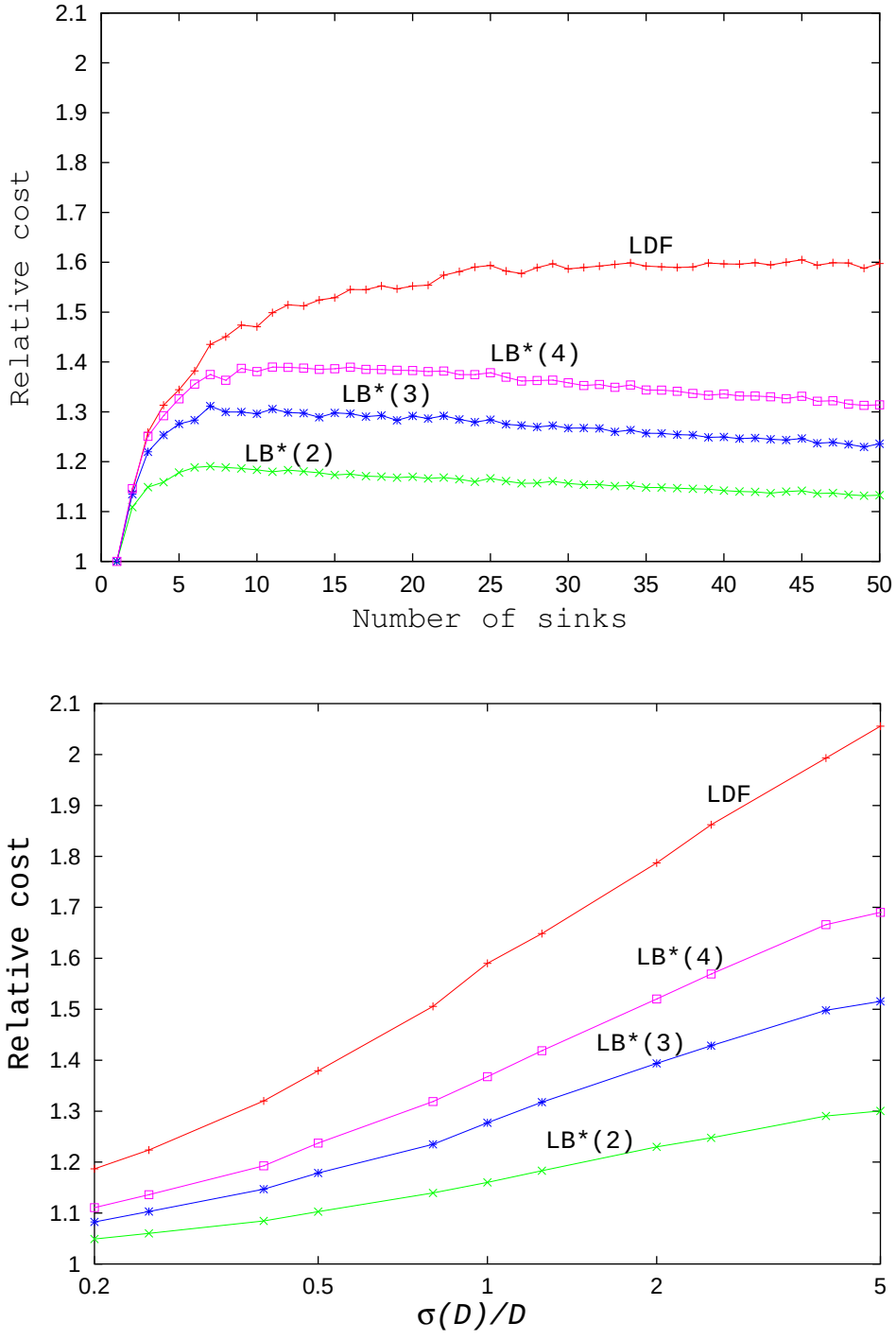


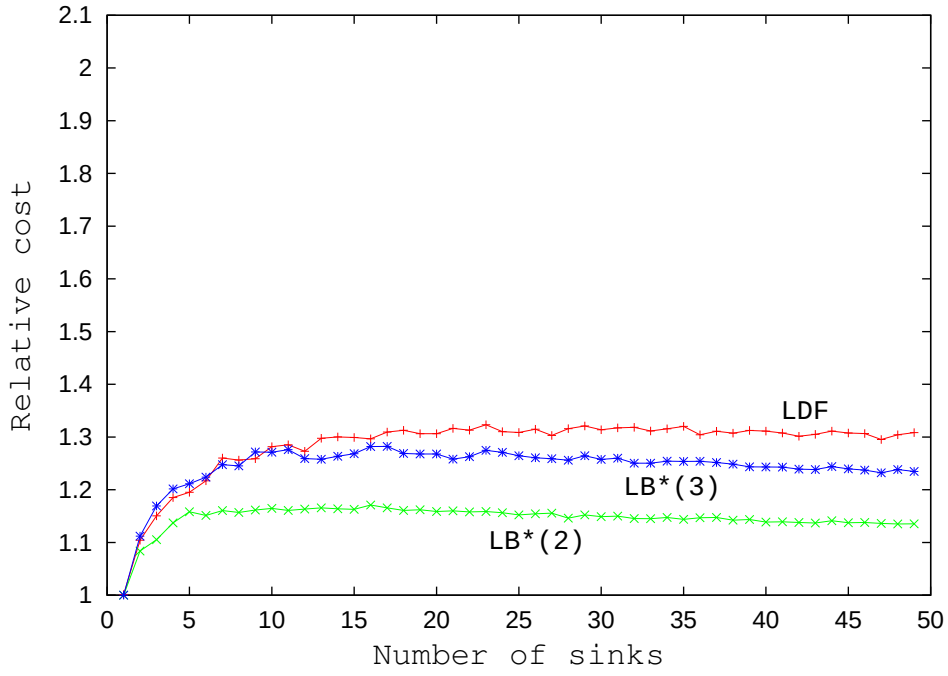
Figure 3.2: Performance of LDF on torus network

of sinks is fixed at 25. For small values of $\sigma(D)/D$, there is less to be gained from sharing paths, so LDF performs better, relative to the lower bound. For larger values of $\sigma(D)/D$, there is much more to be gained by sharing paths, so the gap between the lower bound and LDF gets larger. When $\sigma(D)$ is five times the average demand per sink, the cost of the solutions produced by LDF increases to about 2.05 times the lower bound.

Figure 3.3 shows how LDF performs on the national network. We note that LDF performs generally better in this case, than for the torus, but the general character of the results remains the same. We speculate that the improved performance arises largely because the national network spans a greater east-west distance than north-south, and that the large numbers of cities are near the coasts meaning that often the root is near one of the coasts, which makes it relatively easy for LDF to produce solutions with large amounts of sharing. The wide variance in the link lengths in the torus network may also contribute to the reduced performance in that case (some links in the torus network violate the triangle inequality, preventing them from being used in any solution).

3.3 Improving Solution Quality with Local Search Algorithms

The configuration problem for a single-server RDS can be conveniently formulated as a minimum concave cost network flow problem (MCCNFP) as described earlier in this chapter. However, it is well known that MCCNFP is NP-hard [30], and the existing exact algorithms are all search-based algorithms with some intelligent enumeration methods [31, 32]. However, these algorithms do not scale well for networks with even moderate numbers of nodes, and thus are impractical in real applications. In order to provide solutions for MCCNFP in practice, a number of approximation algorithms have been studied and proposed. Among these approximation algorithms, local search algorithms for MCCNFP has enjoyed tremendous success in solving large and complex problems in practice. Given an existing solution, a local search algorithm examines the "neighborhood" of the existing solution, and identifies a solution that is locally optimal within the "neighborhood". The "neighborhood" is defined as a set of solutions that are reachable from an existing solution with a simple operation. In the case of MCCNFP, it is known that the optimal solution is an



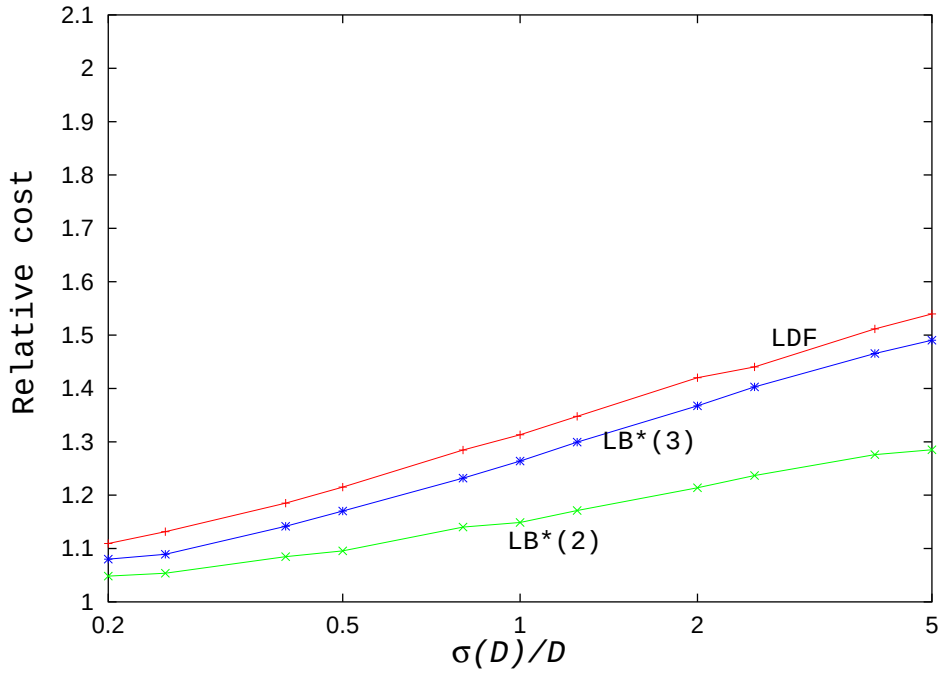


Figure 3.3: Performance of LDF on national network

extreme flow, which is a tree in an uncapacitated network. By taking advantage of this property, a local search algorithm finds a local optimal solution from an extreme flow by examining the adjacent extreme flows reachable from the existing extreme flow with a simple operation, although it may be trapped in a local optimal solution different from a global optimal one.

In the rest of this chapter, the existence of negative cost multi-cycles is observed in a network with concave edge costs. That is, even though there is no negative cost cycle, there could exist a set of cycles with a common path that has a negative total cost. Based on this observation, the cycle reduction algorithm in [27] is not able to include all adjacent extreme flows, and therefore is limited and incomplete. Towards this end, an improved local search algorithm is proposed with bicycle reduction method to consider both negative cost single cycles and bicycles. Both the original and improved local search algorithms are applied to networks with a simple concave edge cost function in our experiments, and demonstrate the improvement of solution quality. Although we focus on negative cost bicycles in this chapter as they are the most likely negative cost multi-cycles, we also show that the bicycle reduction algorithm can be generalized to handle other negative cost multi-cycles too.

Section 3.3.1 briefly discusses the local search algorithms using cycle reduction strategy, and explains why a naive cycle reduction approach fails in a network with concave edge costs. The local search algorithm with cycle reduction method proposed by Gallo and Sodini [27] is reviewed in Section 3.3.2. We also describe a path compression technique to the original algorithm, reducing the number of shortest path trees computations. We illustrate in Section 3.3.3 that how a local minimum can be sub-optimal because of the existence of negative cost bicycles. In Section 3.3.4, we describe the improved local search algorithm with bicycle reduction to identify and remove negative cost bicycles. Section 3.3.5 outlines the simulation environment and analyzes the simulation results. Section 3.3.6 discusses the more general case of negative cost multi-cycles and generalizes the bicycle reduction algorithm to handle negative cost multi-cycles.

3.3.1 Local Search Algorithms Using Cycle Reduction Strategy

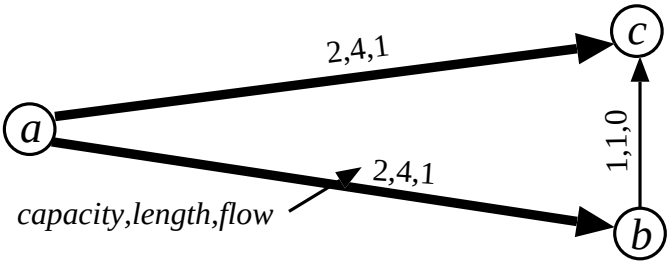
Local search [1] is a well-known approximation method that is applicable to almost all combinatorial optimization problems. Although it can not determine if the best solution

found so far is optimal, local search has enjoyed tremendous success in solving large and complex combinatorial optimization problems in practice. When a local search method is applied to a problem, a simple operation is employed to transform an existing feasible solution to a neighboring feasible solution, and a neighboring solution with lower cost is chosen and further explored until no further improvement can be made. For minimum cost network flow problems, a local search method searches for a flow with the least cost among all neighboring feasible flows obtainable from an existing feasible flow with a simple operation. As for the choice of the simple operation in a local search algorithm, the negative cost cycle reduction method is a natural candidate. The negative cost cycle reduction works by "pushing" flows along a negative cost cycle to transform an existing feasible flow to another feasible flow with lower total cost. This operation is exactly what we expect for a local search algorithm. In addition, the negative cost cycle reduction method can form the basis of efficient algorithms for the minimum cost network flow problems in networks with linear edge costs [2].

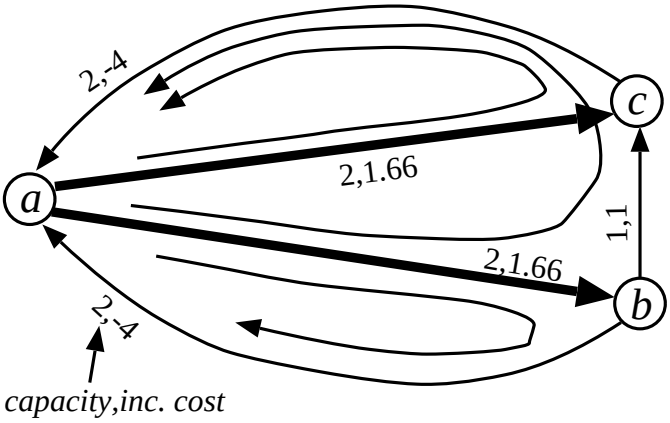
However, we must be careful when we apply the negative cost reduction method in a network with concave edge costs. The difficulty of applying negative cost cycle reduction in networks with concave edge costs can be illustrated in an example in Figure 3.4. We show a small network in Figure 3.4(a) with the capacity, length, and current flow of each edge in the labels. In this simple network, a is the source vertex that supplies the sink vertices b and c. Currently, there is a unit flow on edges (a, b) and (a, c). For this example, we adopt a simple concave edge cost function: the cost $c_e(\mu)$ of an edge e with an average flow of μ is defined as $c_e(\mu) = l\mu^{1/2}$, where l is the length of the edge. Thus, the existing flow has a total cost of 8. The incremental cost on $e = (u, v)$ with a flow increment of Δ is

$$\Delta c_e(\Delta) = \begin{cases} l(\sqrt{\mu + \Delta} - \sqrt{\mu}) & \text{if there is no flow on reverse edge } (v, u) \\ l(\sqrt{\mu - \Delta} - \sqrt{\mu}) & \text{if there is flow on reverse edge } (v, u), \text{ and } \Delta < \mu \\ l(\sqrt{\Delta - \mu} - \sqrt{\mu}) & \text{if there is flow on reverse edge } (v, u), \text{ and } \Delta \geq \mu \end{cases}$$

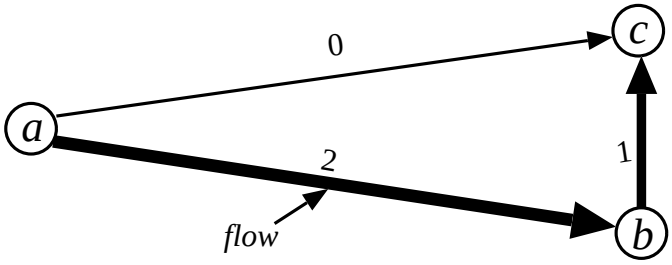
Figure 3.4(b) shows the corresponding residual graph of the current flow when a unit of flow will be changed on all edges. The incremental costs for a unit flow increment are shown on the labels of Figure 3.4(b). Clearly, Figure 3.4(b) has three negative cycles: namely, $\{(a, c), (c, a)\}$, $\{(a, b), (b, a)\}$, and $\{(a, b), (b, c), (c, a)\}$ with cost of -2.34, -2.34 and -1.34, respectively. With the negative cost cycle reduction method, we attempt to redirect



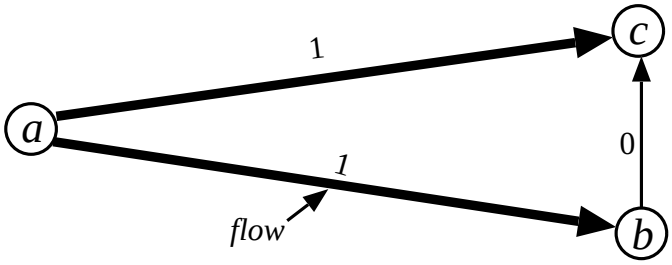
(a) Example network.



(b) Residual network with incremental costs.



(c) A good choice of negative cost cycle.



(d) A bad choice of negative cost cycle.

Figure 3.4: Problems of negative cost cycle reduction in a network with concave edge costs.

flow along a negative cost cycle such that the negative cost cycle is removed and the total cost is lowered after the flow redirection. If we choose the $\{(a,b),(b,c),(c,a)\}$ cycle, and push a unit flow along it, we could get a new flow with a lower cost of 6.656, and there is no more negative cost cycle in the residual graph (Figure 3.4(c)). However, if the $\{(a,c),(c,a)\}$ cycle is picked, and flow is redirected this cycle, it would neither remove the negative cost cycle, nor lower the total flow cost (Figure 3.4(d)). In fact, any edge in an existing flow has a two-edge negative cycle in the residual graph in such a network with concave edge costs. If any of such two-edge negative cycle is chosen by the cycle reduction algorithm, the local search algorithm is stalled. This is caused by the concavity of the edge cost function because on such an edge the absolute incremental costs of increasing and decreasing the same amount of flow are different, causing the asymmetric incremental costs and a "false" negative cycle in the residual graph. In contrast, in a network with linear edge costs, the absolute incremental costs of the two opposite edges are the same, but with different signs. So, there is no negative cost cycle with only two edges in a network with linear edge cost. Thus, we can not implement negative cost cycle reduction in a local search algorithm if we can not distinguish two-edge negative cost cycles from other legitimate negative cost cycles. In particular, we can not pick an arbitrary negative cost cycle and push flow along it in a network with concave edge costs. There are a number of efficient negative cycle reduction algorithms for minimum cost flow problems, such as the Minimum Mean Cycle Canceling algorithm [29] and the most helpful cycle canceling algorithm [4]. However, because they provide no efficient way to characterize the two-edged negative cycles that we want to avoid, these negative cycle reduction algorithms are not good candidates for local search for MCCNFP.

3.3.2 Local Search Algorithm with Cycle Reduction

Gallo-Sodini Cycle Reduction Algorithm

The local search algorithm for uncapacitated networks presented in [27] provides an effective way to implement negative cost cycle reduction that is more efficient than an algorithm that searches for all negative cycles. An extreme flow in an uncapacitated network is a feasible flow in a network for which the edges with non-zero flow constitute a tree with the source vertex at the root of the tree and all sink vertices at the leaves. The Gallo-Sodini

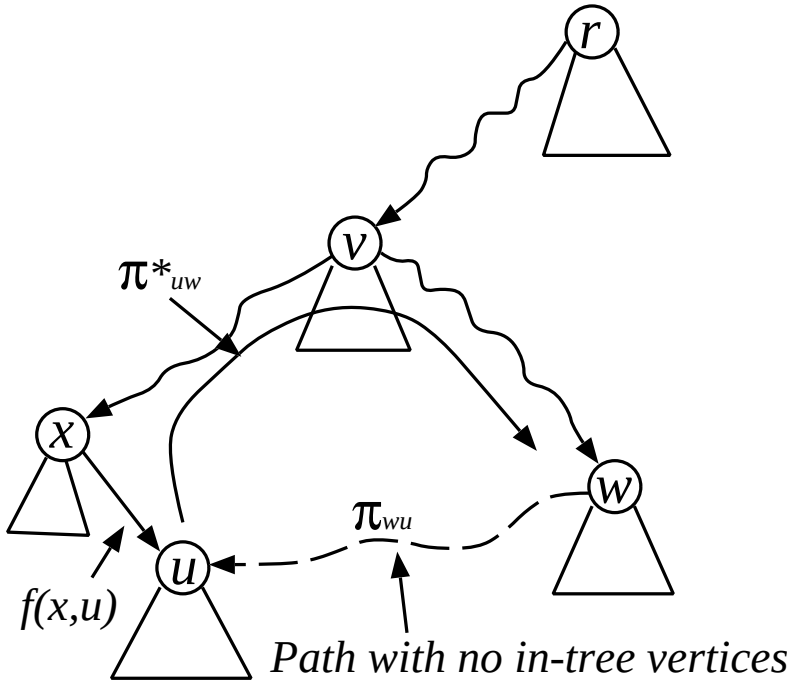
cycle reduction algorithm is based on the idea that an extreme flow x' is adjacent to an existing extreme flow x if and only if all edges that are in x' but not in x constitute a path connecting only two vertices in x. Therefore, for each pair of vertices in an existing extreme flow, the undirectional path between the two vertices and a path consisting only of edges not in the existing extreme flow form a cycle. If the flow created by redirecting flow between the two vertices along this cycle has a lower cost than the original flow, this cycle is a negative cost cycle. A simple but inefficient way to find negative cost cycles in an extreme flow is to check all pairs of vertices individually, which is very slow. The Gallo-Sodini algorithm provides a quick and systematic way to find a negative cost cycle without a complete enumeration of all possible cycles.

Figure 3.5 and Figure 3.6 illustrate the basic operations of the Gallo-Sodini algorithm for finding a neighboring extreme flow from an existing extreme flow. The original extreme flow f is shown as a tree in Figure 3.5(a).

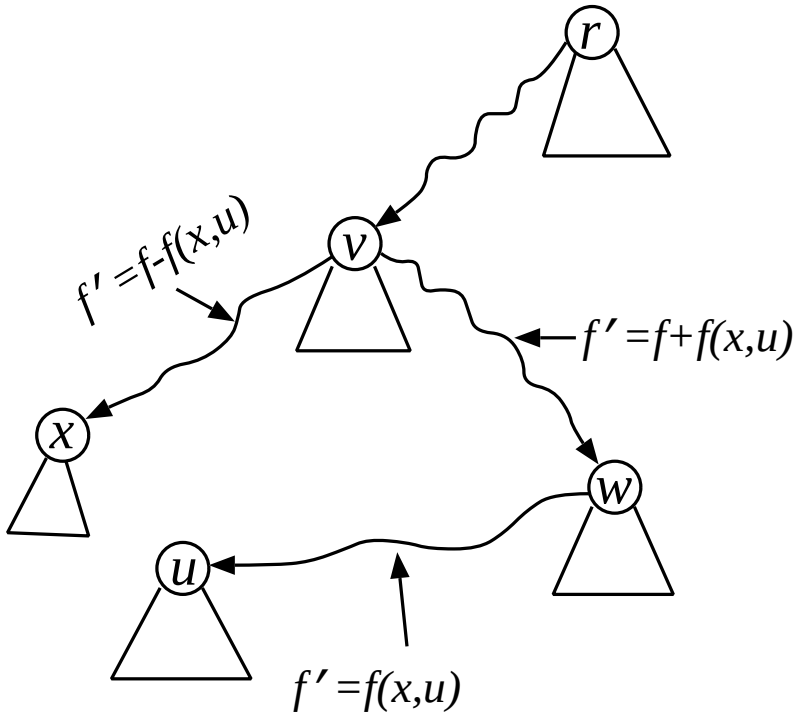
One vertex in the existing flow is processed in each iteration of the algorithm, denoted as the current vertex. First, assuming the existing flow into the current vertex is completely redirected, the incremental cost to another target vertex in the existing flow is computed as if the flow is redirected along the undirectional path from the current vertex to that target vertex.

In the specific example in Figure 3.5(a), for a current vertex u with an incoming flow of $f(x,u)$ in the tree defined by the existing flow, first find the undirectional path π^*_{uw} from u to any other tree vertex w that is not in the subtree rooted at u. Notice that π^*_{uw} could be composed of two directed paths: one from the nearest common ancestor c of u and w to u, and the other from c to w. Compute the incremental cost c^*_w of adding $f(x,u)$ units of flow on π^*_{uw} as $c^*_w = cost_f(-f(p_f(u),u), path_f(u,w)) + cost_f(f(p_f(u),u), path_f(w,u))$, where $cost_f(x,p)$ is the incremental cost of adding x units of flow along path p, relative to existing flow f, $p_f(u)$ is the parent of u in the tree defined by f, and $path_f(u,w)$ is the path from the nearest common ancestor of u and w to u in the tree defined by f. For example, the path from u to w (π^*_{uw}) is shown in Figure 3.5(a) as well as a directed non-tree path π_{wu} from w to u that connects two tree vertices w and u.

For the current vertex, although there is only one undirectional path from it to another vertex in the existing flow, there is a large number of possible paths between them outside



(a) Original extreme flow.



(b) The neighboring extreme flow obtained.

Figure 3.5: Original cycle reduction algorithm.

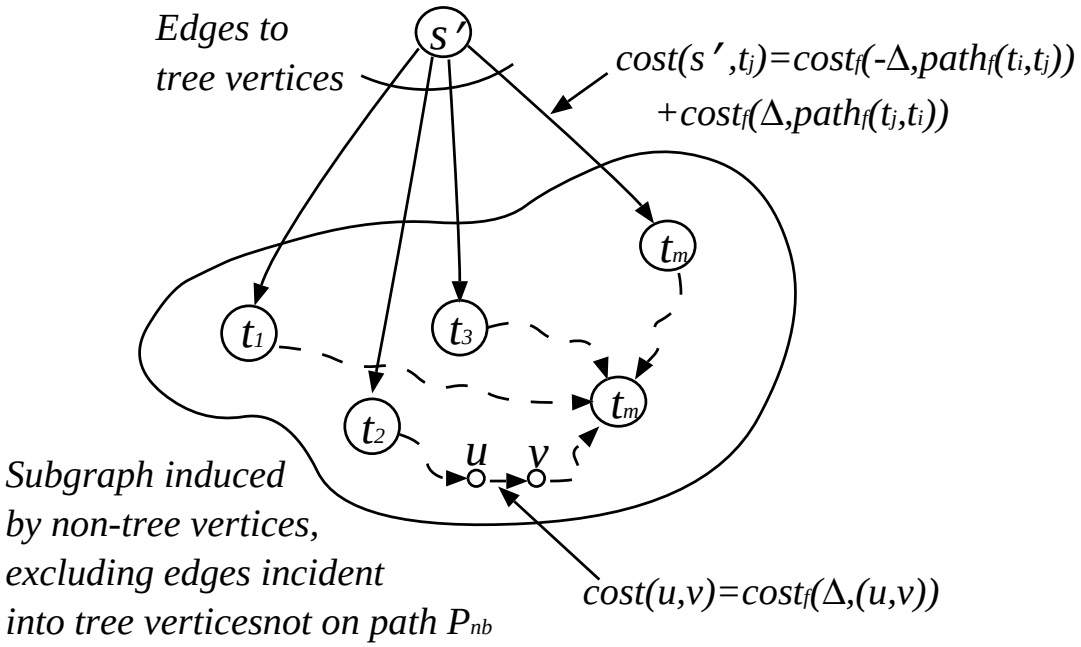


Figure 3.6: Finding the best solution for "target" t_i, where $cost_f(x, p) =$ the incremental cost of adding x units of flow along path p, relative to existing flow f, $\Delta = f(p_f(t_i), t_i)$ is the flow into t_i, $p_f(u) =$ the parent of u in the tree defined by f, and $path_f(t_i, t_j) =$ the path from the nearest common ancestor of t_i and t_j to t_i in the tree defined by f. Note, for the original cycle reduction algorithm, P_{nb} is only the vertex t_i. For the improved algorithm with compressed paths, P_{nb} is the longest "non-branching" in-tree path to t_i.

the existing flow. It is highly inefficient and slow to exhaustively check all these paths individually for the possible negative cost cycles. Instead of a complete enumeration of all possible cycles, a shortest path tree is built using the incremental cost computed in the previous step to quickly determine the least incremental cost cycle, which is a negative cycle if there is one.

In the specific example in Figure 3.5(a), a new network is derived for a specific tree vertex t_m as shown in Figure 3.6. In this transformed network, a pseudo source vertex s' is introduced, and s' connects to every tree vertex t_j defined by the existing flow with a direct edge (s', t_j), except for t_i. All original tree edges are removed, and all non-tree edges incident to any existing tree vertex except t_i are removed too. It we define $\Delta = f(p_f(p_f(t_i), t_i)$ to be the flow into t_i, an edge (s', t_j) is assigned a length of $cost(s', t_j) = c_{t_j} = cost_f(-\Delta, path_f(t_i, t_j)) + cost_f(\Delta, path_f(t_j, t_i))$, and a non-tree edge (u, v) is assigned a length $cost(u, v)$ the same as the incremental cost of adding Δ units of flow on that edge, $cost(u, v) = cost_f(\Delta, (u, v))$. We then find the shortest path from s' to t_i in the transformed network. After the shortest path is determined, the last vertex w on the path from s' to t_i is identified, and Δ units of flow is redirected along the undirectional path $\pi^*_{t_i w}$ and then along the directed path π_{wt_i}. The resulting flow is a neighboring extreme flow to the original flow, as shown in Figure 3.5(b). If the modified flow has a lower cost than the original flow, the above procedure is repeated to find a lower cost neighboring flow of the new flow. Otherwise, the original flow is restored.

The Gallo-Sodini cycle reduction algorithm can be briefly described by the following pseudo code:

Find an extreme flow x^0.
Repeat
  For each tree vertex t_i with an incoming flow of Δ.
    For each tree vertex $t_j \neq t_i$,
      Compute the incremental cost $c^*_{t_j}$
        for redirecting Δ units of flow
        along the undirectional tree path $\pi^*_{t_i, t_j}$.
    For each non-tree edge (u, v),
      Compute incremental cost c_{uv} of adding Δ units of flow.
    Let $G' = (V', E')$ be the subgraph induced by non-tree edges.

$V' \leftarrow V' \cup \{s'\}$,
$E' \leftarrow E' \cup \{(s', v) | v$ is a tree vertex, and $v \neq t_i\}$
$-\{(w, v) | v$ is a tree vertex, and $v \neq t_i\}$,
For each edge $(s', t_j) \in E'$, assign a length of $c^*_{t_j}$,
for any other edge $(u, v) \in E'$, assign a length of c_{uv}.
Find the shortest path from s' to t_i in G'.
Let edge (s', w) be on the shortest path from s' to t_i.
Redirect Δ units of flow along $\pi^*_{t_i,w}$ and π_{w,t_i}, and obtain an updated flow x'.
If the updated flow x' has a higher cost than the current flow,
restore the original flow.
until no flow with lower cost can be obtained.

Note that this algorithm transforms the flow to the first improved neighboring extreme flow found. An alternative is to check all neighboring extreme flows and then transform to the best neighboring extreme flow. However, as suggested by the empirical results in [31], transforming to the first improved neighbor generally requires $25-40\%$ fewer shortest path computations than the best neighbor algorithm, and yields results of comparable quality.

Complexity Analysis Let n and m be the numbers of vertices and edges in the network, for each flow, the cycle reduction algorithm may need to check $O(n)$ vertices before it can determine if a neighboring extreme flow with lower cost exist [27]. For the tree defined by an existing flow with $k(k \leq n)$ vertices, we need to solve k nearest common ancestor problems, each taking $O(k)$ time. We also need to compute $2k^2$ incremental path costs. In addition, we need make k shortest path tree computation. The checking procedure is dominated by the single source shortest path computation. If we denote $S(n, m)$ as the time complexity of a single source shortest path algorithm in a graph with n vertices and m edges, then the time complexity for finding a neighboring flow with lower cost in the cycle reduction algorithm is $O(nS(n, m))$, or $O(n(n + m) \log n)$ if the single source shortest path algorithm is implemented efficiently.

It is easy to see that the cycle reduction algorithm reduces the cost by redirecting flow along negative cost cycles. This local search algorithm can greatly improve the quality of solutions obtained from LDF. Take the network in Fig. 3.7 for example. The source s connects to all n sinks with unit demand. LDF picks only the direct links from s to all

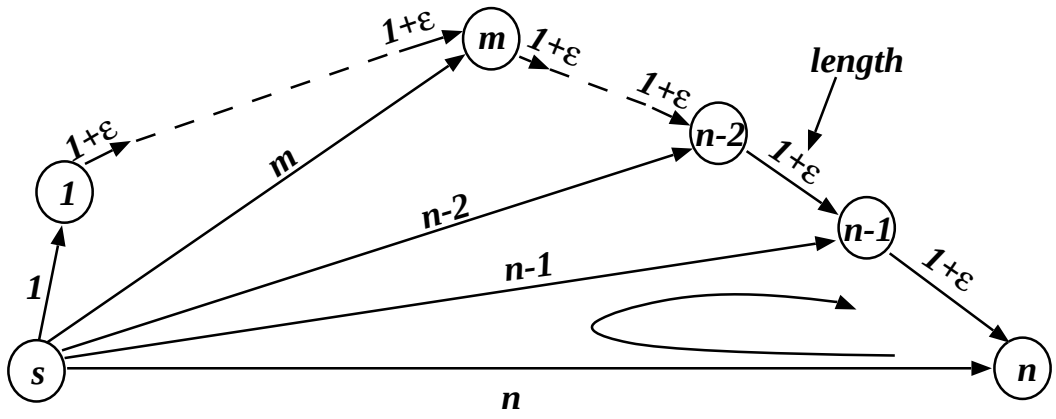


Figure 3.7: A simple network that will benefit from the cycle reduction algorithm.

sinks, resulting in a suboptimal solution with no bandwidth sharing. The cycle reduction algorithm identifies the negative cycles in the LDF solution, redirects the flow along these cycles, and eventually finds the single path $s \rightarrow 1 \cdots \rightarrow n-1 \rightarrow n$ as the solution, which is the optimal solution in this case. With a small ϵ, the improved solution is $O(n)$ times better than the original one.

Performance Improvement in Cycle Reduction Algorithm with Compressed Paths

The original cycle reduction algorithm by Gallo and Sodini has to check every tree vertex for negative cost cycles. If there are k vertices in the tree defined by the existing flow, it requires k shortest path tree computation. However, as Guisewite and Pardalos noted in [31], it is not necessary to check every tree vertices. Instead, we can check all the vertices on a non-branching path in the tree simultaneously. Figure 3.8 shows some non-branching paths in the tree defined by an existing flow. In this figure, x and u are two branching vertices in the tree, and v is a sink vertex. u and x are possible sink vertices too. To determine the best alternative path into a tree vertex, it is sufficient to construct a shortest path tree for every non-branching path in the tree defined by the existing flow. For the example in Figure 3.8, instead of computing a shortest path tree for every vertex on the $x \rightarrow u$ and $u \rightarrow y$ paths, we only need to compute two shortest path trees. Because we consider all vertices on a non-branching path simultaneously, we refer to this improvement heuristic as path compression. If there are m sink vertices, we only need to compute at

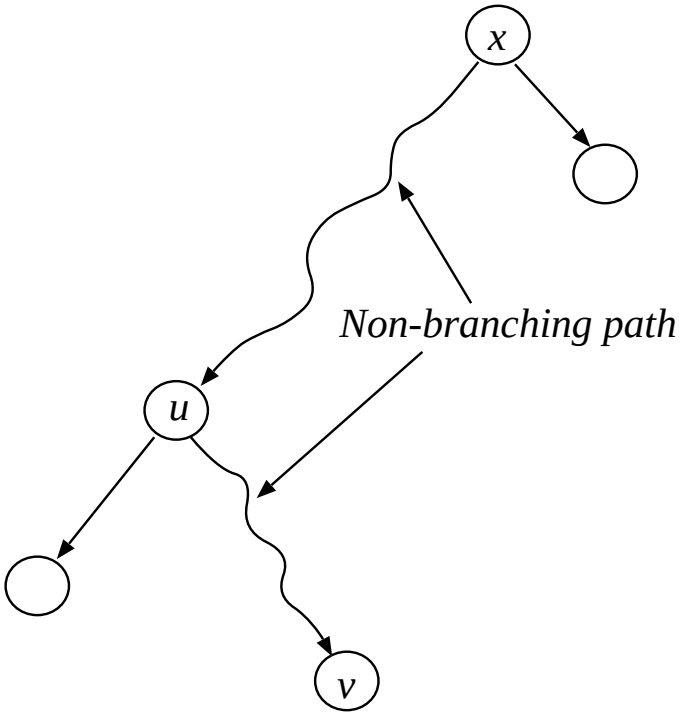


Figure 3.8: Path compression in the cycle reduction algorithm.

most $2m-1$ shortest path trees because there are at most $2m-1$ non-branching paths in a tree. In contrast, if there are n tree vertices, the original cycle reduction algorithm has to compute all n shortest path trees to find the local optimal. It is easy to see that $n > 2m-1$, and the performance improvement could be very substantial. The following pseudo code outlines the improved cycle reduction algorithm with path compression:

Repeat
  For each leaf or branching tree vertex t_i with an incoming flow of Δ,
    For each tree vertex $t_j \neq t_i$,
      Compute $c^*_{t_j}$ the incremental cost of redirecting Δ units of flow
        along the undirected tree path $\pi^*_{t_i,t_j}$.
    For each non-tree edge (u,v),
      Compute incremental cost c_{uv} of adding Δ units of flow on (u,v).
    Let $b(t_i)$ be the nearest branching ancestor.
    $G' = (V', E')$ is the subgraph induced
      by non-tree edges and edges on the non-branching path $(b(t_i), t_i)$.
      $V' \leftarrow V' \cup \{s'\}$
      $E' \leftarrow E' \cup \{(s', v) | v$ is a tree vertex, and $v \neq t_i\}$
        $-\{(w,v) | v$ is a tree vertex, and $v \notin (b(t_i), t_i)\}$,
    For each edge $(s', t_j) \in E'$, assign a length of $c^*_{t_j}$;
    For any other edge $(u,v) \in E'$, assign a length of c_{uv}.
    Find the shortest path from s' to t_i in G'.
    Let edge (s', w) be on the shortest path from s' to t_i in G'.
    Redirect Δ units of flow along $\pi^*_{t_i,w}$ and π_{w,t_i}, and obtain an updated flow x'.
    If the update flow x' has a higher cost than the current flow,
      restore the original flow.
until no flow with lower cost can be obtained.

Complexity Analysis Let n be the number of vertices in the network, and k be the number of sink vertices. The improved cycle reduction algorithm only needs to compute at most $2k-1$ shortest path tree for the non-branching paths in the tree defined by the existing flow to find the local minimal, as in contrast with n shortest path computations in the original cycle reduction algorithm. This improvement speeds up the search for each flow derived from the initial flow, and therefore the whole local search procedure. Because the

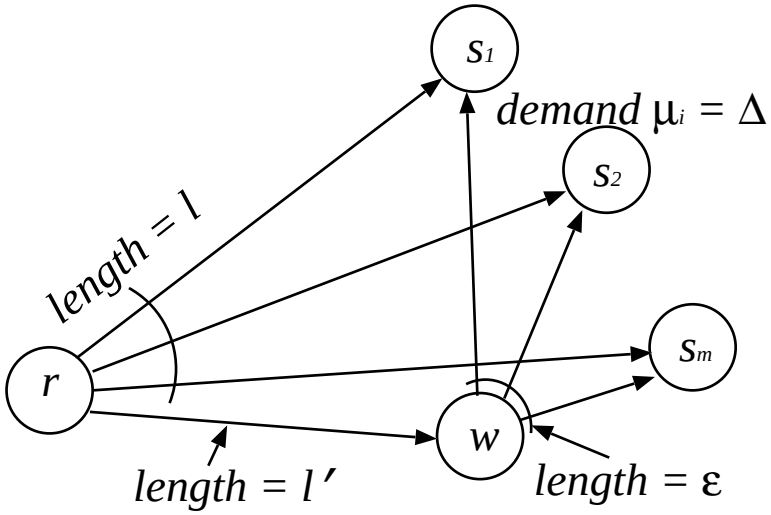
shortest path tree computation is the dominating factor of the time complexity of the cycle reduction algorithm, the path compression heuristic greatly improves the performance of the local search.

These local search algorithms essentially enumerate all neighboring extreme flows reachable from an existing extreme flow by redirecting flows along a negative cost cycles. In a network with linear edge costs, the resulting flow has no neighboring flow that has lower cost because negative cost cycles are sufficient to find local optimal in such a network. However, in a network with concave edge costs, such as an RDS, the result from the original cycle reduction algorithm does not necessarily include all possible neighboring extreme flows with lower costs as we demonstrate in the next section.

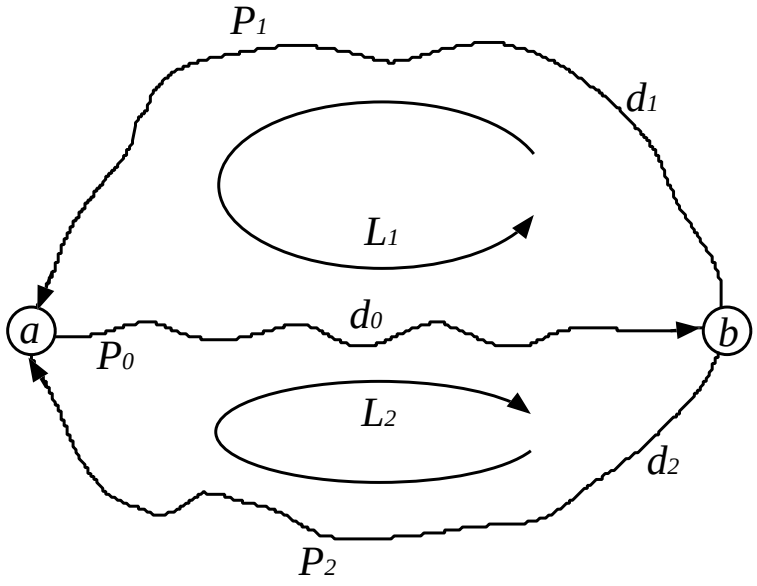
3.3.3 Negative Cost Bicycles in Concave Cost Networks

In this section, we show that the cycle reduction algorithm can be sub-optimal in a network with concave edge costs. Consider the example network shown in Figure 3.9(a). The source vertex r connects to m sink vertices with edges of length l. r also connects to an intermediate vertex w with an edge of length l' that is slightly shorter than l. w also connects to each sink vertex with an edge of length of ϵ. Each sink vertex is associated with a demand of Δ. For simplicity, we still adopt the simple concave edge cost function $cost(x,(u,v)) = l(u,v)x^{1/2}$ of a flow x on an edge (u,v) with a length of $l(u,v)$. With this cost function, the optimal cost is $\epsilon\Delta^{1/2}m + l'(\Delta m)^{1/2}$, while the result based on the shortest path tree (which is the local optima) has a cost of $l\Delta^{1/2}m$. So, if $\epsilon \leq l/m^{1/2}$, the cost ratio of the two solutions is no less than $m^{1/2}/2$, which can be arbitrarily large. This example suggests that reduction on negative cost cycles alone does not guarantee a results with sufficient quality, and better solutions can be reached by redirecting flow along subgraphs with special structures. For the example network in Figure 3.9(a), we notice that we can reach a neighboring flow with lower cost by adding 2Δ units of flow along (r,w), and Δ units of flow along (w,s_1), (s_1,r), (w,s_2), and (s_2,r). The paths we redirect flow on constitute a subnetwork with special structures that we will explore in this section.

In a network with concave link costs, there could exist negative bicycles such as the one in Figure 3.9(a) that could transform an existing flow to a flow with lower cost. We define a *negative cost bicycle* as a pair of directed cycles that share a common segment, with the



(a) An example network.



(b) A general negative cost bicycle.

Figure 3.9: A simple negative cost bicycle example.

remainder of the cycles edge disjoint. When we add flow along the two cycles, the total cost of the resulting flow is lower than the original flow. A general negative cost bicycle is illustrated in Fig. 3.9(b) that has a pair of vertices a and b. There is a common path P_0 from a to b, and two paths P_1 and P_2 from b to a. The sum of the cost of all these path is negative. Let d_0 be the length of the common segment of the negative cost bicycle, and d_1 and d_2 be the length of the disjoint segments. Then, the incremental cost of adding a unit flow along the bicycle could be expressed as $(1+\epsilon)d_0 + d_1 + d_2$, where $0 \leq \epsilon \leq 1$. If $\epsilon = 1$, the cost of the bicycle is equal to the sum of the cost of both cycles with the usual definition of flow costs. If $\epsilon = 0$, the bicycle is only charged once for the shared segment. Any other $0 < \epsilon < 1$ would result in a cost falls in between, showing the benefits of path sharing. It is easy to see that negative cost bicycle is just one subnetwork structure that can lead to lower cost neighboring flows. However, we first focus on finding negative cost bicycles only, because they are more likely to appear in an existing flow, and therefore have greater effects on final costs. We will generalize to deal with more complex subnetwork structures than bicycles in a later section. However, the cost benefits could be offset by the computational complexity of exploring more complicated structures.

For a general negative cost bicycle in a network with the simple concave edge cost function as defined in the example network, when we push flow Δ along the negative cost bicycle, we add 2Δ flow on the common segment of the bicycle P_0, and Δ on the disjoint paths P_1 and P_2. Thus, the incremental cost C_Δ for a flow increment Δ can be expressed as

$$\begin{aligned} C_\Delta &= \sum_{i \in P_0} l_i(\sqrt{\mu_i + 2\Delta} - \sqrt{\mu_i}) \\ &- \sum_{i \in P_1^-} l_i(\sqrt{\mu_i} - \sqrt{\mu_i - d}) + \sum_{i \in P_1^+} l_i(\sqrt{\mu_i + d} - \sqrt{\mu_i}) \\ &- \sum_{i \in P_2^-} l_i(\sqrt{\mu_i} - \sqrt{\mu_i - 2\Delta + d}) + \sum_{i \in P_2^+} l_i(\sqrt{\mu_i + 2\Delta - d} - \sqrt{\mu_i}) \end{aligned}$$

where d is the flow added on the path P_1, P_1^+ is the set of links in path P_1 that have flows in the same direction as P_1, and P_1^- is the set of links in path P_1 that have flows in the opposite direction of P_1; P_2^+ and P_2^- are the sets similarly defined on P_2. Because this is a negative cost bicycle, $C_\Delta < 0$ for some Δ. We must identify these negative cost bicycles with specific flow increment Δ to reduce the cost of an existing flow.

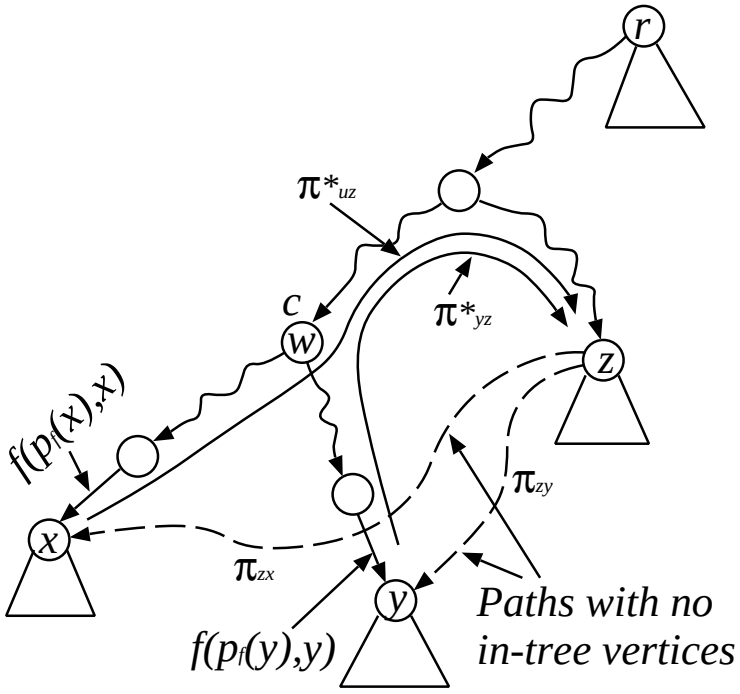
3.3.4 Bicycle Reduction Algorithm

As we described previously, an adjacent extreme flow with lower cost can be reached by redirecting flow along a negative cost bicycle. Thus, in order to extend the cycle reduction algorithm, we must efficiently identify these negative cost bicycles after the negative cost single cycles are all removed by the original cycle reduction algorithm.

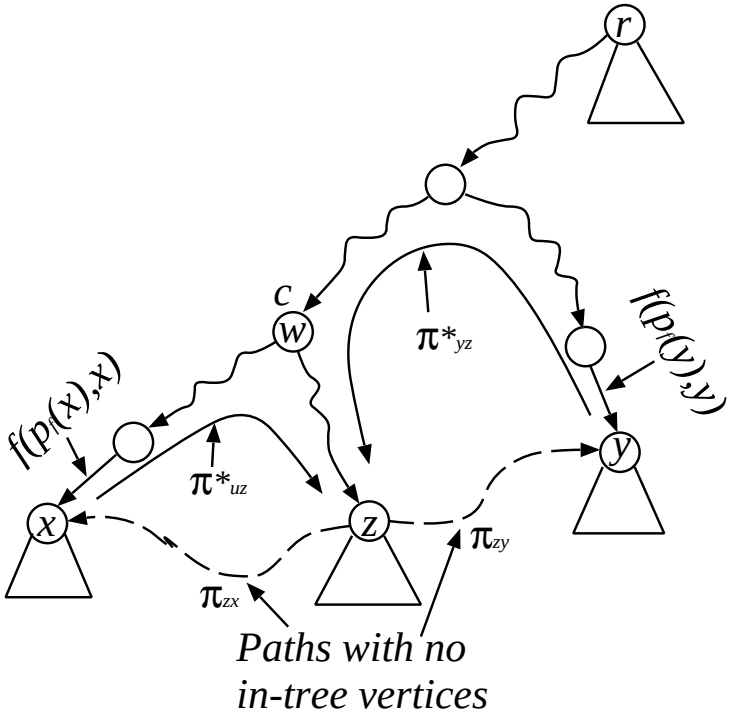
In the original cycle reduction algorithm, for a vertex u in the existing flow, we find another vertex v in the existing flow, where the undirectional path in the existing flow π^*_{uv} and the path π_{vu} not used by the existing flow form a minimum cost cycle. If it is a negative cycle, a neighboring extreme flow with lower cost can be reached by redirecting flow along this cycle.

In contrast to a negative cost single cycle, a negative cost bicycle consists of two cycles with a common path segment. Therefore, in order to find a negative cost bicycle, we start with two vertices, x and y, neither of which is a non-branching vertex in the tree defined by the existing flow. We then search for a third tree vertex z, through which there is a pair of directed non-tree paths π_{zx} and π_{zy}. We define z as the *optimal split point*. In addition, there is another vertex w on both undirectional tree paths π^*_{xz} from x to z and π^*_{yz} from y to z. We define w as the *optimal merge point*. The tree path π^*_{wz} is the common segment of the bicycle, and the two cycles are $(\pi^*_{xw}, \pi^*_{wz}, \pi_{zx})$ and $(\pi^*_{yw}, \pi^*_{wz}, \pi_{zy})$. Figure 3.10(a) and Figure 3.10(b) show two example negative cost bicycles. Let f_x and f_y be the amount of flow into x and y respectively in the existing flow, after redirecting f_x units of flow along $(\pi^*_{xw}, \pi^*_{wz}, \pi_{zx})$ and f_y units of flow along $(\pi^*_{yw}, \pi^*_{wz}, \pi_{zy})$, the adjacent extreme flows are shown in Figure 3.11(c) and Figure 3.11(d).

Given such a pair of vertices x and y in a tree defined by an existing flow, we first observe that the optimal split point z can not be on the undirectional tree path π^*_{xy} between x and y. Because if z is on π^*_{xy}, the optimal merge point w must be the same vertex as z. This means that there is no common path segment in the two cycles (π^*_{xz}, π_{zx}) and (π^*_{yz}, π_{zy}), and thus not a bicycle. In addition, it is clear that the optimal split point can not be in the subtrees rooted at neither x nor y, because it will not lead to a bicycle either. Based on these observations, we limit our search for the optimal split point in the vertices of the existing flow that are neither in the subtree rooted at x or y nor on the undirectional path π^*_{xy}.

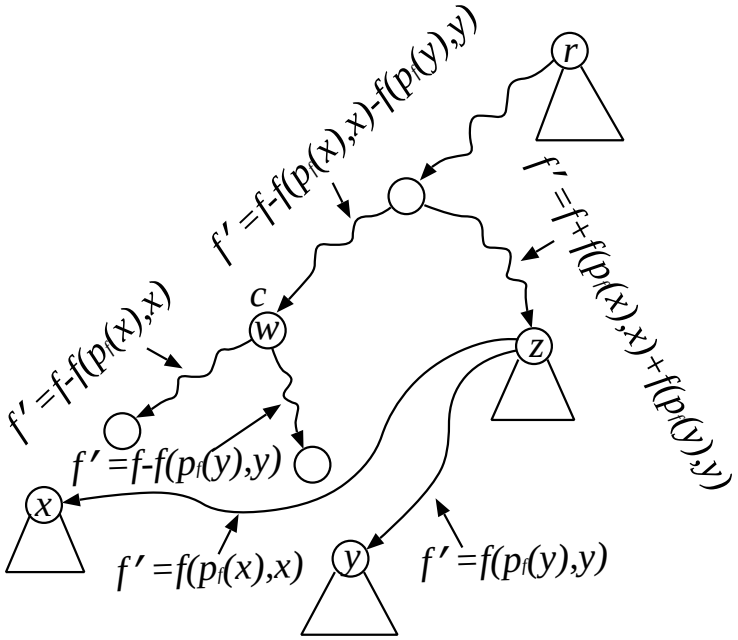


(a) Original extreme flow.

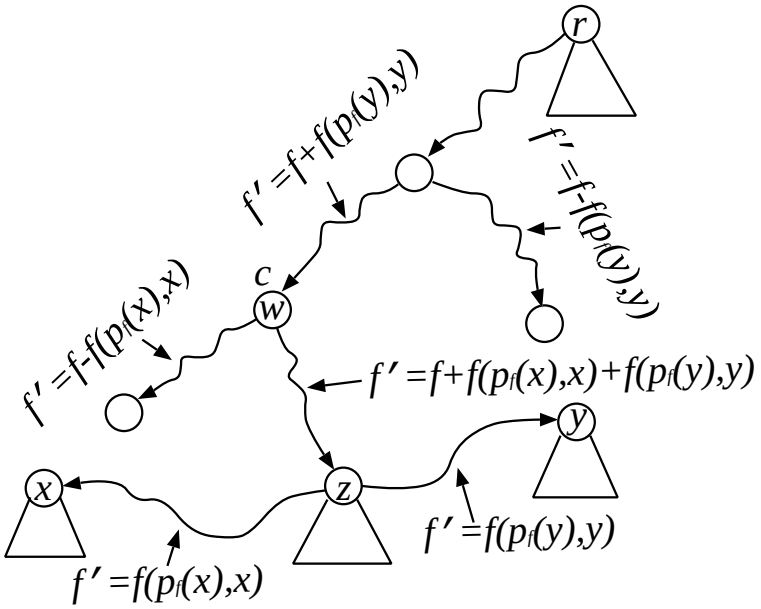


(b) Original extreme flow.

Figure 3.10: Bicycle reduction algorithm.



(c) The neighboring extreme flow obtained.



(d) The neighboring extreme flow obtained.

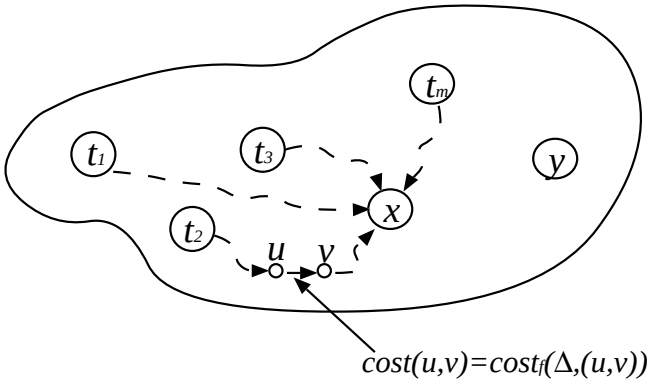
Figure 3.11: Bicycle reduction algorithm.

If we define the nearest common ancestor of x and y as c as in Figure 3.10 and Figure 3.11, the bicycle reduction algorithm can be described as follows:

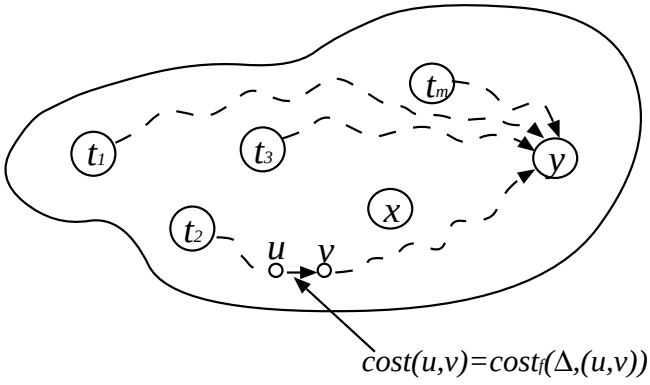
First, compute the incremental cost of redirecting f_x units of flow from x and f_y units of flow from y to all potential split points in the existing flow as follows: for each tree vertex u that is either in the subtree rooted at x or y or on the undirectional path π^*_{xy}, define the incremental cost c^*_u to be ∞. For any other tree vertex u, if u is in the subtree rooted at c, let v be the nearest ancestor of u on π^*_{xy}. Let c^*_{xv} be the incremental cost of redirecting f_x units of flow along the undirectional path π^*_{xv} from x to v, c^*_{yv} be the incremental cost of redirecting f_y units of flow along the undirectional path π^*_{yv} from y to v, and c^*_{vu} be the incremental cost of redirecting $f_x + f_y$ units of flow along the undirectional path π^*_{vu} from v to u. The total incremental cost c^*_u for u is defined as $c^*_u = c^*_{xv} + c^*_{yv} + c^*_{vu}$. If u is not in the subtree rooted at c, let c^*_{xc} be the incremental cost of redirecting f_x units of flow along the undirectional path π^*_{xc} from x to c, c^*_{yc} be the incremental cost of redirecting f_y units of flow along the undirectional path π^*_{yc} from y to c, and c^*_{cu} be the incremental cost of redirecting $f_x + f_y$ units of flow along the undirectional path π^*_{cu} from c to u. The total incremental cost c^*_u for u is defined as $c^*_u = c^*_{xc} + c^*_{yc} + c^*_{cu}$.

Next, make two copies $G_x = (V, E_x)$ and $G_y = (V, E_y)$ of the network with existing flow $G = (V, E)$. In G_x, remove the following edges: all tree edges, all non-tree edges that incident into any tree vertex other than x, and all non-tree edges originated from any tree vertex in the subtrees rooted at x or y, or from any tree vertex on the undirectional tree path π^*_{xy}. Compute the incremental cost c_{uv} for an edge (u, v) in G_x as adding f_x amount of flow on (u, v). Similarly remove edges from G_y, and compute the incremental cost c_{uv} for an edge (u, v) in G_y as adding f_y amount of flow on (u, v). Then, compute the shortest paths from all vertices in G_x to x, and shortest paths from all vertices in G_y to y. This can be achieved by running a single destination shortest path algorithm (or a simple modified single source shortest path algorithm) with x or y as the destination vertex. For each vertex u in G_x and G_y, record the shortest paths π_{ux} in G_x and π_{uy} in G_y as well as the shortest distance c_{ux} and c_{uy}. Figure 3.12 shows the transformed graphs from the example networks in Figure 3.10. In particular, Figure 3.12(a) shows G_x and Figure 3.12 (b) shows G_y created from the existing flow.

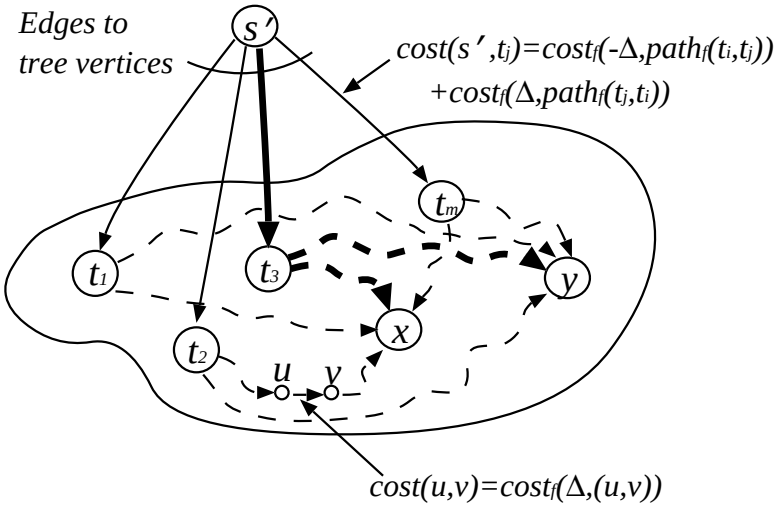
At last, for any tree vertex u other than x and y, define the final total cost $C_u = c_{ux} + c_{uy} + c^*_u$. Find the vertex with the minimum final total cost. This step is illustrated in Figure 3.12(c).



(a) G_x.



(b) G_y.



(c) Combination of G_x and G_y.

Figure 3.12: Negative bicycle reduction algorithm.

In particular, it is equivalent to first combined G_x and G_y, then adding a pseudo source vertex s', and connect s' to any vertex u in the combined graph where c_{ux} and c_{uy} are less than ∞. An additional edges from s' to a vertex u has the length equal to C_u. Thus, the shortest edge (s', z) corresponds to the optimal split point z, and the paths π_{zx} and π_{zy} are already recorded in G_x and G_y. (s', z) and π_{zx} and π_{zy} are highlighted in Figure 3.12(c). After the optimal split point z is identified, redirect f_x amount of flow along π_{zx} and π^*_{xz}, and f_y amount of flow along π_{zy} and π^*_{yz}. Compute the total cost of the updated flow. If the updated flow has a lower cost than the original flow, record the updated flow and repeat the above procedure. Otherwise, restore the original flow and stop.

The following pseudo code describes the bicycle reduction operations:

Repeat
  For any two leaf or non-branching tree vertices x and y
    with incoming flows of f_x and f_y, respectively.
    For another tree vertex u,
      If u is on the undirected tree path π^*_{xy}, or in the subtree rooted at x or y,
        $c^*_u \leftarrow \infty$.
      else
        Find the nearest common ancestor c of x and y.
        If u is in the subtree of c,
          Let v be the nearest ancestor of u on π^*_{xy}.
          $c^*_{xv} \leftarrow$ incremental cost of redirecting f_x unit of flow on π^*_{xv}.
          $c^*_{yv} \leftarrow$ incremental cost of redirecting f_y unit of flow on π^*_{yv}.
          $c^*_{vu} \leftarrow$ incremental cost of redirecting $f_x + f_y$ unit of flow on π^*_{vu}.
          $c^*_u \leftarrow c^*_{xv} + c^*_{yv} + c^*_{vu}$.
        else
          $c^*_{xc} \leftarrow$ incremental cost of redirecting f_x unit of flow on π^*_{xc}.
          $c^*_{yc} \leftarrow$ incremental cost of redirecting f_y unit of flow on π^*_{yc}.
          $c^*_{cu} \leftarrow$ incremental cost of redirecting $f_x + f_y$ unit of flow on π^*_{cu}.
          $c^*_u \leftarrow c^*_{xc} + c^*_{yc} + c^*_{cu}$.
    Create $G_x = (V, E_x)$ and $G_y = (V, E_y)$ from G.
    Find the shortest paths from all vertices to x in G_x and to y in G_y.
    For a tree vertex u,

c_{ux} is the shortest distance from u to x in G_x,
c_{uy} is the shortest distance from u to y in G_y.
$C_u \leftarrow c_{ux} + c_{uy} + c^*_u$.
Find the optimal split point z that gives the minimum C_z.
Redirect f_x units of flow along paths π^*_{xz} and π_{zx}, and f_y units of flow along paths π^*_{yz} and π_{zy},
If the modified flow x' has a higher cost than the original flow,
Restore the original flow.
until no flow with lower cost can be obtained.

Complexity Analysis Let n and m be the numbers of vertices and edges in the network, and k be the number of sink vertices. For each flow, it may need to check $O(k^2)$ vertices pairs before it can determine if a neighboring extreme flow with lower cost exist through flow redirection along a negative cost bicycle. It requires solving k nearest common ancestor problems and computing $4k^2$ incremental path costs. However, the checking procedure is dominated by the shortest path tree computation. If we denote $S(n, m)$ as the time complexity of a single source shortest path algorithm in a graph with n vertices and m edges, then the time complexity of find a neighboring flow with lower cost in the bicycle reduction algorithm is $O(n^2 S(n, m))$, or $O(n^3(m + n^2 \log n))$ if the shortest path algorithm is implemented efficiently.

3.3.5 Experimental Results and Analysis

In this section, we present the simulation studies of the bicycle reduction algorithm in networks with a simple concave edge cost function. We start with the description of the problem instances we generate in our simulations on different network topologies. Next, we explain the two estimated lower bounds we use for performance comparison. The results from our simulation studies are then presented with analysis.

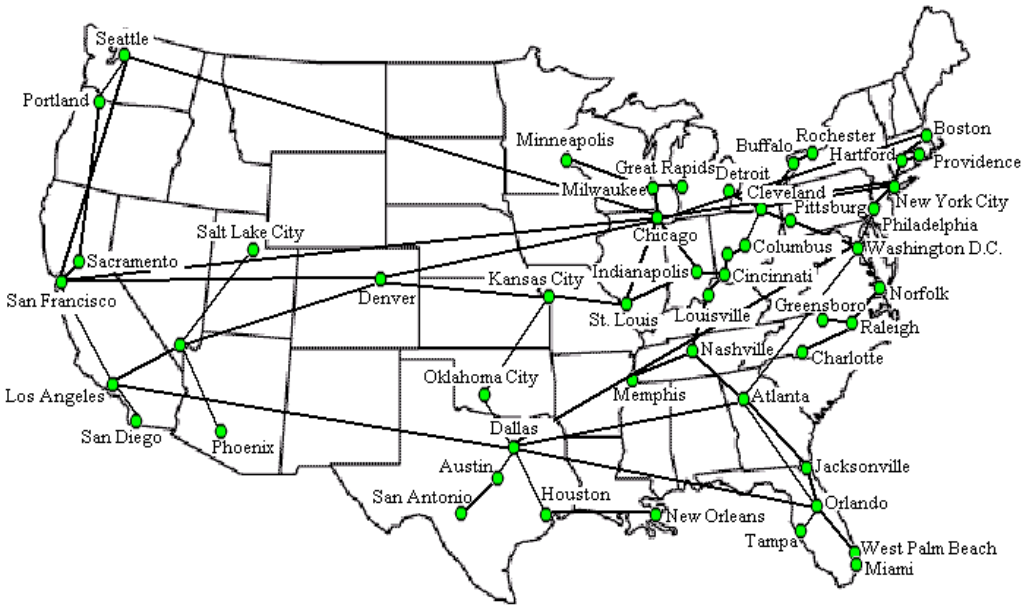


Figure 3.13: National network configuration.

Simulation Setup

We evaluated the bicycle reduction algorithm on a number of network topologies. The first is a 15×15 torus (each node is connected to four neighbors forming a rectangular grid with "wrap-around edges" linking the top and bottom rows and the leftmost and rightmost columns). We used two types of networks with different link lengths: links in a *random torus* were uniformly distributed, with the longest links being ten times longer than the shortest, while links in a *uniform torus* have a fixed length. Links in random torus are restricted such that triangular inequality is observed. The demands for the sinks were uniformly distributed, all with the same mean demand.

The second network, shown in Figure 3.13, includes a node at each of the fifty largest metropolitan areas in the United States; the link lengths were chosen to be equal to the geographic distances between the locations, and the demands were chosen to be proportional to the populations of the metropolitan ares [79]. The locations of sources and sinks were selected randomly, with every node having the same probability of selection.

The vertices and edges of the third network are uniform randomly generated inside a unit square. The source node is located in the corner of the unit square as this choice creates extreme flows with deeper trees and greater effect of flow aggregation. The sinks are uniform randomly chosen in the unit square with uniform random sink demands drawn from the range of $[1, 10]$. Such a network would result in extreme flow solutions represented in deep trees, and thus would increase the running time of the local search algorithms. However, it should be noted that such a network has a higher degree of "incidental sharing", and is closer to the optimal solution already.

Besides these three network topologies, we also simulate the bicycle reduction algorithm in networks generated by a topology generator, Inet [82]. Recent studies showed that degree-based topology generators creates networks that have high resemblance of the Internet even though these generators do not consider the network structure specifically [75]. The original Inet is intended for large network with at least 3037 vertices. In our simulation studies, we used a modified version of the Inet generator so that smaller networks could be generated. The networks we generated for our simulations have 100 vertices, and the fraction of degree one vertices is 0.3. Because it uses a seed for the random number generator, the number of networks for a fixed number of vertices is at most 64. Thus, each data point is the average of results of 64 independent problem instances, instead of 100 instances as in simulations in other topologies.

We measure the relative costs of flows generated by different algorithms with the estimated bound. We first obtain an initial solution with the *largest demand first (LDF)* algorithm we developed in our previous study of the RDS configuration problem in [62], and apply the original cycle reduction algorithm as well as the bicycle reduction algorithm to find two local optimal solutions from the flow produced by LDF. The number of sinks is varied to show the performance of different algorithms in a variety of network conditions. We also measure the percentage improvements to the flows obtained by LDF after applying the cycle reduction algorithm and the bicycle reduction algorithm.

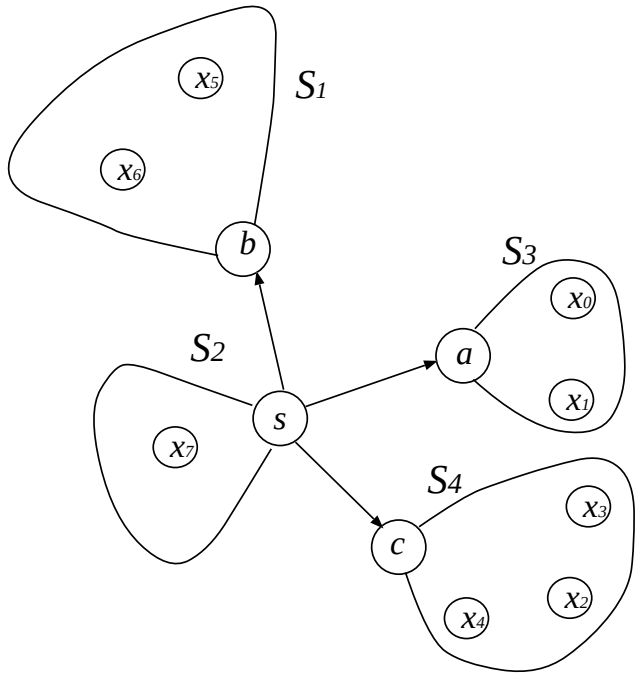
Lower Bounds Comparison

In our previous study of the RDS configuration problem, we used an easily computed estimated lower bound for the performance evaluation [62]. The idea is to assume all the

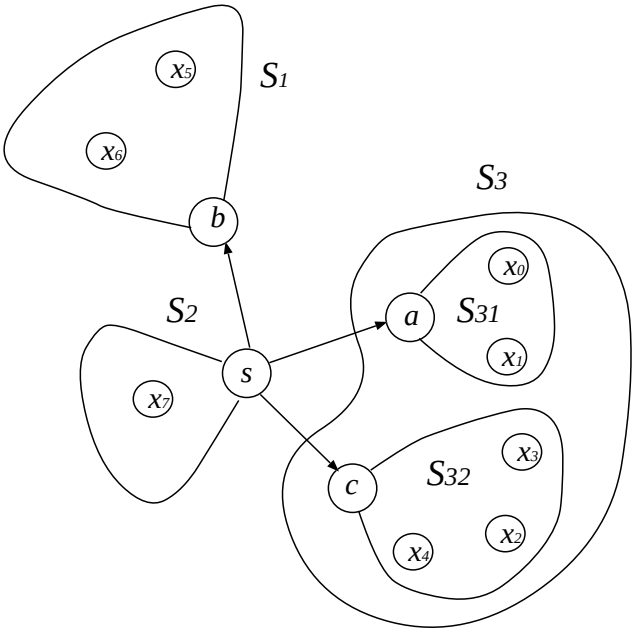
sinks that are located in a certain geographical area share a single path to the source vertex, and thus achieve maximum possible path sharing among all the sink vertices in that area. In particular, estimated bound EB*(2) is computed by first dividing the sinks into two sets, those to the “left” of the source and those to the ”right” of the source. Each of these subsets is then sorted by distance from the source and each node is assumed to share its path to the source with all nodes in the same subset that are at greater distance from the source. EB*(3) (and EB*(4)) is computed similarly, by first dividing the sinks into three (respectively four) sets of nodes defined by “pie-shaped” regions centered on the source, then sorting the subsets by distance from the source and assuming the maximum possible sharing of paths among nodes in the same set. For larger numbers of randomly distributed sinks, it’s reasonable to expect EB*(2), EB*(3) and EB*(4) to be no larger than the cost of an optimal solution, although they do not constitute true lower bounds.

A tighter lower bound for a network with a small number of sink nodes can be computed as follows: define a *partial solution* as a subtree rooted at the source along with a partition of sinks among tree nodes. For instance, Figure 3.14(a) shows a partial solution in which the sink vertices are divided among sets S_1, S_2, S_3 and S_4. We can get a lower bound with partial solutions for a network $G = (V, E)$ with s as the source vertex in the following way: let T be a subtree of G rooted at s with three edges. Partition the sink vertices such that there is a subset of sink vertices S_i for each tree node t_i. Compute the total cost of T assuming each tree node has the demand equal to the total demands of the sinks in the subset associated with that tree node. For each tree node t_i, compute the lower bound cost for supplying all sink vertices in S_i from t_i, assuming all sink vertices in S_i share a single path to t_i and the distance from t_i to a sink vertex x is the same as the shortest path from t_i to x. Add all these costs associated with tree nodes to the cost of T to obtain an estimated total cost value. The pseudo code for this is shown below:

Create a subtree T rooted at s with three edges.
For each 4-partition (S_1, S_2, S_3, S_4) of sink vertices,
    Associate each tree node t_i with a subset of sink vertices S_i.
    Assign each tree node with a demand equal to the sum of all sink vertices
        in the associated subset.
    Compute the costs of all tree edges.
    For each sink vertices subset S_i,



(a) An example partial solution.



(b) Optimization with connected components.

Figure 3.14: Lower bound computation.

Compute lower bound cost for supplying from t_i,
assuming shortest distance and maximum sharing.
Add all contributions.

If we iterate over all such partial solutions with three edges, we get overall lower bound. The trouble with is this is that the best lower bound is likely to involve large set at root; however, if we iterate over all subsets of edges from the root, we don't need to leave any behind at the root.

We can speed up the computation by avoiding some sink vertices partitions that obviously can not produce good lower bounds. This can be done by considering assignment of subsets of sink vertices only to the tree nodes in the same connected component. In this case, we consider the subtree T rooted at the source vertex with all the edges out of the source vertex. For each leaf node u in T, if removing (s, u) creates a connected component C_u with some sink vertices, then we only assign the subset of sink vertices in C_u to u. If a connected component C can only be created after removing multiple edges in T, then we apply the original lower bound operation on the subgraph that includes C and all edges that connect s and C. Figure 3.14(b) shows an example of this optimization. In this example, we only assign subset of sink vertices $\{x_5, x_6\}$ to tree node b, and $\{x_7\}$ to s. In the subgraph that includes a connected component that is connected to s through (s, a) and (s, c), we compute the lower bound by iterating the partitions of the sink vertices of $\{x_0, \cdots, x_4\}$, but this subgraph can be smaller than the original network, improving the computation time.

Similarly, if we can iterate over all depth two subtrees or "radius d" subtrees where tree nodes are within radius d (for each sink must consider smallest radius for its incoming neighbors), we would get tighter lower bounds. We can also apply the similar optimization operations with connected components to improve the performance too.

Figure 3.15 shows the comparison of the two lower bounds on a randomly generated network with 32 vertices and a variant number of sink vertices. In this plot, the relative cost of a lower bound to the estimated lower bound when we assume all sink vertices share a single path ($EB(1)$). Each of the data point is the average of 100 problem instances. The plot indicates that the tighter lower bounds fall mostly between $EB(2)$ and $EB(3)$, and

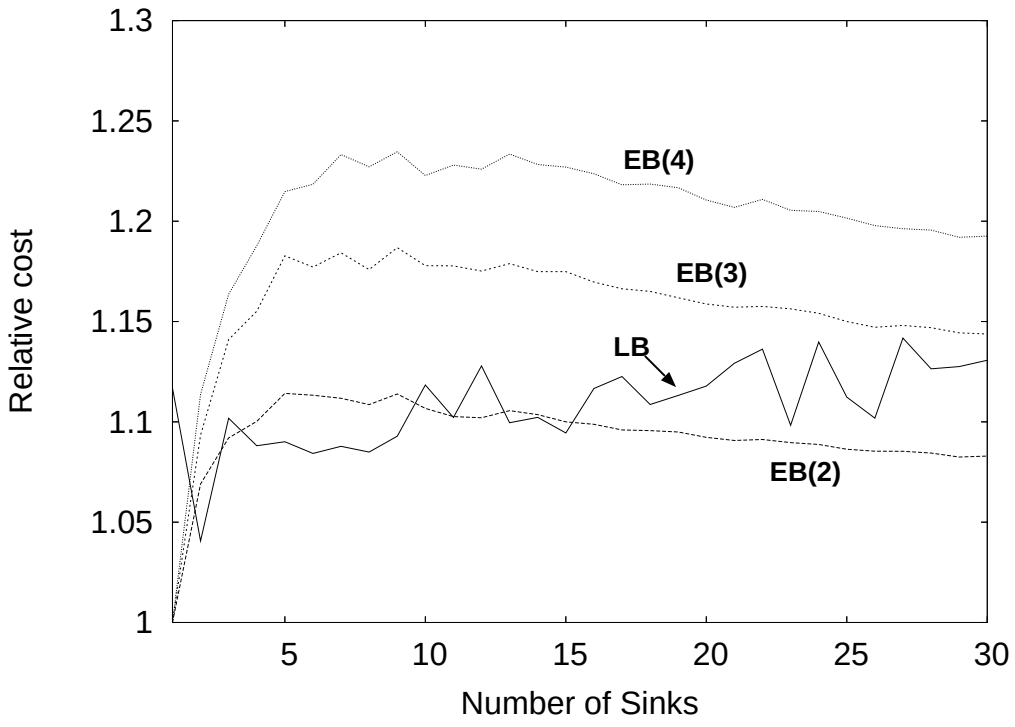


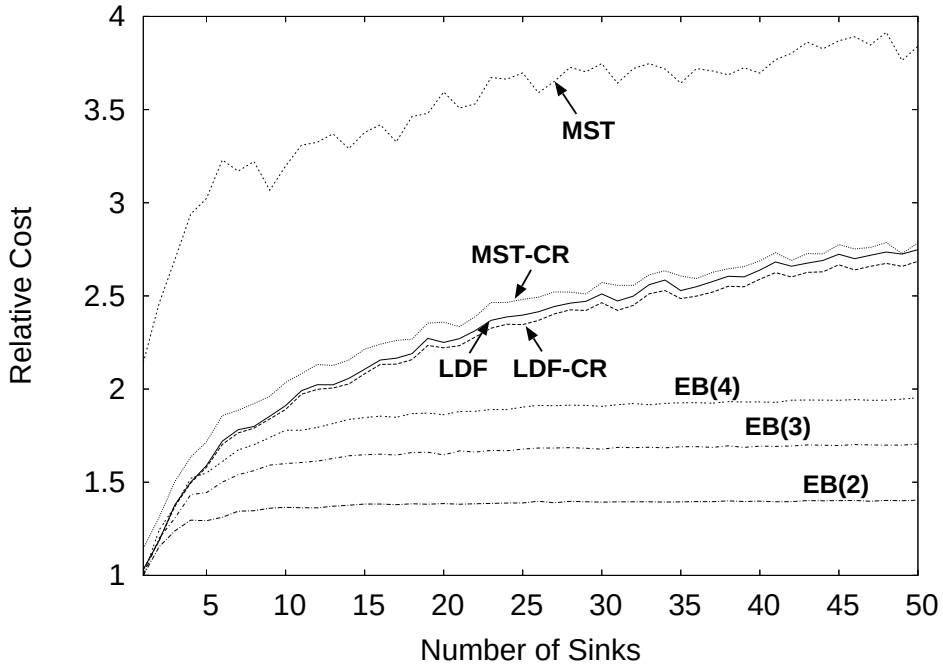
Figure 3.15: Comparison of estimated bounds and lower bounds.

gradually approach $EB(3)$ as the number of sink vertices increases. This results suggests that $EB(3)$ and $EB(4)$ can serve as sufficient lower bounds for networks with larger sizes.

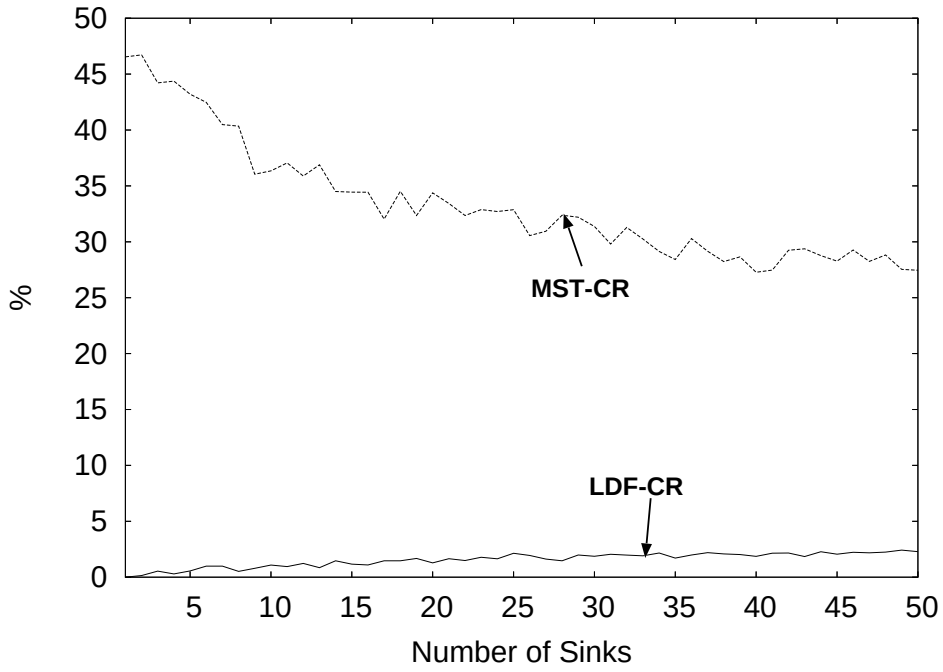
The performance of the above lower bound algorithms are largely determined by the partition enumeration method employed. Although these algorithms avoid a total enumeration of possible partitions, they are exponential in the worst case. Therefore, we compare the estimated lower bounds and the tighter lower bounds described above only for networks with small number of vertices (less than 30) to show how approximate the estimated lower bounds are. We use the Bit-Vector Representation (BVR) as used in [24] to efficiently enumerate through the partitions for the connected components. In particular, a subset of vertices are represented as a number bits in a word, in which the ith bit is set to 1 if vertex i is in the subset and 0 otherwise. Thus, a k-partition is represented as k integers that sum to 2^d- where d is the number of vertices in the connected component. For more general networks with larger numbers of vertices, we only compare the total cost relative to the estimated lower bound with the assumption that the tighter lower bounds maintain the similar ratio to the estimated lower bounds.

Simulation Results and Analysis

Figure 3.16 shows how the initial solution may make a difference. We apply the original cycle reduction algorithm to initial solutions obtained with the largest demand first (LDF) described in [62] and a minimum spanning tree (MST) algorithm in random torus networks, and compare the results. Note that although we use MSTs in our experiments, other simple initial solutions such as random spanning trees have similar results. Figure 3.16(a) shows the ratio of the cost of the solution produced by different algorithms to the estimated lower bound, as the number of sink vertices increases from 1 to 50. Each data point represents the average of results from 100 independent problem instances. As shown in the charts, for large numbers of sink vertices, the improved solutions obtained from the cycle reduction algorithm are similar: improved solutions from MST solutions are on average 2.75 times of the estimated lower bound, and the improved solutions from the initial LDF solutions are around 2.7 times the estimated lower bound. However, they are both closer to the initial LDF solutions obtained as they are no more than 2.8 times of the estimated lower bound, while the MST initial solutions are up to 3.85 times of the estimated lower



(a) Cost comparison with varied numbers of sinks.

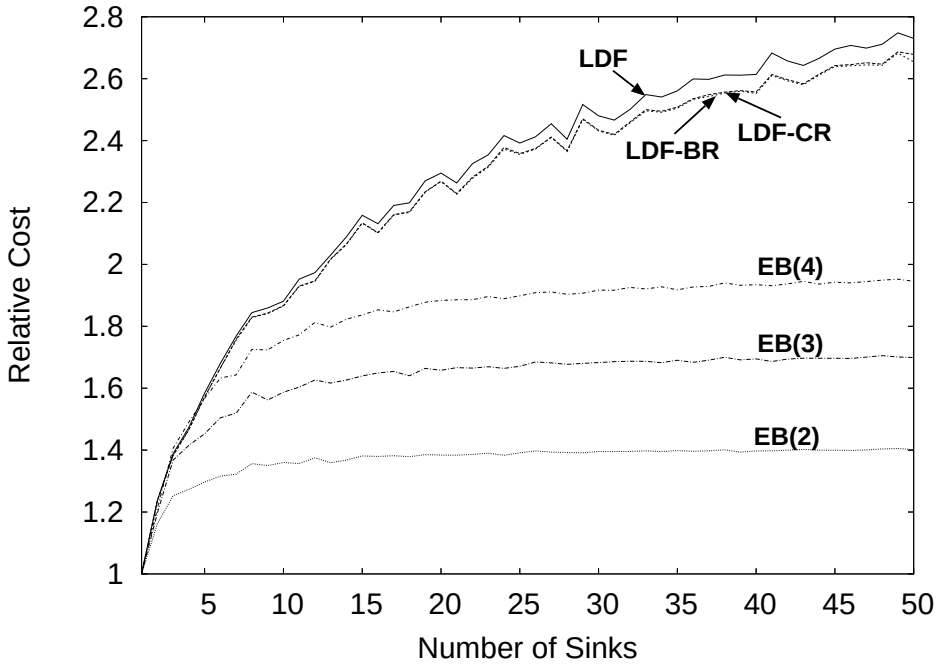


(b) Percentage cost improvement.

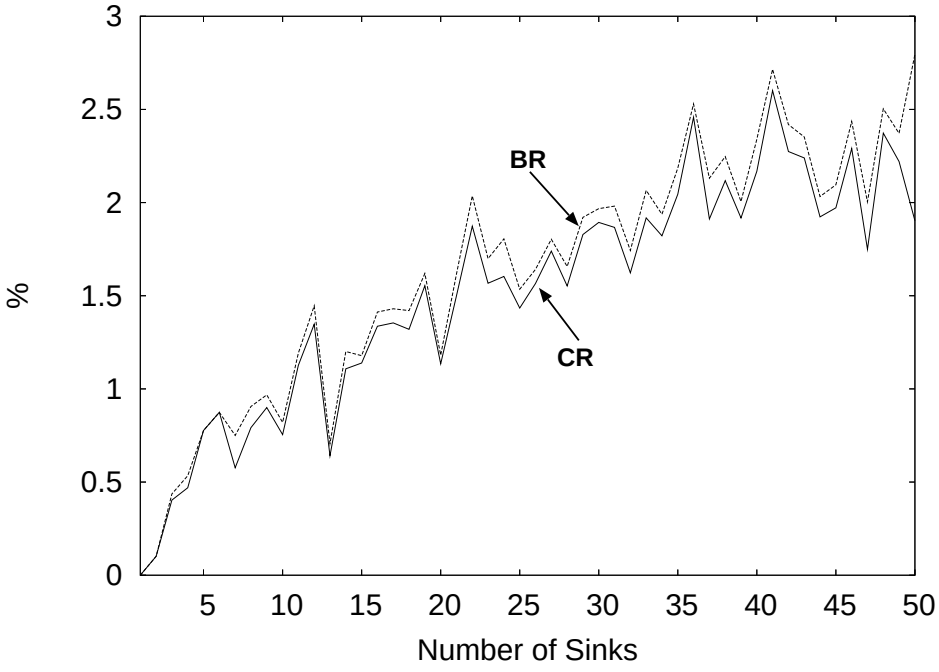
Figure 3.16: Cost comparison of cycle reduction algorithm with initial LDF solutions and MST solutions in torus networks.

bound. Figure 3.16(b) shows the percentage improvements of the cycle reduction algorithm from the MST and LDF solutions with varied numbers of sinks. It shows that the space for improvements from a MST solution is much larger than a LDF solution, as the cycle reduction algorithm improves the MST solutions by no less than 26% and even up to 46% in Figure 3.16(b), while the improvements from LDF solutions are all less than 3%. Because it is closer to the local optimal solutions, the initial solutions obtained by LDF has less negative cost cycles than those obtained by MST. Thus, it takes shorter time to reach improved solutions when we use an initial solution obtained from LDF. When we start with an arbitrary tree solution, the results are similar to the MST case. This result indicates that LDF algorithm provides solutions that are reasonablely close to optimal.

Figure 3.17 shows the simulation results on random torus networks. The curve labeled with LDF is the relative cost of the initial LDF solution, and the curves labeled LDF-CR and LDF-BR are the results obtained by applying the bicycle reduction algorithm and the original cycle reduction algorithm to the initial solution, respectively. Figure 3.17(a) shows the ratio of the cost of the solutions produced by different algorithms to the estimated lower bound, as the number of sink vertices increases from 1 to 50. For large numbers of sink vertices, the LDF algorithm produces solutions costing no more than about 2.8 times the estimated lower bound. The cycle reduction algorithm (LDF-CR) improves the solutions from LDF to no more than 2.7 times the estimated lower bound. The bicycle reduction algorithm makes some further improvements to the cycle reduction algorithm, but it is relatively small. This is more clear in the percentage improvements results of both the bicycle reduction and the original cycle reduction algorithms as the number of sink vertices increase in Figure 3.17(b). It shows that the original cycle reduction algorithm improvement of the LDF solutions grows as the number of sink vertices increase. When there are a large number of sink vertices, the improvement is about 2.5%. The bicycle reduction algorithm improves the cycle reduction results by up to 0.5% in some cases, but the average improvement is about 0.1%. A similar set of results obtained in uniform torus networks are shown in Figure 3.18. In these charts, when there are many sink vertices, the cycle reduction and bicycle reduction algorithms improved the average total cost from about 2.5 times of the estimated lower bound to about 2.4 times (Figure 3.18(a)), or about 2.25% average improvement (Figure 3.18(b)). The improvement grows as the number of sinks increases from 0 to 2.5%. However, the improvement contributed from the bicycle reduction algorithm is noticeablely smaller than in the random torus networks, ranging from 0 to 0.05%

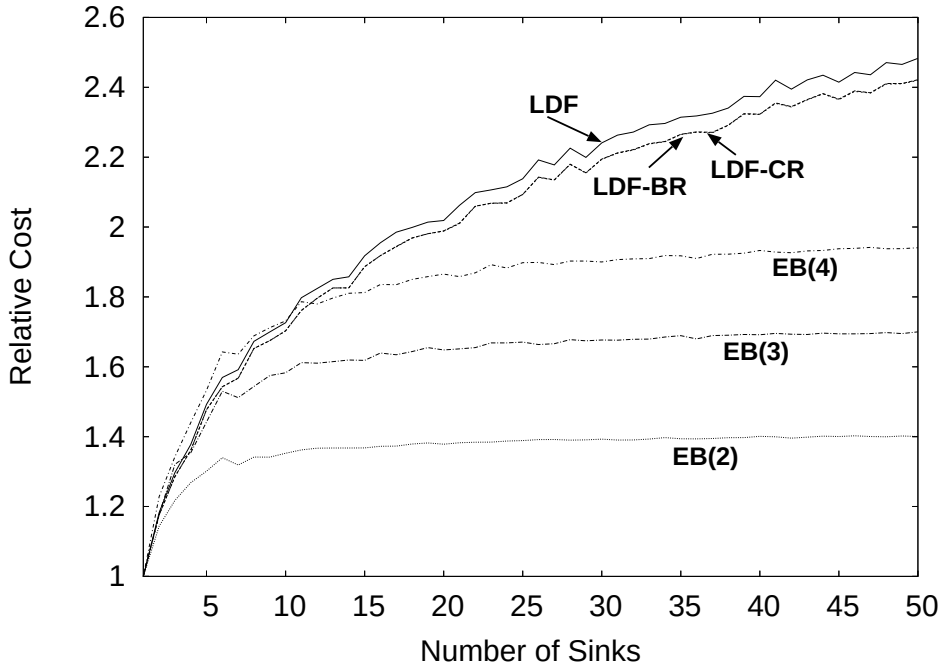


(a) Relative cost comparison.

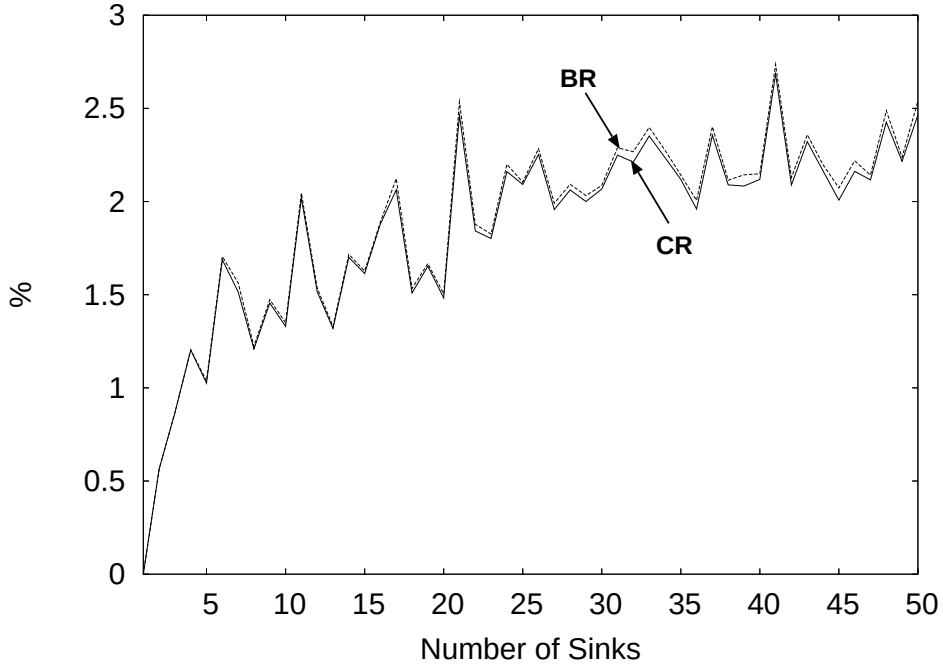


(b) Percentage cost improvement.

Figure 3.17: Cost comparison on torus networks.



(a) Relative cost comparison.



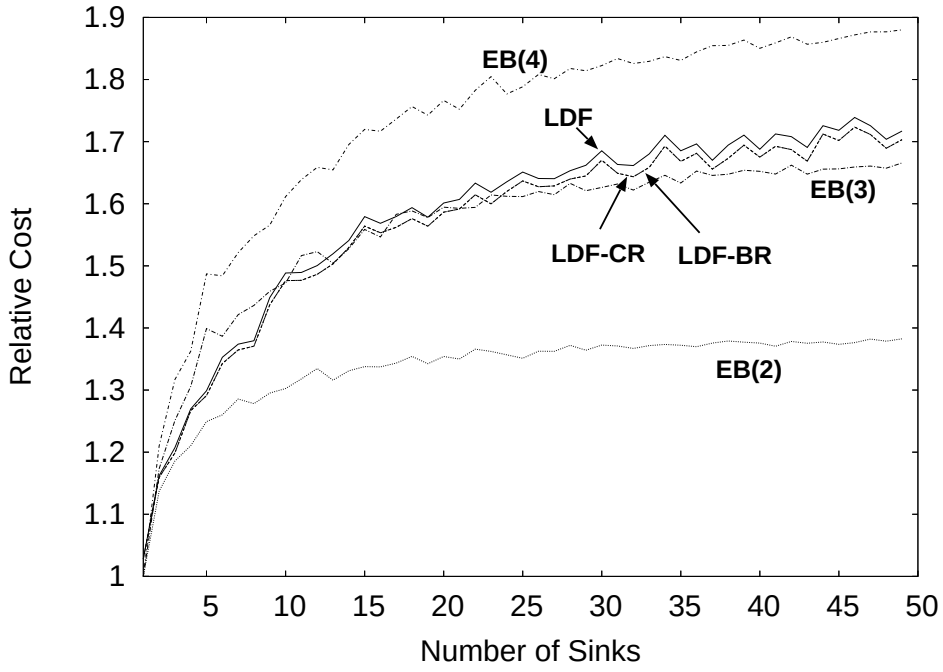
(b) Percentage cost improvement.

Figure 3.18: Cost comparison on uniform torus networks.

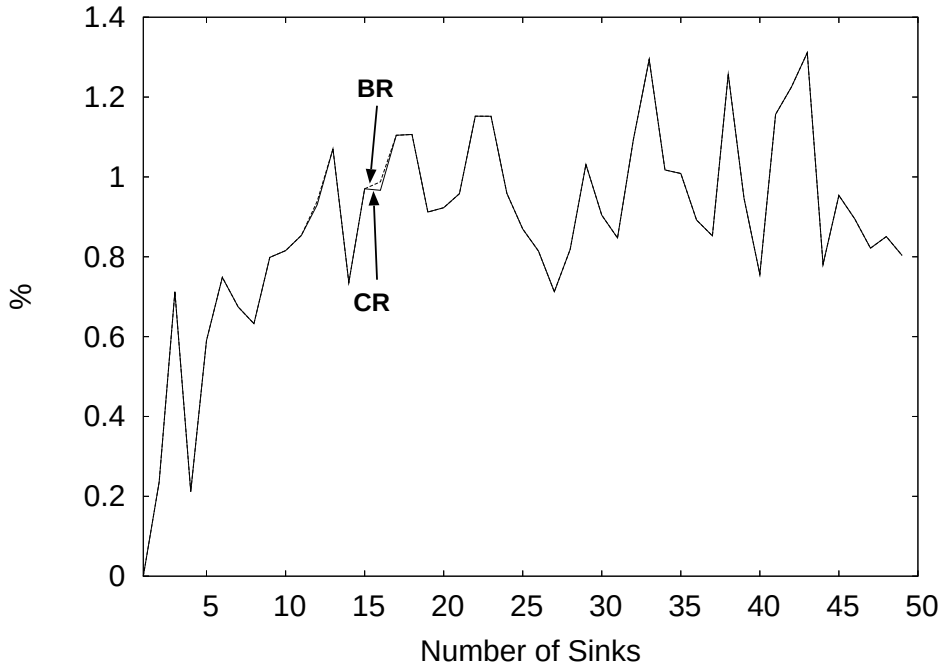
with an average improvement of 0.02%. This result indicates that negative cost bicycles (or other negative cost multi-cycles) are less likely to exist in torus networks, especially in uniform torus networks. The local optimal flows obtained by the cycle reduction and bicycle reduction algorithm have marginal difference, and they offer only very small improvements to the LDF solutions, while the two local search algorithms have much higher computation complexity. Thus, the LDF algorithm offers solutions very close to local optimal, while more time consuming local search algorithms with cycle and bicycle reductions only provide marginal improvements.

Figure 3.19 shows the simulation results on the national network topology. The curves in the charts are similarly labeled as the previous charts for torus networks. Figure 3.19(a) shows the ratio of the cost of the solution produced by different algorithms to the estimated lower bound, as the number of sink vertices increases from 1 to 50. For large numbers of sink vertices, the LDF algorithm produces solutions costing no more than about 1.75 times the estimated lower bound. Both cycle reduction algorithm (LDF-CR) and bicycle reduction algorithm (LDF-BR) improve the LDF solutions, but the bicycle reduction algorithm offers very marginal improvements beyond the cycle reduction solutions, as this is more clearly showed in Figure 3.19(b), the percentage improvements of the bicycle reduction algorithm over the original algorithm when the number of sink varies. First, it shows that the improvements by the cycle and bicycle reduction algorithms in the national network ($\leq 1.4\%$)are less than in the torus networks ($\leq 2.7\%$). It also shows that the cycle reduction algorithm improves the LDF solutions by an average of 1%, and the bicycle reduction algorithm does not offer further improvement in most cases, and very small improvement in small number of cases. These results are largely because the national network is very parse, creating a greater degree of incidental path sharing. In such a sparse network, the LDF algorithm usually is sufficient to create solutions close to optimal, as the more complex local search algorithms can not offer much improvements.

Figure 3.20 shows the simulation results on randomly generated networks. The relative cost results (Figure 3.20(a)) show that LDF produces results up to 2.6 times the estimated lower bound when there are 50 sink vertices, while the cycle reduction and bicycle reduction algorithm both improve the quality of the LDF results, and the bicycle reduction algorithm provides more consistent improvement to the solutions produced by the cycle reduction algorithm. As Figure 3.20(b) shows more clearly that the cycle reduction algorithm improves up to 1.2% over the LDF solutions, and the bicycle reduction algorithm can improve up to

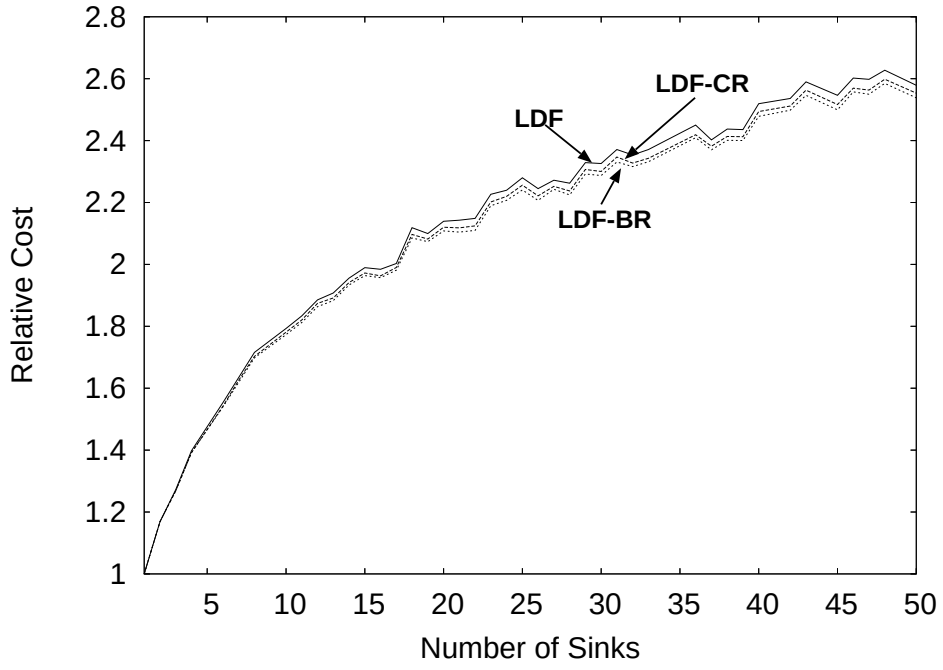


(a) Relative cost comparison.

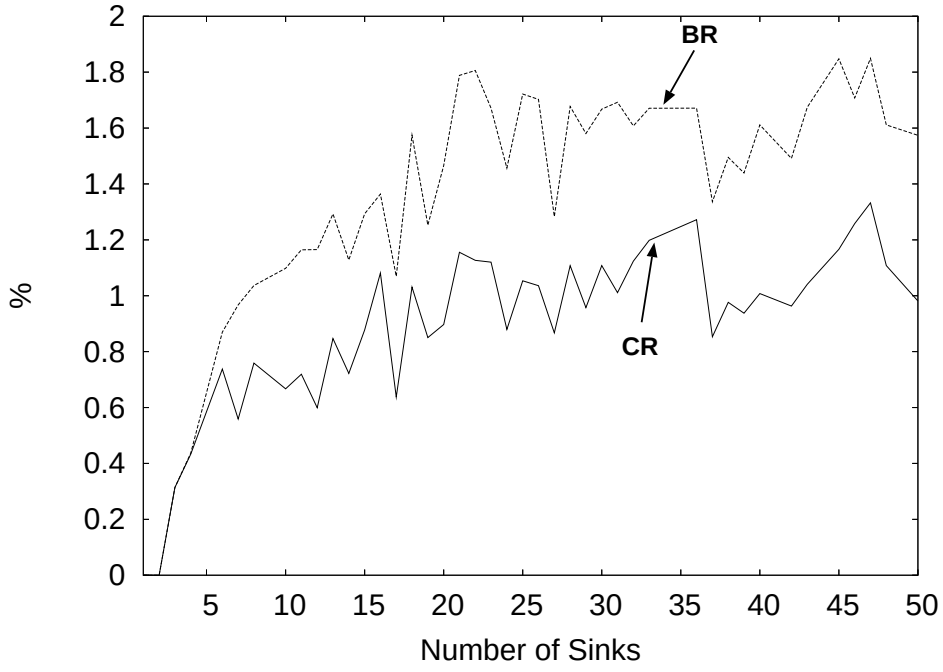


(b) Percentage cost improvement.

Figure 3.19: Cost comparison on the national network.



(a) Relative cost comparison.



(b) Percentage cost improvement.

Figure 3.20: Cost comparison on random networks.

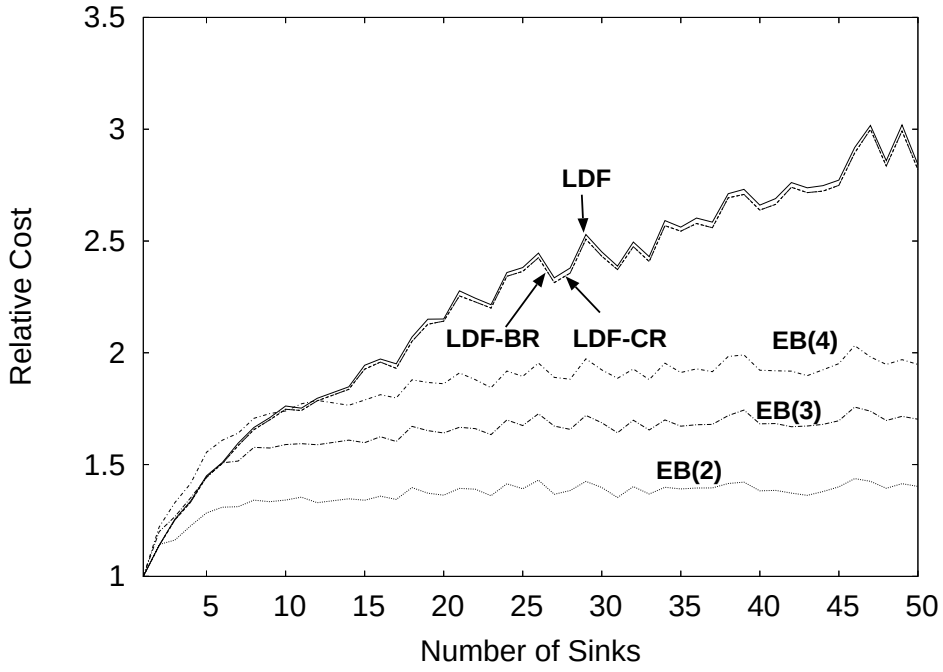
1.85% over the LDF solutions. In addition, the bicycle reduction algorithm offers up to twice the improvement than the cycle reduction algorithm in most cases, the biggest improvement by bicycle reduction algorithm among the topologies simulated. This indicates that there are more negative cost bicycles in the randomly generated and relatively dense networks than the more regular torus networks and sparse national network.

Figure 3.21 shows the cost comparison on the networks generated by the Inet topology generator [82]. It shows similar improvements of the cycle and bicycle reduction algorithms as the number of sink vertices increases in Figure 3.21(a). Figure 3.21(b) shows that both the cycle and bicycle reduction algorithms offer small improvements ($\leq 1.2\%$) over the LDF results, and the bicycle reduction algorithm only improves over the cycle reduction results very marginally in a small number of cases. This is an indication that the small topologies generated by Inet with default parameters are relatively sparse, and thus contain less negative cost cycles and even less bicycles.

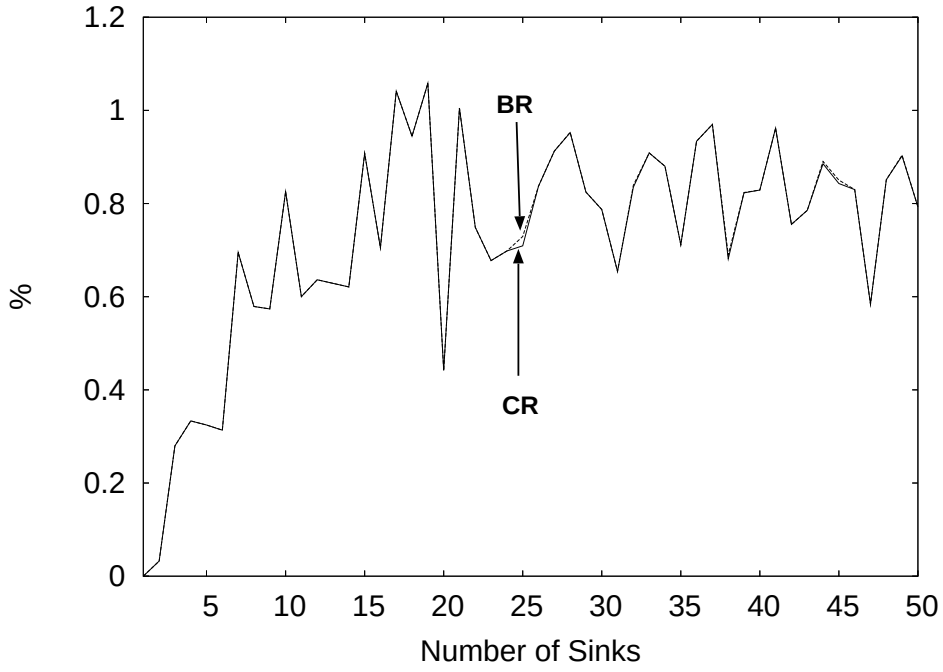
3.3.6 Negative Cost Multi-cycles Reduction

Negative Cost Multi-cycles

Besides negative cost cycles and bicycles, in a network with concave edge costs, there could exist negative multi-cycles that could transform an existing flow to flows with lower costs. We define a *negative cost multi-cycle* as a group of m directed cycles that share a common segment, with the remainder of the cycles edge disjoint. When we add flow along the m cycles, the total cost of the resulting flow is lower than the original flow. We refer to such a multi-cycle as negative cost m-cycle. For some special cases, when $m = 2$, it is a negative cost bicycle; when $m = 3$, it is a negative cost tricycle. A general negative cost multi-cycle is illustrated in Fig. 3.22 with a pair of vertices a and b. There is a common path P_0 from a to b, and m paths from b to a. The sum of the cost of all these path is negative. Let d_i be the length of path P_i. Then, the incremental cost of adding a unit flow along the multi-cycle could be expressed as $(1 + \epsilon)d_0 + \sum_{i=1}^{m} d_i$, where $0 \leq \epsilon \leq m - 1$. If $\epsilon = m - 1$, the cost of the multi-cycle is equal to the sum of the cost of all m cycles with the usual definition of flow costs. If $\epsilon = 0$, it is only charged once for the shared segment. Any other $0 < \epsilon < 1$ would result in a cost falls in between, showing the benefits of path sharing.



(a) Relative cost comparison.



(b) Percentage cost improvement.

Figure 3.21: Cost comparison on networks generated by inet topology generator.

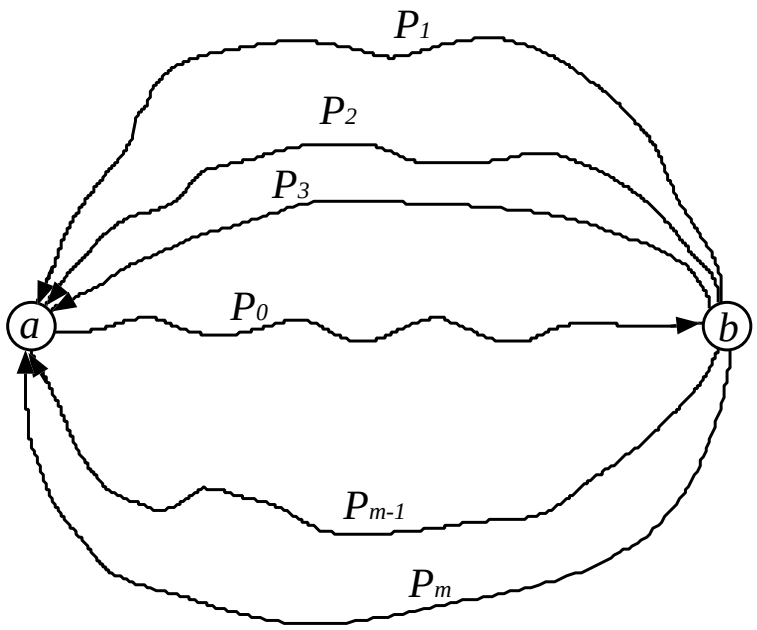


Figure 3.22: Negative cost multi-cycles.

General Multi-cycle Reduction Algorithm

The bicycle reduction algorithm can be further extended to handle negative cost multi-cycles. In particular, we can find the adjacent extreme flows of an existing extreme flow by redirecting flow along a negative cost multi-cycle with k cycles and a common path segment, or negative cost k-cycle.

We pick k tree vertices $(v_1, v_2, \cdots, v_k)$, and search for the optimal split points inside the existing flow to redirect flow through k paths out of the existing flow. In particular, for each of the k vertices, we first determine the undirectional paths to to all potential split points in the tree. These potential split points are similarly defined as in the bicycle reduction algorithm, namely, all tree vertices that are neither in the subtree of any one of the k vertices nor on the undirectional paths between any pair of the k vertices.

For each of the potential split point, construct a subtree rooted at that split point connecting all k vertices, and sum up the total incremental cost of redirecting flows originally into the root of the subtree in a similar way to the computation in the bicycle reduction algorithm. Note that in this case, some of the k vertices share some edges on their paths to the root.

The flow increment on these edges is the sum of the flow into these subset of vertices in the existing flow. As a result, each potential split point has an associated total incremental cost from the k vertices.

Next, for rach of the k vertices, construct a subgraph from the subgraph induced by the non-tree edges in the existing flow, in which edges into the tree vertices except the ones into the chosen vertex are removed as well as the edges leaving the tree vertices either in the subtrees of the k vertices or on the paths between any pair of vertices in the k vertices. In the constructed subgraph, assign the incremental cost of adding the flow into the chosen vertex as the edge length on an edge. Apply the single destination shortest path algorithm in the subgraph to the chosen vertex, and record the shortest distances and paths from all the potential split points.

After all k vertices are processed, each potential split point has k shortest paths to the k vertices. Pick the potential split point u with the least total distance as the split point. Once the split point is chosen, redirect the flow from the k vertices to u along the paths in the original tree, and also redirect flows to the k vertices along the recorded paths recorded in the constructed subgraphs. If the modified flow has a lower cost than the original flow, repeat the above procedure; otherwise, restore the original flow, and stops with the local optimal solution.

This generalized multi-cycle reduction algorithm describes the cycle reduction algorithm when $k = 1$ and the bicycle reduction algorithm when $k = 2$. Clearly, it has much greater complexity when k grows larger for general multi-cycle reduction. However, as our simulation results indicate, the degree of quality improvements is limited even for bicycle reduction algorithm. As k increases, we would expect more diminishing returns for increased complexity. Therefore, we think bicycle reduction would be sufficient for most practical problems, while the general multi-cycle reduction algorithm has theoretical values but may not be necessary for practical problems.

3.4 Summary

This chapter studied the configuration problem for a basic RDS with a single server. The configuration problem was formally defined and formulated as a minimum cost flow problem. A concave link cost model is used in the problem formulation to capture the bandwidth economy of aggregation. However, the concave link cost also makes the configuration problem an NP-hard problem. An efficient approximation algorithm, Largest Demand First (LDF), has been proposed based on the Least Cost Augmentation algorithm for practical network configuration problems with hundreds of nodes.

The second part of this chapter studied local search heuristics to improve the quality of an existing solution for RDS configuration problem, which can be extended to the general minimum concave cost network flow problem (MCCNFP). The original cycle reduction algorithm proposed by Gallo and Sodini [27] only searches for adjacent extreme flows reachable from an existing flow by redirecting flow along negative cost cycles. A path compression technique was implemented to improve the original cycle reduction algorithm such that it only has to compute at most $2m$ shortest path trees, where m is the number of sink nodes, compared with n (n is the total number of vertices) shortest path tree computations in the original algorithm.

In addition, it is shown in this chapter that there exists *negative cost multi-cycles* in a concave cost network with an existing flow. By redirecting flow along these multi-cycles, more local optimal extreme flows can be reached. We present a multi-cycle reduction algorithm by identifying the negative cost multi-cycles and redirecting flow along these multi-cycles to get a local optimal extreme flow. Although we focus on the identification of negative cost bicycles, we describe how it can be extended to negative cost multi-cycles. We study the performance of the bicycle reduction algorithm using simulations on different topologies. The experimental results as well as our analysis show that the bicycle reduction can improve the quality of results, but it would reach a point of diminishing return as the quality improvement is limited but the computational complexity grows when we attempt to identify more general negative cost multi-cycles.

Chapter 4

Multi-server RDS

The previous chapter studied the configuration problem for an RDS with a single server. This chapter will study the issues that arise in an RDS with multiple servers. Such an RDS uses multiple distributed replicated servers to reduce the transmission latency and improve service quality and reliability.

The first part of this chapter studies the configuration problem for multi-server RDSs. Based on a similar problem definition as in our study of the single-server RDS, we show that the configuration problem for a multi-server RDS can be transformed into a single-server RDS configuration problem with the introduction of a pseudo source. However, what makes the multi-server RDS configuration problem more complicated than a single-server RDS configuration problem is the selection of optimal server locations. Therefore, a number of server placement algorithms designed for a variety of network applications are surveyed and evaluated. A series of simulation studies are conducted in various networks, and our simulation studies indicate that among all the server placement algorithms, one class of greedy algorithms gives close to optimal results.

The second part of this chapter studies the configuration problem for dynamic load redistribution in order to improve the fault tolerance of a multi-server RDS. A redirection subnetwork topology is presented that handles any single-server failure in a group of servers, while minimizing the amount of additional bandwidth that must be reserved. An algorithm is first presented to configure such a redirection subnetwork for server pairs so that if one server fails, the other server can handle the traffic redirected from the failed server. This configuration algorithm is then extended to configure redirection subnetworks for groups of four servers such that if any server fails in such a group, the traffic to the failed server

will be redirected to the other three servers through the redirection subnetwork. Simulation studies are conducted in a variety of network topologies to evaluate the redirection subnetwork configuration algorithms. Our simulation results reveal that the proposed redirection subnetworks can handle dynamic load redistribution for single-server failures while making efficient use of reserved bandwidth.

The chapter is organized as follows: Section 4.1.2 reviews some related work with respect to the server placement problem. Section 4.1.3 gives a more detailed definition of the configuration problem for a multi-server RDS, and formulates the configuration problem as a single source minimum cost network flow problem. Section 4.1.4 surveys and compares a number of candidate server placement algorithms originally designed for other network applications in the literature. Section 4.1.5 lays out the evaluation studies we conduct for various server placement algorithms, and presents the simulation results and our analysis. In the second part of this chapter, Section 4.2.1 describes the load unbalance problem in a multi-server RDS, and introduces the configuration problem for redirection subnetworks to handle these situations. A redirection subnetwork topology and a configuration algorithm are presented to find redirection subnetworks for the simple case of redirection server pairs in Section 4.2.2. Section 4.2.3 extends the redirection subnetwork configuration algorithm for server pairs to handle a single-server failure in groups of four servers. Section 4.2.4 shows the results of our simulation studies and Section 4.3 summarizes this chapter.

4.1 Multi-server RDS Configuration

4.1.1 Introduction

The previous chapters have introduced the concept of a reserved delivery subnetwork (RDS) as a new network service to allow an information service provider to deliver more consistent quality of service to its customers. Until now, we have focused our attention on the configuration problem for an RDS with a single central server. However, when there is a large number of customers in many distributed areas for a certain service, an information service provider may find it advantageous to place multiple replicas of the server at separate locations. From the customers' perspective, this reduces the transmission latency and

hence increases the perceived quality of service. From the information service provider's point of view, the additional replicated servers eliminate the single point of failure in the RDS, and release the bandwidth tied up on the long haul connections from a central server to various remote locations. These benefits of improved quality of service and bandwidth efficiency can offset the cost of deploying the replicated servers.

For a multi-server RDS, the configuration problem is similar to that for a single-server RDS, but clearly becomes more complicated as it involves two major additional subproblems to solve. First, we must determine where to put the servers to obtain optimal performance, whereas the server location is fixed for the single-server RDS configuration problem. Second, after we determine the locations of the servers, we need to decide how different locations should connect to a server to get good performance and overall cost efficiency, while all locations connect to the one server in the single-server RDS. The choices for server placement and sink partitioning strategies are vital to the configuration of a multi-server RDS.

4.1.2 Multi-server RDS Configuration

Many network applications have to deal with some form of placement problem similar to the server placement problem in a multi-server RDS. To determine a suitable server placement algorithm in a multi-server RDS, we surveyed a variety of placement algorithms developed for a broad range of network applications. These placement algorithms serve as a basis for our evaluation of our placement algorithm.

Qiu, Padmanabhan and Voelker [60] first studied a web server replicas placement problem that is similar to our server placement problem in a multi-server RDS, although they looked for a relatively dynamic placement solution for periods of 24 hours, as in contrast to our longer operation time frame for RDSs. They formulated the placement problem as an uncapacitated K-median problem, and evaluated four algorithms (tree-based algorithm, greedy algorithm, random algorithm, and hot spot algorithm) in synthetic random graphs as well as Internet topologies (derived from BGP routing information) with actual web server trace data. Their simulation results showed that a greedy algorithm that places replicas based on distance and sink demands consistently delivered the best performance across the network topologies tested.

Jamin et al [39] investigated a similar problem of constrained mirror placement in the Internet. They studied the correlation between the number of mirrors (equivalent to the replicated servers) in a limited number of sites and the performance improvement perceived at both the server and client sides for different placement algorithms. Their results showed a diminishing return as the number of mirror sites increases. Radoslavov, Govindan, and Estrin [66] later extended the evaluation of the fanout-based replica placement algorithm with more accurate network topologies, and found similar results.

Jamin et al [38] described two instrumentation center placement algorithms in a network with known topology: a greedy algorithm based on the l-hierarchically well-separated trees (l-HST) and an approximation minimum K-center algorithm. The first algorithm recursively divides the graph into small partitions with decreasing partition radii, and places a center for a partition that is sufficiently small. When they formulated the center placement problem as a minimum K-center problem in a graph $G = (V, E)$, a 2-approximate algorithm finds the subgraph G_i^2 with K stars as the approximate solution, where $G_i^2 = (V, E_i^2)$ is the graph that contains all the vertices such that there is an edge $(u, v) \in E_i^2$ if there are no more than two hops between u and v in E, and E_i is the set of i edges with least cost in an increasing order.

Shi and Turner [72] looked into the server placement problem in overlay networks. They formulated the placement problem as a set cover problem, and compared the solutions by both a linear programming relaxation and greedy heuristics with simulations on random graphs as well as geographic graphs.

There has also been a substantial amount of research on web proxy and web cache placement for more restricted network topologies. For example, when Li and his colleagues investigated the optimal placement problem for web proxies in the Internet in [50], they assumed the underlying network topologies are trees, and solved the proxy placement problem with a dynamic programming approach. Korupolu, Plaxton and Rajaraman [42] proposed a constant-factor approximation web cache placement algorithm that works for hierarchical cooperative web caching. Although they formulated the placement problem as a minimum cost flow problem, the algorithm only works for hierarchical cache placement.

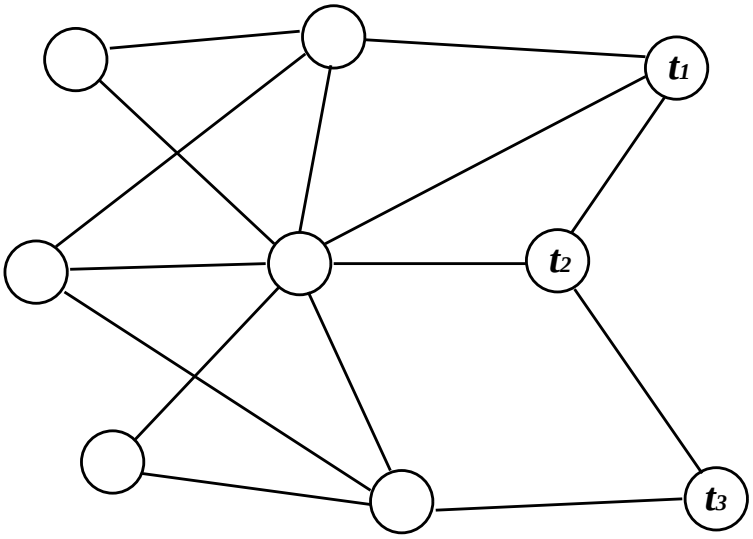
One common feature of all these server placement algorithms is that they use a single measurement metric for evaluating the solutions. Although the measurement metric is the

distance between a sink vertex and the server in most cases, it could also be other similar metrics such as latency. However, the measurement metric does not consider the traffic loads on paths. As a result, they produce solutions that do not take advantage of the benefits of traffic aggregation in practice. In our study of the configuration for a multi-server RDS, we use the cost metric that incorporates both the distance and traffic loads, just as the cost metric used in the single-server RDS configuration.

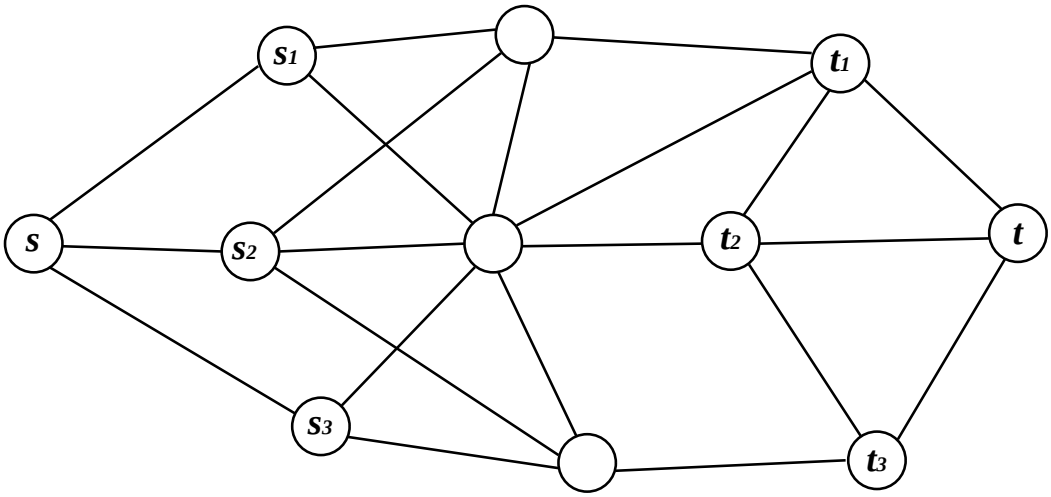
4.1.3 Problem Definition and Formulation

With the existing notations from the preceding chapter, the configuration problem for a multi-server RDS can be defined as follows: we are given a directed graph $G = (V, E)$, an integer k and a set of sinks $T = \{t_1, t_2, \cdots, t_m\} \subseteq V$ with each sink t_i having an associated demand μ_i. The objective is to partition T into k subsets $T_1, \cdots, T_k$ and find a directed tree for each T_i with root s_i. The tree for T_i should include all elements of T_i. The cost of a tree is determined by the flow on its links needed to satisfy the sink demands.

This formulation of the multi-server RDS configuration problem resembles the formulation of the configuration problem for a single-server RDS described in the preceding chapter, except that there is more than one possible location to place the servers, and the server locations are not specified in advance. Assuming that we already have the k server locations, we can apply the following transformation to convert the multi-server RDS configuration problem into a single-server RDS configuration problem, and eventually a single-source single-sink minimum cost network flow problem: first, we add a pseudo source vertex s, and connect s to all vertices corresponding to the server locations. Each added edge has a length of 0 and capacity equal to the total sink demands. After this step, the multi-server RDS configuration problem is transformed into a single-server RDS problem, in which s is the root vertex. Next, add a pseudo sink vertex t, and connect all sink vertices to t. Each added edge has a length of 0 and capacity equal to the demand of the connected sink vertex. This transformation step is the same as in the single-server RDS configuration problem, after which the problem is transformed into a traditional single-source single-sink minimum cost network flow problem. An optimal solution for the transformed minimum cost network flow problem corresponds to an optimal solution for the original configuration problem for multi-server RDS. Figure 4.1 shows the transformation of a multi-server



(a) Original graph.



(b) Transformed graph.

Figure 4.1: Problem transformation.

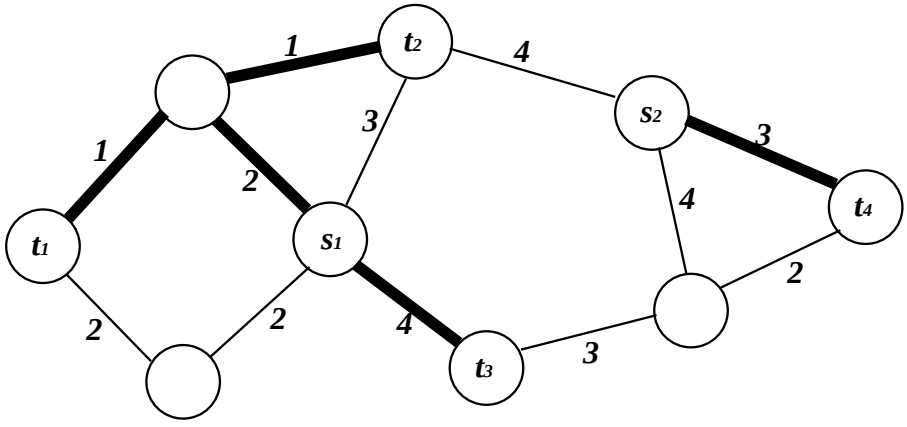
RDS configuration problem into a single-source single-sink network flow problem. In this example, we are trying to place three servers in a network with three sink vertices. The original network is shown in Figure 4.1(a) with sink vertices marked as t_1, t_2 and t_3. After the transformation, the network is shown in Figure 4.1(b) with an added pseudo sink vertex t, an added pseudo source vertex s and edges connecting s to a chosen subset of server vertices s_1, s_2 and s_3.

4.1.4 Server Placement in a Multi-Server RDS

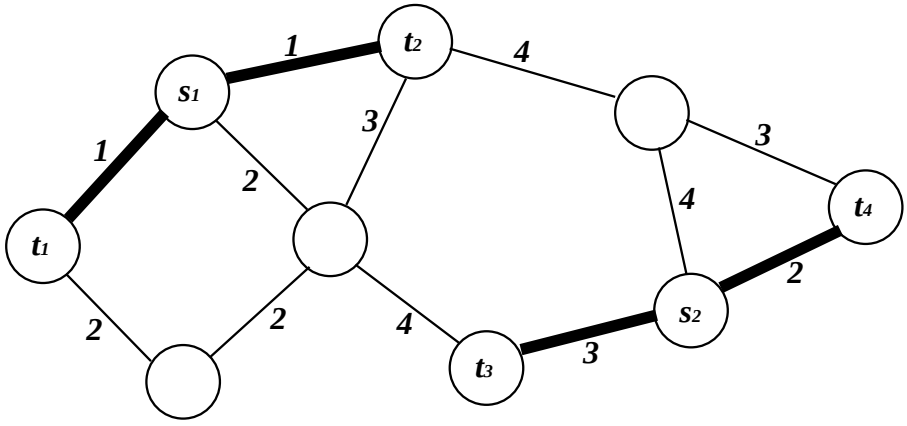
If the locations of the k servers are given in advance, it is clear that the configuration problem for such a multi-server RDS is essentially equivalent to the single-server case because every sink connects to one of the k servers in the solution in the derived single-server RDS configuration problem, and the subsets of the sinks connected to the k servers defines a partition on the sinks. However, the k server locations are unknown for the multi-server configuration problem. Therefore, the key to the multi-server RDS configuration problem is to find good sets of servers.

Take the simple network in Figure 4.2 for example. There are four sinks (t_1 through t_4), each with a unit sink demand. The link lengths are shown next to each link. Assume the link cost is $f(\mu) = l \cdot (\mu + 3\mu^{1/2})$, where μ is the flow on the link, and l is the length of the link. If we place two servers s_1 and s_2 as in Figure 4.2(a), the total cost of the multi-server RDS is 44. However, if the two servers are placed as in Figure 4.2(b), the total cost is only 28. Therefore, an optimal server placement is important in the configuration of an optimal multi-server RDS.

Placement problems are often encountered in many practical applications such as facility location and telecommunication network resource allocation. They are normally formulated as some forms of graph theoretic problems, such as minimum K-median problem and minimum K-center problem, or some other non-graph problem, such as the set cover problem. Therefore, the solutions to these placement problems are often quite different. In this section, we first survey a number of placement algorithms for similar problems in the literature, and compare them as potential placement algorithms for server placement in a multi-server RDS. In particular, we evaluated the web server replica placement algorithms



(a) Bad choice of server locations.



(b) Good choice of server locations.

Figure 4.2: Comparison of different server placement.

in [60], the constrained mirror placement algorithms in [39], and the Internet instrumentation tracer placement algorithms in [38].

Candidate Server Placement Algorithms

Random placement algorithm is the simplest approach. It simply places replicated servers in k randomly chosen locations. It is normally used for performance comparison with other placement strategies because of its simplicity and random nature, as in the cases of [60, 39, 66].

Transit node algorithm is another simple heuristic used in [39, 66]. It places replicated servers on candidate locations in descending order of their outdegrees. The goal of this heuristic is to place replicated servers at locations that can reach the largest possible number of sink vertices with small latency.

Hot spot algorithm was proposed and studied by Qiu, Padmanabhan and Voelker for web server replica placement [60]. It tries to place replicated servers near the sink vertices that generate the largest bulk of traffic to the servers. All candidate vertices are first sorted by their sink demands, and the replicated servers are then put on the k sink vertices with the highest sink demands.

l-greedy algorithm was proposed by Jamin et al [39] for the constrained mirror placement problem. It first exhaustively checks each possible vertex to identify the vertex that gives the least cost to cover all sink vertices. If $l = 0$, it proceeds to check the rest of the candidate vertices to find another vertex such that the new vertex and the previously selected replicated server vertices together have the minimum cost to cover all sink vertices. This special case greedy algorithm was studied by Qiu, Padmanabhan and Voelker [60] for web server replica placement. If l is non-zero, the algorithm allows for l step(s) backtracking by checking all the possible combinations of removing l of the already placed replicated servers and replacing them with $l + 1$ new replicated servers.

l-HST algorithm was presented by Jamin et al [38] for placing Internet instrumentation tracers for latency measurement. Starting with the whole network as a single partition, this algorithm tries to divides the network into overlapping partitions recursively in the following way: pick an arbitrary vertex in the current (parent) partition, a new (child) partition with all the vertices within a random radius of the chosen vertex is created, and the vertices in the newly created (child) partition are not used for future partitioning in the current (parent) partition. The mean diameter of the child partition is l times smaller than the diameter of the parent partition. This recursive procedure is applied to all partitions until all partitions have only one vertex. When it halts, a hierarchical tree of partitions is formed in which the root node is the partition of the whole network and the leaves are all single-vertex partitions. A virtual node is designated for each partition, and the virtual node of a parent partition and the virtual node of its child partitions are connected using a link with half the parent partition diameter. These virtual node together form a hierarchical tree, and is called a l-hierarchically well-separated tree (l-HST). The choice of a random radius in each partitioning step makes the probability of a short edge being cut by partitioning decrease exponentially as one climbs the tree. Thus, it keeps the vertices close together in the same partition in lower level of the tree.

In order to use l-HST for server placement, a maximum partition diameter bound D is specified to limit the size of a partition. A greedy placement algorithm using l-HST always maintains a list of partitions sorted in the decreasing order of the partition diameter. The greedy algorithm always removes the partition with the largest diameter, and creates two child partitions, one of which contains the vertices with a random radius of a chosen vertex, and the other one contains the rest of the vertices in the parent partition. These child partitions are then inserted in the sorted partition list for further processing. If the largest partition of sink vertices has a diameter less than the maximum partition size D, the algorithm halts, and a set of partitions each with a diameter less than D is obtained. A server can be placed in a vertex of each partition of sink vertices.

4.1.5 Evaluation

To evaluate candidate server placement algorithms for multi-server RDS, we simulate these algorithms on three classes of networks: random networks, unit torus networks and a national network. In a random network, a fixed number of vertices are randomly placed in a unit square, and a fixed number of edges are randomly added between vertices to make it a connected network. The number of edges is a random number between three and four times the number of vertices. If the number of edges reaches the fixed edge limit but the network is still not connected yet, an existing edge is randomly moved to connect two other random vertices. Sink vertices are randomly selected among all vertices, each with a randomly assigned demand in a range. In a unit torus network, each vertex is connected to its four neighbors, forming a rectangular grid with "wrap-around" edges linking the top and bottom rows as well as the leftmost and rightmost columns. All edges have a unit length. The sink vertices are selected randomly with a random sink demand uniformly distributed in a range. In the national network, we use the 50 largest metropolitan areas in the United States [79] as 50 vertices. A set of cities are randomly selected as sink vertices, and each city has a sink demand proportional to its population. The edges of the national network are drawn based on the backbone networks of some national network service providers. Some additional links are added to further increase the network density.

For simulations in random networks, we use random networks with 300 vertices and 100 sinks. For simulations in unit torus networks, we use 15 x 15 unit torus networks with 100 sinks. For simulations in the national network, we randomly select 32 sinks in each simulation. In addition, each data point in the simulation results represents the average value of the results from 100 different problem instances with the same specific parameters (network sizes, number of servers, and number of sinks). The average total costs of RDSs generated by different server placement algorithms are measured and compared.

Simulation Results and Analysis

In Figure 4.3, we compare the total costs of subnetworks created by different server placement algorithms to a lower bound (LB) in a unit torus network with no more than three servers. Because of the small number of servers, we use exhaustive search to obtain the

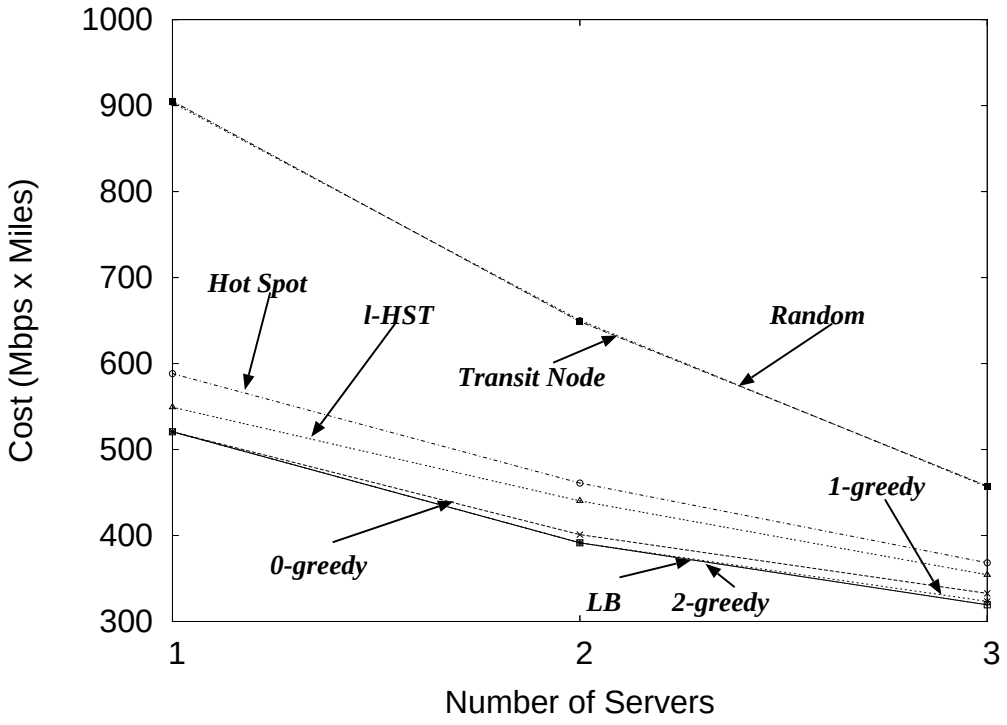


Figure 4.3: Server placement algorithms comparison with optimal solutions obtained by exhaustive searches for smaller numbers of servers in uniform torus networks.

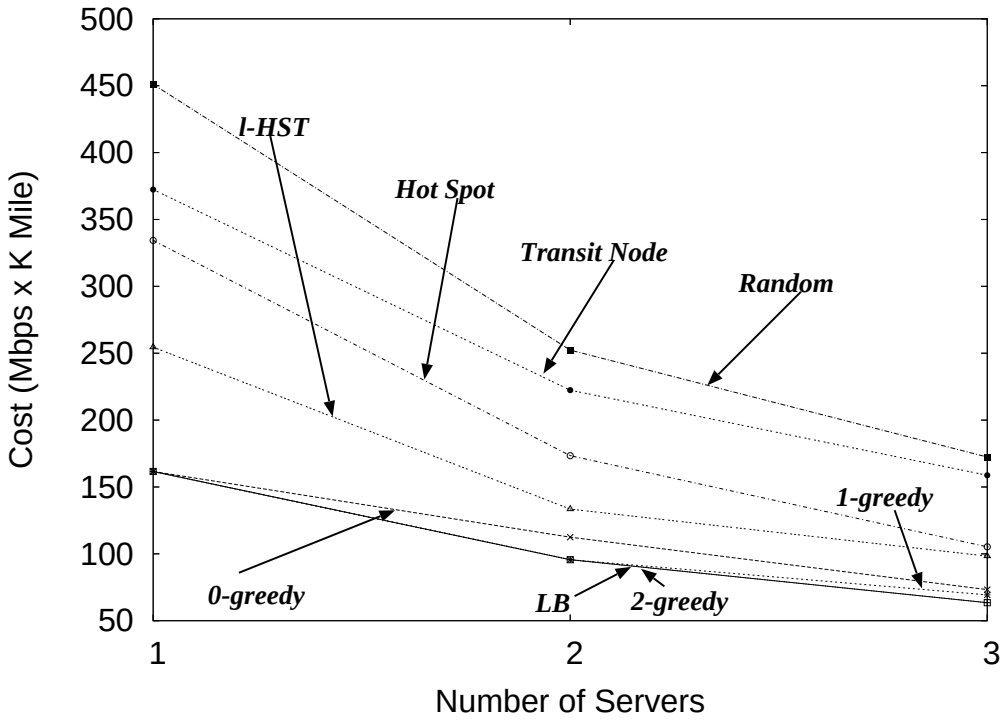


Figure 4.4: Server placement algorithms comparison with optimal solutions obtained by exhaustive searches for smaller numbers of servers in random networks.

lower bounds. As showed in the plot, all classes of greedy algorithms achieve results very close to the optimal solutions. In particular, the 2-greedy algorithm produces the optimal solutions for problems with no more than three servers, because it is essentially conducting exhaustive search just like the lower bound algorithm. Similarly, the 1-greedy algorithm produces optimal solutions for single server RDS. These greedy algorithms produce good solutions because they go through different combinations of server locations to find a good solution, and resemble the search-based algorithms when l becomes large. The l-HST algorithm has very good performance, next to the greedy algorithms. The Hot Spot algorithm does not perform well for networks with smaller numbers of servers, but the performance improves when the number of servers increases. The Transit Node algorithm produces the worst results that are comparable to the random placement algorithm. This is because the outdegree of a vertex in a uniform torus network is always four, and the Transit Node algorithm ends up picking up arbitrary server locations as in the random placement algorithm.

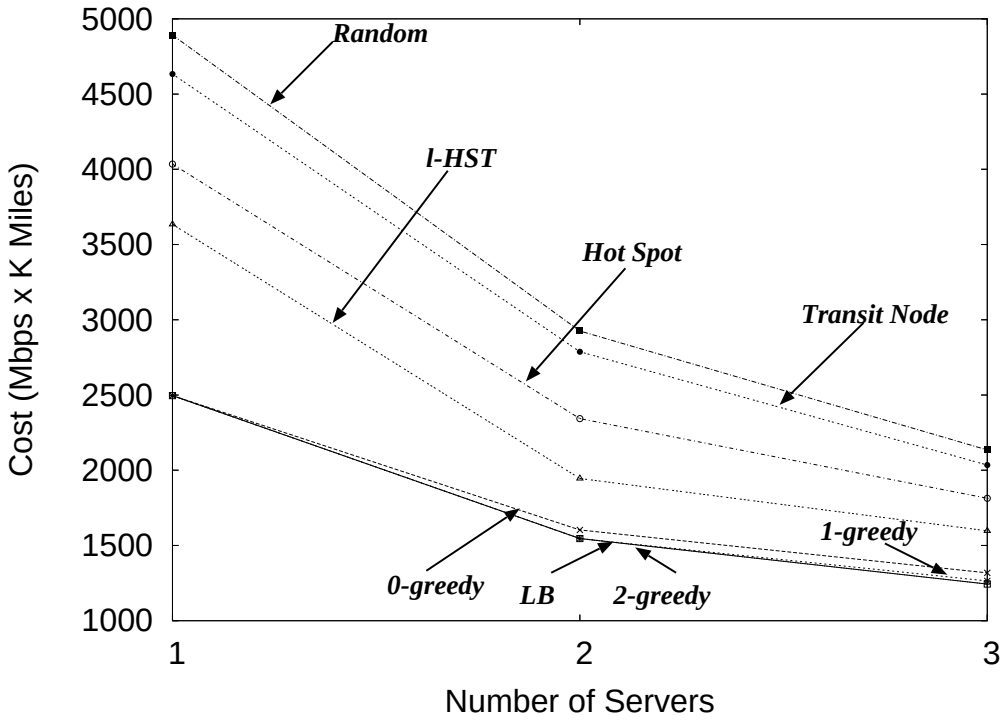


Figure 4.5: Server placement algorithms comparison with optimal solutions obtained by exhaustive searches for smaller numbers of servers in the national networks.

In Figure 4.4, we compare the total costs of subnetworks created by different server placement algorithms to a lower bound (LB) in a random network with no more than three servers. The results are very similar to those in the uniform torus networks, except for the results of the Transit Node algorithm. Specifically, the greedy algorithms have the overall best solutions; the l-HST algorithms produces solutions better than other non-greedy algorithms, but its advantages over the Hot Spot algorithm reduces as the number of servers increases; the Transit Node algorithm has performance only slightly better than the random placement algorithm, showing that placing servers on nodes with large outdegrees does not significantly reduce the cost of the network; the random placement algorithm has the worst overall results.

In Figure 4.5, we compare the total costs of subnetworks created by different server placement algorithms to a lower bound (LB) in a national network with no more than three servers. The results are similar to those in the random and torus networks.

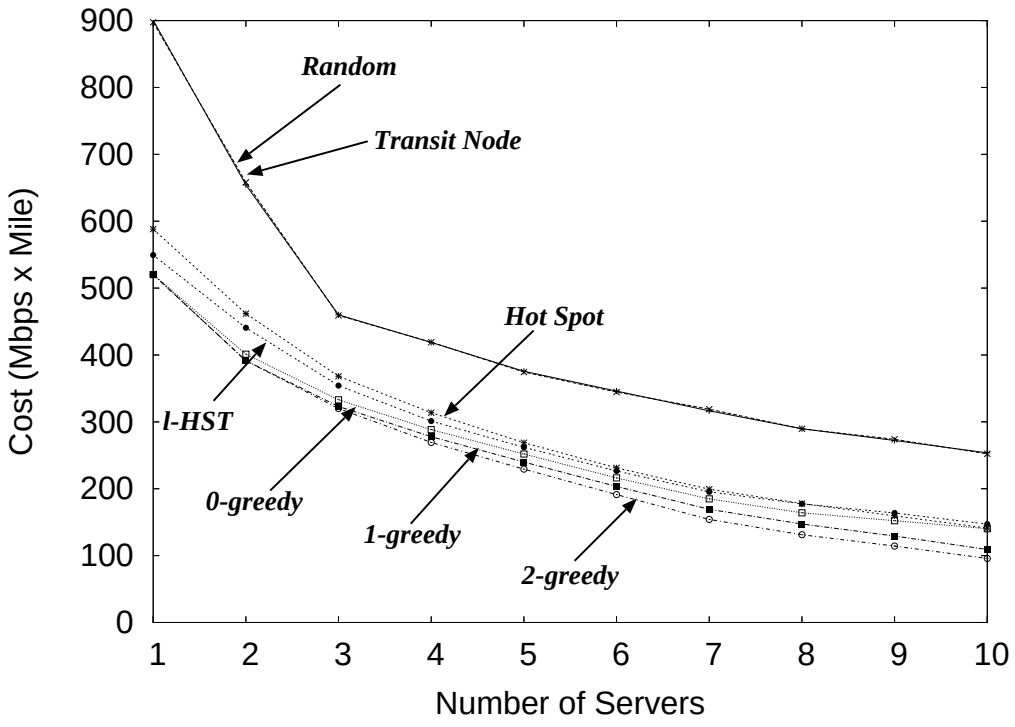


Figure 4.6: Comparison of server placement algorithms in uniform torus networks.

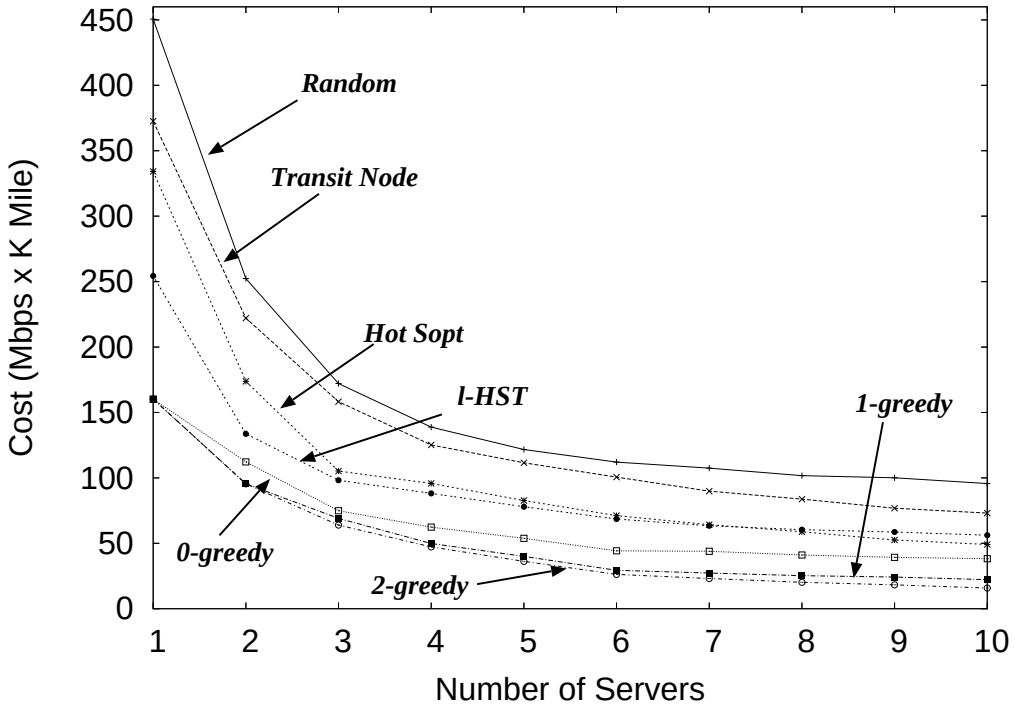


Figure 4.7: Comparison of server placement algorithms in random networks.

Figure 4.6 shows the results of different server placement algorithms in a unit torus network with up to 10 servers. The results are similar to those in Figure 4.3. All greedy algorithms get the best performance. The l-HST algorithm outperforms the other non-greedy algorithm when there are less than eight servers. The Hot Spot algorithm does not perform well when there are small numbers of servers, but the results improve as the number of servers increases. The results improve more quickly than the l-HST algorithm, and the Hot Spot algorithm eventually produces better results when there are more than eight servers. The Transit Node algorithm still performs comparably with the random placement algorithm, which has the worst performance.

Figure 4.7 shows the results of different server placement algorithms in a random network with up to 10 servers. The results are similar to those in Figure 4.4. All greedy algorithms get the best performance. The l-HST algorithm outperforms the other non-greedy algorithm when there are small numbers of servers, but the Hot Spot algorithm eventually improves over it when there are more than seven servers. The Transit Node algorithm still

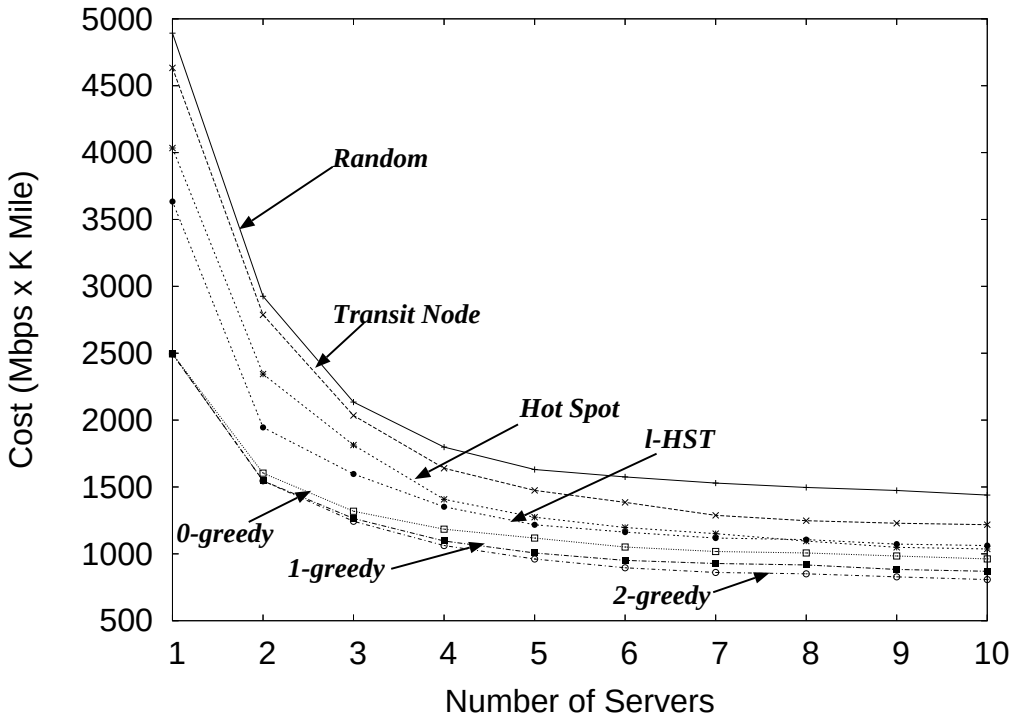


Figure 4.8: Comparison of server placement algorithms in the national network.

performs relatively worse than other non-random algorithms, and the random placement algorithm has the worst performance.

The results in the national network with up to 10 servers showed in Figure 4.8 reveal the similar results as in Figure 4.5. All greedy algorithms get the best performance. The l-HST algorithm outperforms the other non-greedy algorithm when there are small numbers of servers, and the Hot Spot algorithm improves over it when there are more than eight servers. The Transit Node algorithm still performs relatively worse than other non-random algorithms, and the random placement algorithm has the worst performance.

These simulation studies indicate that the l-greedy algorithms can produce solutions with lower costs than the other server placement algorithms. However, because they use an approach approximating an exhaustive search in the solution space, they have high computational complexity when l is greater than 2. On the other hand, solutions created by 0-greedy and 1-greedy algorithms have lower cost than the solutions produced by the other server

placement algorithms with low computational complexity. Therefore, these two greedy algorithms are good choices for determining the server locations for the configuration of a multi-server RDS when the number of servers is much smaller than the number of sinks.

4.2 Dynamic Load Redistribution in Multi-server RDS

4.2.1 Server Load Unbalance in a Multi-Server RDS

In an RDS, software or hardware failure can make a server incapable of providing service to its assigned sinks. In a single-server RDS, such server failure will cause interruption of quality of service to customers. In a multi-server RDS where a number of replicated servers exist, the demands from the customers to the failed server can be redirected with a redirection subnetwork to other unaffected servers which have some extra capacity so that the interruption to quality of service can be minimized. We study the configuration of a redirection subnetwork infrastructure to provide improved tolerance to server failures in a multi-server RDS in the remainder of this chapter.

The redirection subnetworks improve the tolerance to server failures, but they also increase the communication cost because of the additional bandwidth reserved on the links in the redirection subnetworks that is not used under normal conditions. In particular, to configure the redirection subnetwork that handles the failure for a specific server, we can remove the potentially failing server vertex from the network and rerun the multi-server RDS configuration algorithm. The additional edges and extra reserved bandwidth in the newly created RDS constitutes the redirection subnetwork for that server. However, if we use this method to deal with a potential failure of *any* one of the k servers, we would end up with k different redirection subnetworks. Each redirection subnetwork will be used when the specific server fails. It is unlikely that all these redirection subnetworks will share many of their edges and reserved bandwidth, and thus this could incur high communication cost overhead. Although this configuration can ensure that every possible server failure can be handled effectively, the additional cost of the reserved bandwidth on the links in the redirection subnetworks could be prohibitively high.

An alternative solution is to set up a common redirection subnetwork shared by a group of servers, instead of one redirection subnetwork for each server that will fail potentially. If one server fails, the demands of the sinks to the failed server can be redirected only to the other servers in the same group through the redirection subnetwork. Although this type of redirection subnetwork limits the choice of the servers that traffic can be redirected to, and may have higher communication cost than an individual redirection subnetwork dedicated to a specific single server, it reduces the number of separate redirection subnetworks and the total amount of reserved bandwidth in the redirection subnetworks, and can be expected to have a lower cost than the collective total cost of all the individual redirection subnetworks for individual servers in the group. We will focus on this type of redirection subnetwork in our study in this chapter.

We must configure the redirection subnetworks with two goals in mind: first, we must be able to satisfy the demand of sinks affected by the failed server as much as possible. This is largely determined by the amount of additional reserved bandwidth in the redirection subnetwork and the extra capacity of the healthy servers in a server group. Second, we must keep the extra communication cost and bandwidth reservation incurred by the redirection subnetworks as low as possible. The cost of a link in a redirection subnetwork is determined with the same concave link cost function of the average traffic on the link, as in the RDS configuration problem. Thus, reusing the existing RDS links when configuring the redirection subnetworks would reduce the additional cost incurred by the extra bandwidth in the redirection subnetworks.

In order to implement dynamic load redistribution, we first assign each server s_i an extra capacity C_r^i to handle extra demands from the sinks of failed servers. C_r^i can be either a fixed amount or a fraction of the original server capacity. We only discuss the case with fixed extra capacity $C_R^i = C_r$ in this dissertation for simplicity.We also focus our attention on the case of single server failure.

The two subsequent sections will study the configuration algorithms for a redirection subnetwork for groups of servers of different sizes. We start with the study of the simplest redirection subnetworks for server pairs, and then extend our study to redirection subnetworks for groups of four servers. Optimal redirection subnetwork solutions for server pairs can be obtained efficiently because the amount of redirection traffic is fixed between two

servers. When there are more than two servers, an approximation algorithm is needed to find a good solution efficiently for networks of practical sizes.

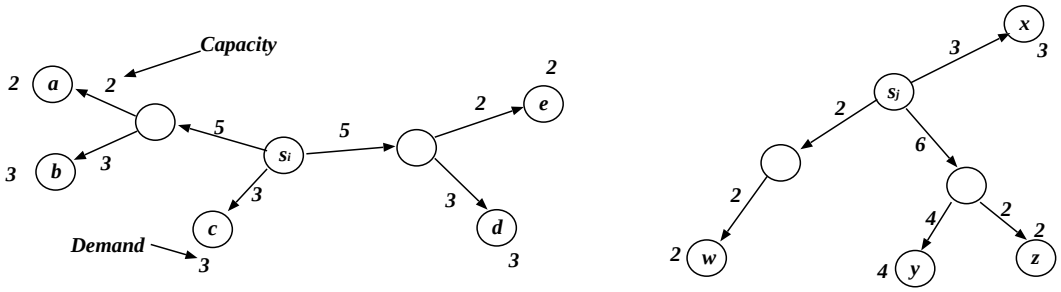
4.2.2 Configuration of Redirection Subnetworks for Server Pairs

Problem Statement

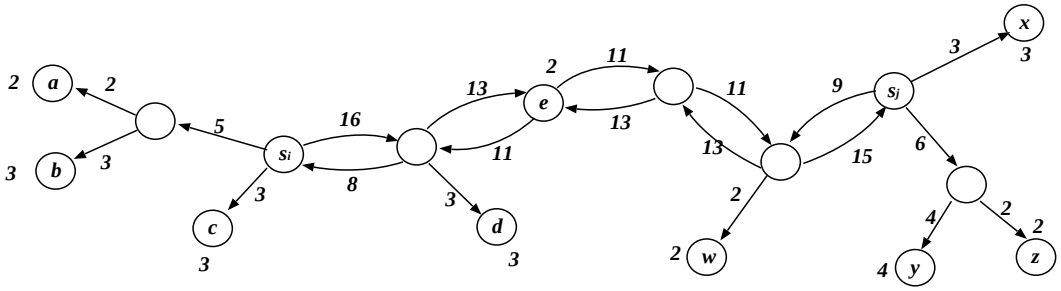
We start with the configuration of the simplest redirection subnetworks in a multi-server RDS. Specifically, we find a peer server for each server to form a server pair in a multi-server RDS. If a server in this server pair fails, the sink demands for the affected server will be redirected to the other server through a common redirection subnetwork for the pair of servers. The additional communication cost is the cost of the reserved bandwidth in the redirection subnetwork for the server pairs. An overall optimal redirection subnetwork includes all the server pairs. We only consider the cases for even numbers of servers.

Using the existing notations from the early part of this chapter, the redirection subnetwork configuration problem for server pairs can be formally specified as follows: given a pair of servers s_i, s_j and their subnetworks of the RDS, $G_i = (V_i, E_i)$ and $G_j = (V_j, E_j)$. A redirection subnetwork is defined by a bidirectional path joining s_i and s_j that can be divided into three parts, P_i, P and P_j, where P_i contains only vertices in V_i, P_j contains only vertices in V_j and P contains only vertices that are in neither V_i nor V_j. Let $G_{ij} = (V_{ij}, E_{ij})$ be the graph formed by combining G_i, G_j and this path. For each edge (u, v) on the path, we require that the edge capacity equal the demand of all the sinks reachable from v in the graph $(V_{ij}, E_{ij} - \{(v, u)\})$. An example illustrating these definitions appears in Figure 4.9. The two original subnetworks shown in Figure 4.9(a) are connected with the redirection subnetwork to form an augmented subnetwork shown in Figure 4.9(b). Note that this changes some of the original capacities. The cost of links in G_{ij} is determined using the same cost function as for the original RDS configuration problem. The cost of G_{ij} is the sum of its edge costs. The cost of pairing s_i with s_j is cost of $G_{ij} -$ (cost of $G_i +$ cost of G_j).

The redirection subnetwork configuration problem partitions the set of servers into pairs, with the objective of minimizing the overall cost and can be solved optimally using a weighted matching algorithm. We define a new complete graph consisting of only servers,



(a) Original subnetworks.



(b) Augmented subnetwork.

Figure 4.9: An example redirection subnetwork for a server pair.

and define the weight of an edge as the cost of pairing the two vertices on both ends of the edge. A minimum weight maximum size matching on this complete graph gives us the least-cost way of pairing the servers.

The algorithm is described in the following pseudo code:

For any server s_i
    Compute incremental cost on all links
    Compute the shortest path tree from s_i using incremental costs as edge lengths
Derive a complete graph G_s of servers only,
    with the least cost between two servers as edge lengths
Find the minimum weight maximum matching M in G_s
For each pair of matched servers (s_i, s_j),
    connect s_i to all sinks in G_j with the paths in the shortest path tree
    connect s_j to all sinks in G_i with the paths in the shortest path tree
end

4.2.3 Configuration for Redirection Server Group

Problem Statement

Although redirection subnetworks for server pairs in a multi-server RDS are simple and produces optimal solutions, it requires that every server must be over-engineered by a factor of 2 to handle the redirected traffic from the peer server in its server pair. If we organize larger groups of servers, we can still recover from any single server failure, but each server in the group requires less extra capacity and handles less extra demand than a server in a server pair. Specifically, If there are m servers in each server group, a server only needs to be over-engineered by a factor of $m/(m-1)$, and has extra capacity to handle $1/(m-1)$ extra sink demand. However, when the groups of servers grow larger, the optimal solution can no longer be efficiently obtained because a solution depends on how the extra capacity is distributed among the servers, which greatly increases the size of the solution space. On the other hand, the search based exact algorithm is impractical for real world network configuration problems, which can have hundreds of sinks. Thus, we study the efficient solutions to configuration problem for redirection subnetworks for groups of more than two servers using approximation algorithms. We limit our study to groups of four servers because it reduces the reserved extra server capacity by $2/3$ and is more practical in real world applications.

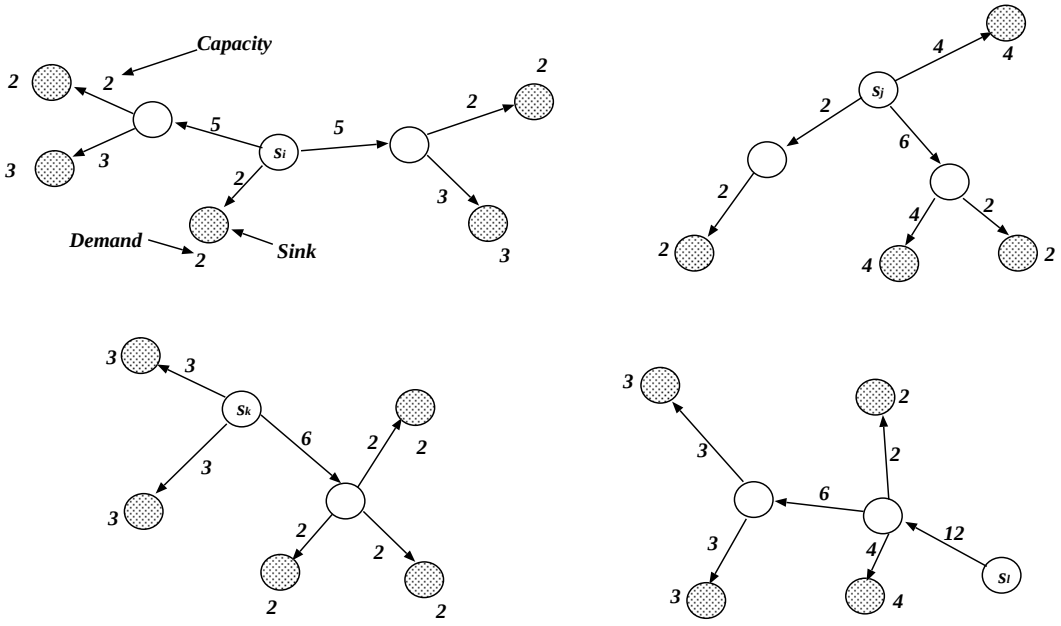
The redirection subnetwork configuration problem for server groups can be similarly described as follows: given four servers s_i, s_j, s_k, s_l and their subnetworks of a multi-server RDS, $G_i = (V_i, E_i)$, $G_j = (V_j, E_j)$, $G_k = (V_k, E_k)$, and $G_l = (V_l, E_l)$. A redirection for the group of four servers is defined by a subgraph that consists of a center vertex C and four bidirectional paths joining C and each of the four servers. Each path can be divided into two parts: the path from C to s_i can be divided into P_i and P_{Ci}, where P_i contains only vertices in V_i, and P_{Ci} contains only vertices not in any of V_i, V_j, V_k, and V_l. Similarly, the path from C to s_j is divided into P_j and P_{Cj}, the path from C to s_k is divided into P_k and P_{Ck}, and the path from C to s_l is divided into P_l and P_{Cl}. Let $G_C = (V_C, E_c)$ be the graph formed by combining G_i, G_j, G_k, G_l and this subgraph. For each edge (u, v) on the path from C to any of the servers, we require that the edge capacity equal the demand of all the sinks reachable from v in the graph $(V_C, E_C - \{(v, u)\})$. For each edge (u, v) on the path from s_i to c, we require that the edge capacity equal to one third of the maximum

total demand to s_j, s_k and s_l plus the demand of all the sinks reachable from v in the graph $(V_i, E_i - \{(v,u)\})$. The capacity on edges from s_j, s_k and s_l are similarly assigned. An example illustrating these definitions appears in Figure 4.10. The four original subnetworks shown in Figure 4.10(a) are connected to a center vertex C with the redirection subnetwork paths to form an augmented subnetwork shown in Figure 4.10(b). Note that this changes some of the original capacities. The cost of links in G_c is determined using the same cost function as for the original RDS configuration problem. The cost of G_C is the sum of its edge costs. The cost of connecting the four servers is cost of $G_C -$ (cost of G_i+ cost of G_j+ cost of G_k+ cost of G_l). The redirection subnetwork configuration problem partitions the set of servers into groups of four servers, with the objective of minimizing the overall cost and can be approximately solved using an approximation algorithm.

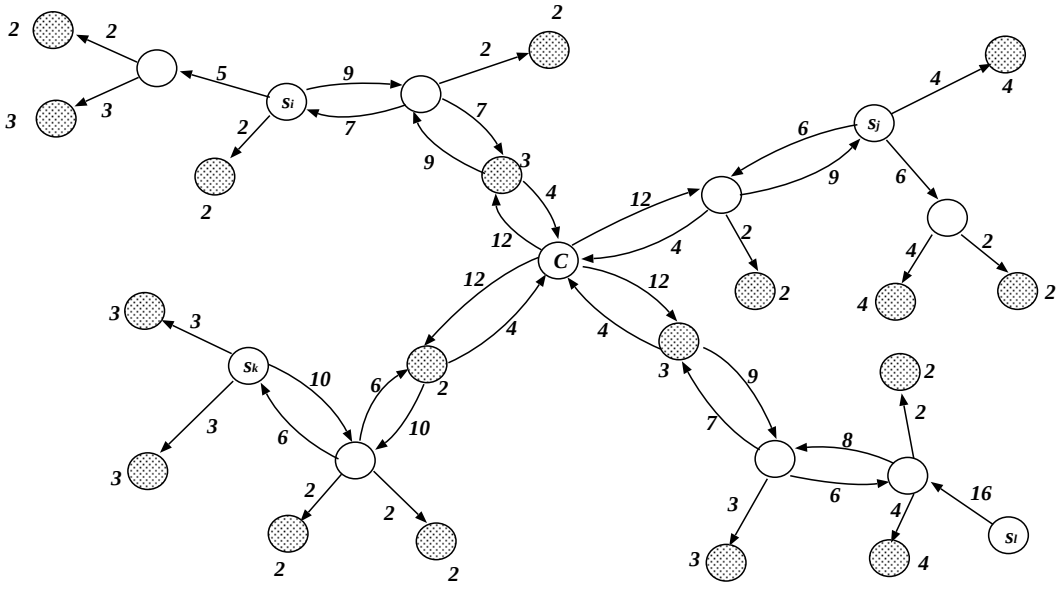
Configuration Algorithm

The redirection subnetwork configuration for a group of four servers starts with the redirection subnetwork configuration for server pairs. First, the optimal server pairs are found using the configuration algorithm for server pairs. This reduces the number of four-server combinations that have to be checked. Instead, only pairs of the optimal server pairs are checked to determine the four-server groups. Specifically, we take each pair of server pairs obtained in the first phase, (s_i, s_j) and (s_k, s_l), and find the best redirection subnetwork for this set of four servers. The best redirection subnetwork for the four servers is determined by trying all possible center vertices for the four servers, and using the center vertex resulting in the least cost redirection subnetwork. When all the best redirection subnetworks for all server pairs are determined, use the cost of the redirection subnetwork for the server pair (s_i, s_j) and (s_k, s_l) as the weight joining (s_i, s_j) with (s_k, s_l) on a complete graph consisting all server pairs from the first phase. We then find the minimum weight maximum size matching in this derived complete graph, and the resulting matching corresponds to the groups of four servers for the best redirection subnetworks.

When the groups of four servers are determined as pairs of server pairs, we then use the center vertex for each matched pair of server pairs as the center node for the group of the four servers in the pair of server pairs. The incremental cost from the center node to each server of the four subgraphs is C_r, and the incremental cost from each of the four server to the center node is $C_r/3$. The total incremental cost for a specific group of four servers is



(a) Original subnetworks.



(b) Augmented subnetwork.

Figure 4.10: An example redirection subnetwork for a four-server group.

the sum of total cost of the redirection subnetwork connecting the center node and all four servers.

The following pseudo code describes the configuration algorithm:

Run configuration algorithm for server pairs
For every pair of server pairs (s_i, s_j) and (s_k, s_l) produced in the first phase,
  For each vertex c,
    Compute incremental cost with flow increment of C_r
      between c and each of s_i, s_j,s_k, and s_l
    Compute shortest path tree with incremental costs as edge lengths
    Compute the cost of the resulting shortest path tree
  Find the center node C for s_i, s_j,s_k, and s_l with the least cost
Create the complete graph G' of all server pairs
Find the minimum weight maximum matching M in G'
For each group of the matched servers,
  Connect the center node C of the four server to the subgraphs
    with appropriate reserved bandwidth
end

4.2.4 Experimental Results

We study our redirection subnetwork configuration algorithm with simulation studies on three classes of networks: random networks, national networks, and unit torus networks. In a random network, a fixed number of vertices are randomly placed in a unit square, and a fixed number of edges are randomly added between vertices to make it a connected network. The number of edges is a random number between three and four times the number of vertices. If the number of edges reaches the fixed edge limit but the network is still not connected yet, an existing edge is randomly moved to connect two other random vertices. Sink vertices are randomly selected among all vertices, each with a randomly assigned demand in a range. In a unit torus network, each vertex is connected to its four neighbors, forming a rectangular grid with "wrap-around" edges linking the top and bottom rows as well as the leftmost and rightmost columns. All edges have a unit length. The sink

vertices are selected randomly with a random sink demand uniformly distributed in a range. In the national network, we use the 50 largest metropolitan areas in the United States [79] as 50 vertices. A set of cities are randomly selected as sink vertices, and each city has a sink demand proportional to its population. The edges of the geographical national network are drawn based on the backbone networks of some national network service providers. Some additional links are added to further increase the network density.

For simulations in random networks, we use random networks with 300 vertices and 100 sinks. For simulations in unit torus networks, we use 15 x 15 unit torus networks with 100 sinks. For simulations in the national network, we randomly select 32 sinks in each simulation. For simulations in random and unit torus networks, we use networks with 4, 8, 12 and 16 servers; while for simulation in the national network, we use 4, 8 and 12 servers. In addition, each data point in the simulation results represents the average value of the results from 10 different problem instances with the same specific parameters (network sizes, number of servers, and number of sinks). The error bars in the results show the range of these results. The unit of the network cost is Mbps $\times$ Miles.

Figure 4.11 shows an example redistribution subnetwork for a server pair in the national network topology. In this example, there are two servers in Chicago and San Francisco (marked with larger squares), each connecting to a set of sinks (marked with smaller squares) with certain traffic demands. The original RDS links are marked with thick solid lines, and the redirection subnetwork links connecting the two subnetworks through Las Vegas and St. Louis are marked with thick dashed lines.

Figure 4.12 shows an example redistribution subnetwork for a group of four servers in the national network topology. In this example, the four servers are in Chicago, New York City, Orlando, and San Francisco (marked with larger squares), each connecting to a set of sinks (marked with smaller squares) with certain traffic demands. The original RDS links are marked with thick solid lines. The center node is at Oklahoma City (marked with a large circle, and the additional redirection subnetwork links are marked with thick dashed lines.

Figure 4.13, Figure 4.14 and Figure 4.15 show the total costs of the RDSs with and without redirection subnetworks in random networks, national networks and torus network, respectively. The curves labeled “Base RDS” are the costs of RDSs with no redirection

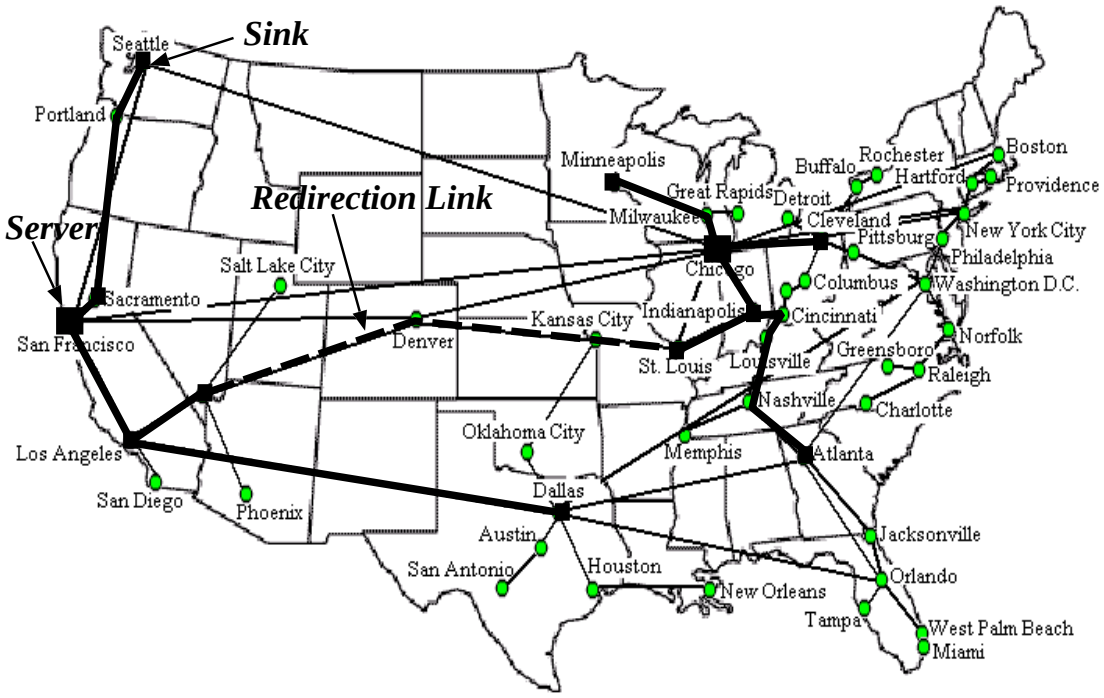


Figure 4.11: An example server-pair redirection subnetwork in the national network topology.

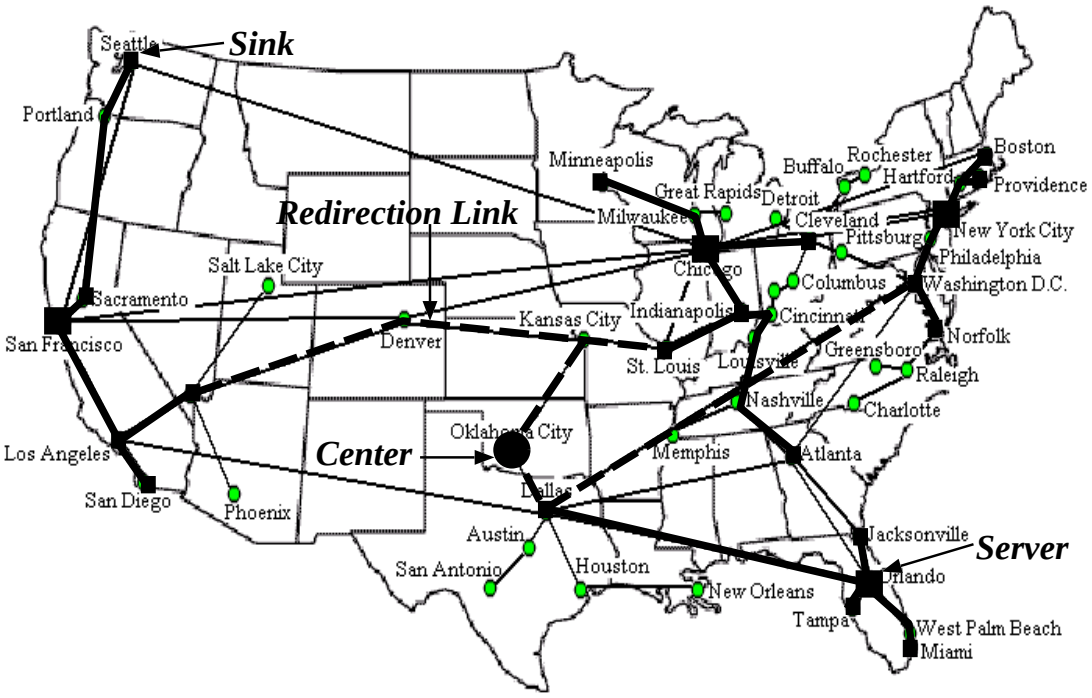


Figure 4.12: An example redirection subnetwork for groups of four servers in the national network topology.

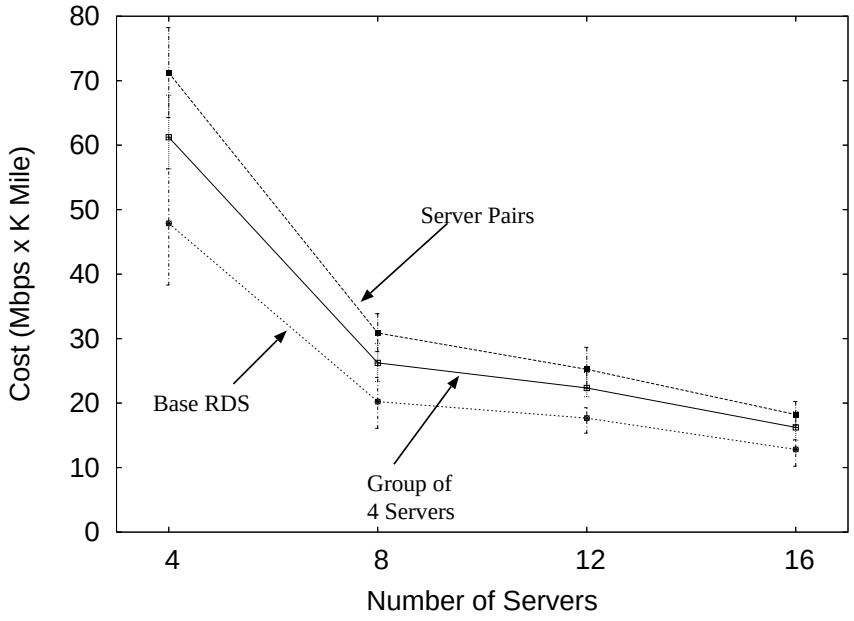


Figure 4.13: Simulation results of redirection subnetwork in random networks.

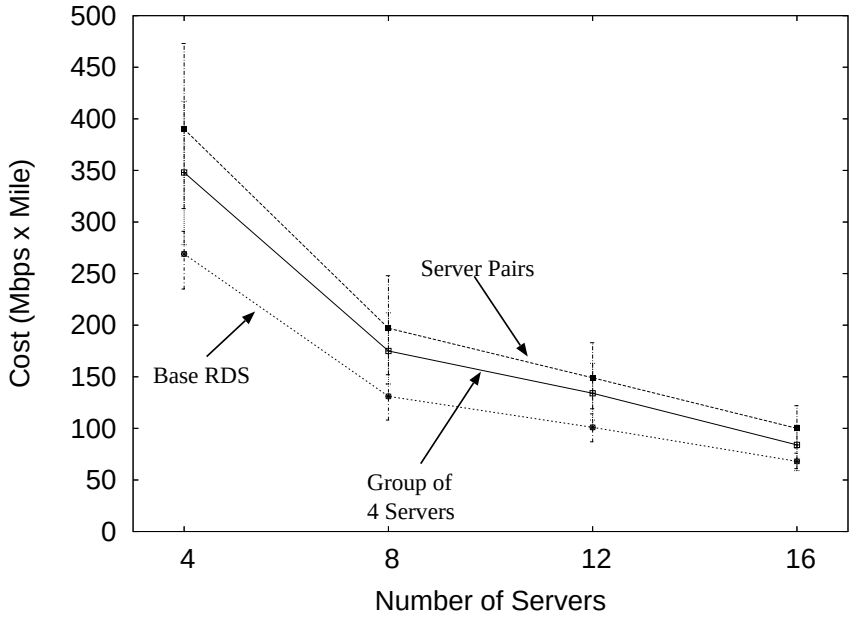


Figure 4.14: Simulation results of redirection subnetwork in torus networks.

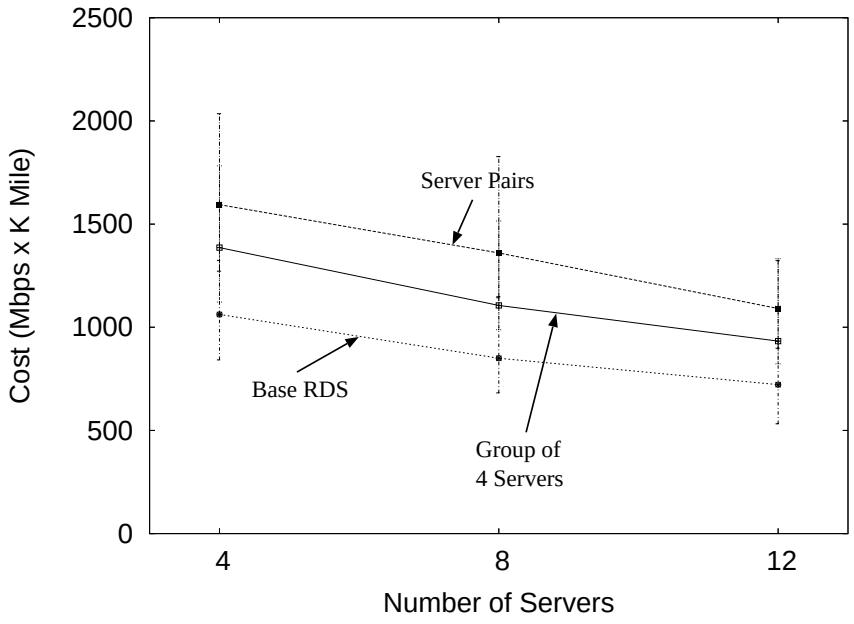


Figure 4.15: Simulation results of redirection subnetwork in the national network.

subnetworks, the curves labeled with “Server Pairs” are the costs of RDSs with redirection subnetworks for server pairs, and the curves labeled “Group of 4 Servers” are the cost of RDSs with redirection subnetworks for groups of four servers. As these plots show, when the number of servers increases, the costs of the RDSs drop, and the additional cost of redirection subnetworks is small, relative to the total communication cost of an RDS. They also show that although redirection subnetworks for server pairs can get optimal solution, the additional communication cost incurred by the additional reserved bandwidth is higher than those of the subnetworks for groups of four servers.

4.3 Summary

This chapter studies two issues in a multi-server RDS, the configuration problem for a multi-server RDS and the dynamic load redistribution in a multi-server RDS. The configuration of a multi-server RDS can be similarly formulated as a configuration problem for a single-server RDS. However, the key issue of optimal placement of servers makes the

multi-server RDS configuration problem complicated. A number of server placement algorithms are evaluated using simulation studies. Our simulation results show that a class of greedy server placement algorithms render the best solutions. The second part of this chapter studies the configuration algorithm of redirection subnetworks in a multi-server RDS such that temporary server overloading can be resolved by redirecting traffic to backup servers in a group with extra capacity. We presented solutions for configuration of traffic redirection subnetworks for server pairs and for groups of four servers.

Chapter 5

Source Traffic Regulation in Reserved Delivery Subnetworks

The previous chapters show that reserved delivery subnetworks (RDSs) can provide more consistent quality of service to users by reserving bandwidth on an aggregate basis. Besides the benefit of exclusive bandwidth access, there are other potentials to further improve end-to-end performance in an RDS because the end hosts can utilize the knowledge about the underlying network to achieve better performance than in the ordinary Internet. In this chapter, we propose a source traffic regulation technique to improve end-to-end performance in the environment of an RDS. The basic idea is to regulate the traffic from a server to sink end hosts such that bandwidth usage does not exceed the reserved link bandwidth and overloaded sinks do not affect other well behaved sinks. We propose a per-connection as well as an aggregated source traffic regulation algorithm for both single server and multi-server RDSs. We evaluate our algorithms with simulation studies in the *ns-2* network simulator, and outline implementations on end hosts, proxies, and as loadable modules on extensible routers.

The rest of the chapter is organized as follows: Section 5.1 briefly discusses the motivation for source traffic regulation in an RDS. Section 5.2 describes and analyzes the unbalanced bandwidth utilization problem in an RDS. In Section 5.3, we present both per-connection and aggregated source traffic regulation algorithms for single server RDSs. These algorithms are modified in Section 5.4 to suit an RDS with multiple servers. Details of our simulation studies are presented in Section 5.5 along with simulation results and analysis. We discuss algorithm implementation options on various platforms in Section 5.6. Section 5.7 summarizes this chapter.

5.1 Introduction

As we showed clearly in the preceding chapters, the exclusive bandwidth access in an RDS can improve end-to-end performance, especially during extreme situations when the network is under attack. In addition to the benefits of exclusive bandwidth access, we want to show that there is potential for more end-to-end performance improvements in an RDS. As Savage et al. [70] pointed out, the transport protocols in today's Internet are highly conservative, because they have to deal with the underlying network as a black box with unknown characteristics. On the other hand, if some information about the underlying network is available, end-to-end performance can be improved.

Unlike the ordinary Internet, servers in an RDS have information about the underlying network, including topology and reserved bandwidth. By utilizing this available information, we can make end-to-end performance improvements that are not possible in the ordinary Internet.

One possible end-to-end performance improvement in an RDS can be found in solutions to the unbalanced bandwidth utilization problem. In particular, in an RDS, when a sink node is under sustained overload, traffic flows to the overloaded sink consume more reserved bandwidth from the source than its fair share of the reservation. Because all traffic flows from the source node share the reserved bandwidth, flows to other sink nodes with lighter loads also get blocked. In addition, links close to the sink nodes with light traffic loads suffer from poor bandwidth utilization. This results in inefficient use of reserved bandwidth, reduced service quality, and sink starvation problems in an RDS. This situation can be mitigated by regulating the traffic flows at which the source node sends to the overloaded sink nodes so that these traffic flows do not interfere with flows to other unaffected sink nodes.

In this chapter, we present source traffic regulation techniques to improve end-to-end performance in the context of an RDS. These techniques may be extended to general overlay networks with reserved bandwidth. The idea of source traffic regulation was inspired by the distributed queueing packet scheduling algorithms in packet switches [58]. In a packet switch, when an output port is under sustained overload, the queues at the intermediate switch elements fill up, causing packet drops in the traffic flows to other sinks on other output ports that should otherwise be unaffected. To mitigate this problem in a packet

switch, the packets are scheduled in such a way that the traffic flows to all output ports are in accordance with the bandwidth available at the output ports, avoiding internal blocking.

Similar ideas can be applied to an RDS to improve the bandwidth utilization and thus end-to-end performance. There are two goals: first, we want to avoid excessive traffic flows on the RDS links; second, we want to keep the bandwidth utilization high so that the resources are not wasted. We present a basic per-flow source regulation algorithm and an aggregated regulation algorithm. More general regulation algorithms for multi-server RDSs are also outlined. We evaluated the regulation algorithms with simulation studies using the ns-2 network simulator [80]. In addition, we describe the implementation of the regulation algorithms on end hosts, performance enhancing proxies, and extensible routers.

5.2 Unbalanced Bandwidth Utilization Problem in RDSs

In an RDS, when a sink node is under sustained overload, and the demand is kept on a higher than average level, the traffic flow to such a sink will consume more bandwidth than it should on links in the path from the source. This bandwidth over-consumption will lead to an undesirable situation where the other sinks sharing some of the same upstream links to the source receive less bandwidth than their fair shares, even though the links close to these sinks have sufficient reserved bandwidth. As a result, the reserved bandwidth on these links is under utilized, and the users at the locations of these affected sinks will experience reduced quality of service.

Take a simple RDS in Figure 5.1 for example. There are three sinks (a, b, c), and the source is s. Assume each sink has a number of connections to s, and the traffic on each connection is a bursty on/off flow. Each traffic flow has an average burst (or "on") time of 0.5 seconds, an average idle (or "off") time of 9.5 seconds, and a burst rate of 20 Mbps. The average bandwidth of each flow is therefore 1 Mbps. We use 1K bytes packets. If a flow uses a TCP connection, the maximum window size is set to 250 packets, or 250KB. In the example in Figure 5.1, each sink has 70 connections on average, and thus an average demand of 70 Mbps.

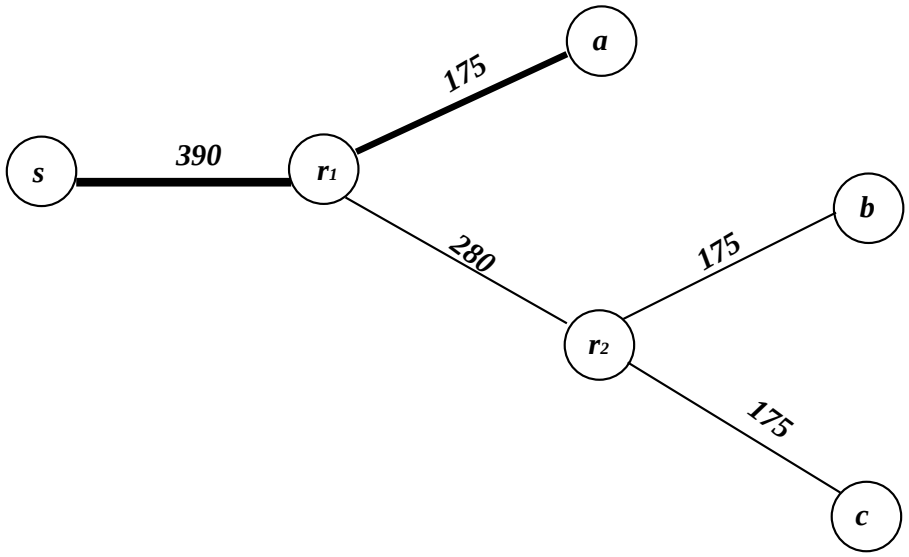


Figure 5.1: A simple single-server RDS example.

The reserved bandwidth to satisfy these sink demands are labeled along the links (in units of Mbps) in the figure. The reserved bandwidth on a link is set to accommodate the average traffic plus three times the standard deviation of the aggregate flow. For a link with n flows on average, the aggregate standard deviation is about $20\sqrt{npq}$, where $p = 0.95$ and $q = 0.05$. Thus, link $(r_1, a), (r_2, b)$ and (r_2, c) all get a reserved bandwidth of about 175 Mbps, link (r_1, r_2) gets about 280 Mbps, and link (s, r_1) gets about 390 Mbps. The transmission delays on the links $(s, r_1), (r_1, a), (r_1, r_2), (r_2, b), (r_2, c)$ are 25 ms, 25 ms, 50 ms, 25 ms, 75 ms, respectively. Thus, the round trip delay is 100 ms for a, 200 ms for b, and 300 ms for c. The queue length on a link is configured to be equal to the bandwidth-delay product for the largest round trip delay through that link.

Initially, each sink has 100 flows, 30 more than its average; after 30 seconds, sink c suddenly has 200 more flows, indicating an overloading condition. All traffic flows stop after 60 seconds. In theory, initially, all sinks should receive 100 Mbps on average. After 30 seconds, when the path to c becomes congested, a should get about 78 Mbps. b and c should get 78 Mbps and 234 Mbps on average on link (s, r_1), respectively. Therefore, traffic to b and c will congest link (r_1, r_2). b will get 70 Mbps on average, c will only get 210 Mbps on average on (r_1, r_2), and eventually an average of 175 Mbps after link (r_2, c).

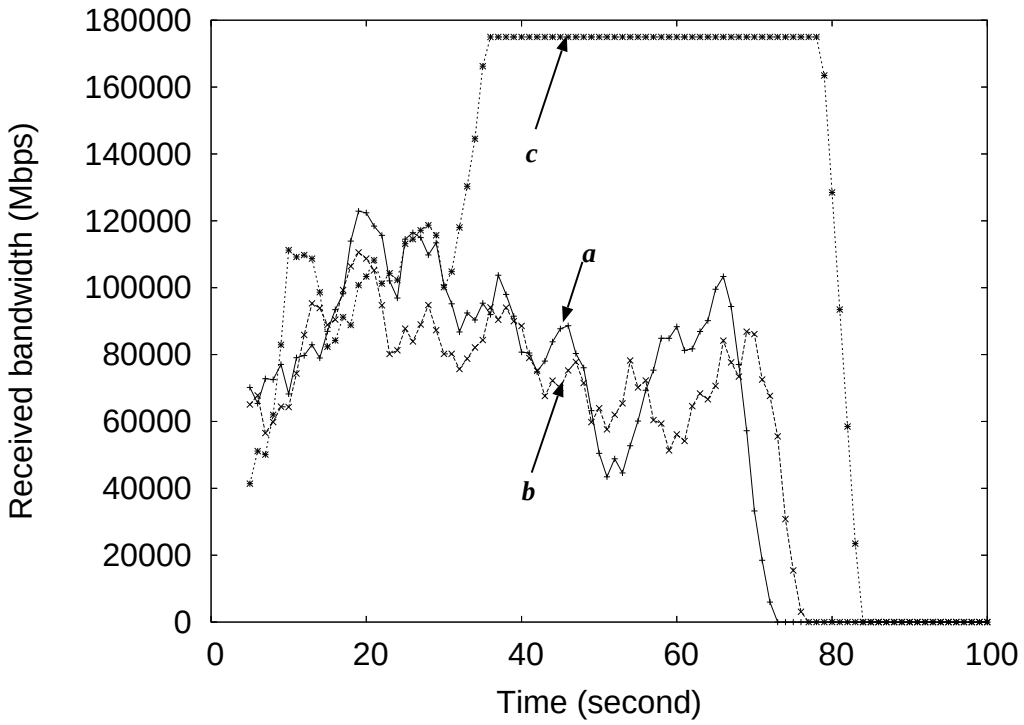


Figure 5.2: Unbalanced bandwidth utilization problem for bursty UDP traffic flows in the example network simulation. The received bandwidth measured shown in this plot is the moving average of the past five seconds.

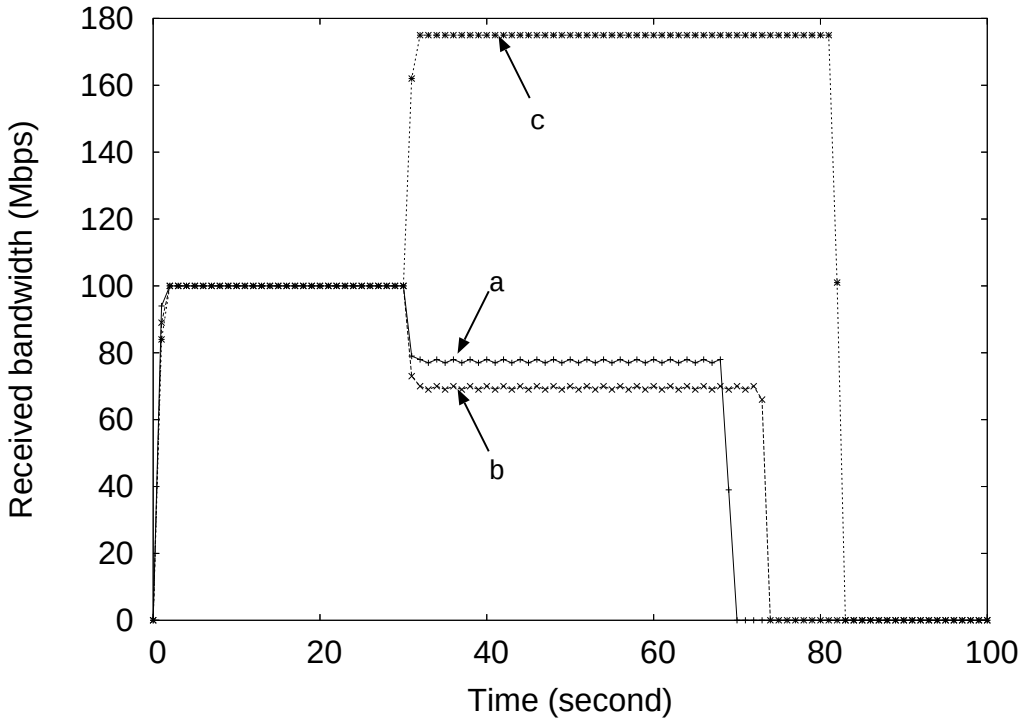


Figure 5.3: Unbalanced bandwidth utilization problem for CBR UDP traffic flows in the example network simulation.

Fig. 5.2 and Figure 5.3 plot the total sink perceived bandwidth of the three sinks in our simulation run in ns-2, demonstrating the unbalanced bandwidth utilization problem in this simple RDS. In the test results shown in Figure 5.2, we use the bursty flows as described. In comparison, in the test results shown in Figure 5.3, we use constant bit rate (CBR) traffic to do the same tests. Each CBR flow is 1 Mbps. In these tests, all connections are UDP connections. As we can see, all sinks can get an average bandwidth of 100 Mbps initially, but after 30 seconds, when c becomes overloaded, and gets bandwidth close to 175 Mbps, which is the maximum bandwidth allowed by the reservation on the last link, a and b only get about 75 Mbps each on average, down by 25%. The effects are more clear in Figure 5.3 when there is no burstiness. It actually confirms our estimated bandwidth. These tests show that a and b can not get their fair share of reserved bandwidth because c over-consumes the reserved bandwidth by overloading the path from the source, although there is plenty of bandwidth on the last link to a and b. Ideally, a and b should not be affected by the overload at c, and only c should be penalized for its excess traffic. However, c uses more than its fair share of the reserved bandwidth, reducing the quality of service to a and b.

In addition, these results show that the bursty traffic flows and CBR flows exhibit about the same average bandwidth, but the bursty traffic makes the results harder to interpret. We will use only CBR traffic in the rest of this chapter.

A similar problem with all TCP traffic flows is shown in Figure 5.4. We use the same CBR traffic sources on all TCP (Reno) connections. In this case, initially, all flows try to acquire the maximum available bandwidth, causing bandwidth surges. When congestion occurs, all sinks fall back to their fair share of bandwidth at about 100 Mbps. After 30 seconds, the received bandwidth for a and b drop to as low as about 65 Mbps and 72 Mbps respectively, while the additional new flows make c surge to about 175 Mbps. Flows to a and b eventually stabilize at about 100 Mbps available bandwidth after about 12 seconds. The received bandwidth at c drops to about 85 Mbps after congestion is encountered, and eventually stabilizes at 175 Mbps after 12 seconds.

Such an unbalanced bandwidth utilization problem on RDS links and reduced bandwidth on affected sinks is similar to the blocking problem in a packet switch with a sustained overload at an output port. By applying similar ideas to those used in distributed queueing algorithms for packet switches [58], these problems can also be resolved in an RDS, and

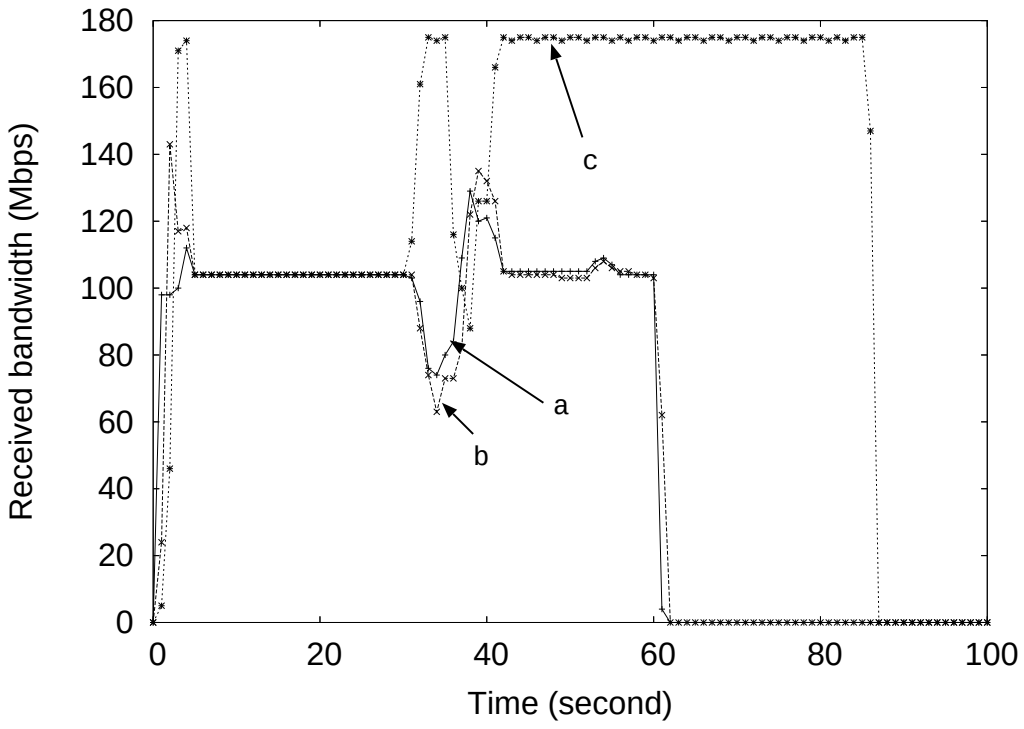


Figure 5.4: Unbalanced bandwidth utilization problem for CBR TCP traffic flows.

end-to-end performance improvements can be achieved. The essential idea of the solution is to regulate the transmission rates of the source node according to the data backlog to individual sinks in such a way that bandwidth utilization on links is balanced and maintained at high levels, and the quality of service at a sink is not be affected by other overloaded sinks.

5.3 Source Traffic Regulation in a Single Server RDS

5.3.1 Per-connection Traffic Regulation

We first present the source traffic regulation algorithm that regulates the source traffic on each individual connection. A transport protocol, such as TCP, keeps the status information about an individual connection on hosts at both ends of the connection (for example, the socket, the Internet PCB and the TCP PCB data structures in a BSD implementation). In order to implement source traffic regulation, we need to include additional information for each connection. Specifically, we need to keep track of how fast the receiver is consuming data from the source, how much data is waiting to be forwarded to the receiver at the source, and how much data is to be transmitted to the receiver at the sink. By receiver, we mean the user application that is consuming data at the sink end of the connection.

At the source, for each connection with a receiver x, we define the input data backlog $B_i(x)$ as the amount of data that the source has to send to the receiver x, the output data backlog $B_o(x)$ as the amount of data awaiting delivery to the receiver x at the sink, and the drain rate $r(x)$, the rate at which the receiver x consumes data from the sink. The drain rate and output data backlog for a connection can be constantly measured at the sink that monitors the flows to end hosts. This information is then fed back to the source on a regular basis. A *virtual sink queue* (VSQ) is maintained for each connection at the source to keep track of the input data backlog to the receiver at the other end of the connection. The VSQ is the counterpart of a *virtual output queue* (VOQ) used on the input port of a router to keep track of data to be forwarded to a specific output port [58].

The source regulates the transmission rate on each active connection based on its input data backlog, its output data backlog and the receiver drain rate, subject to the reserved

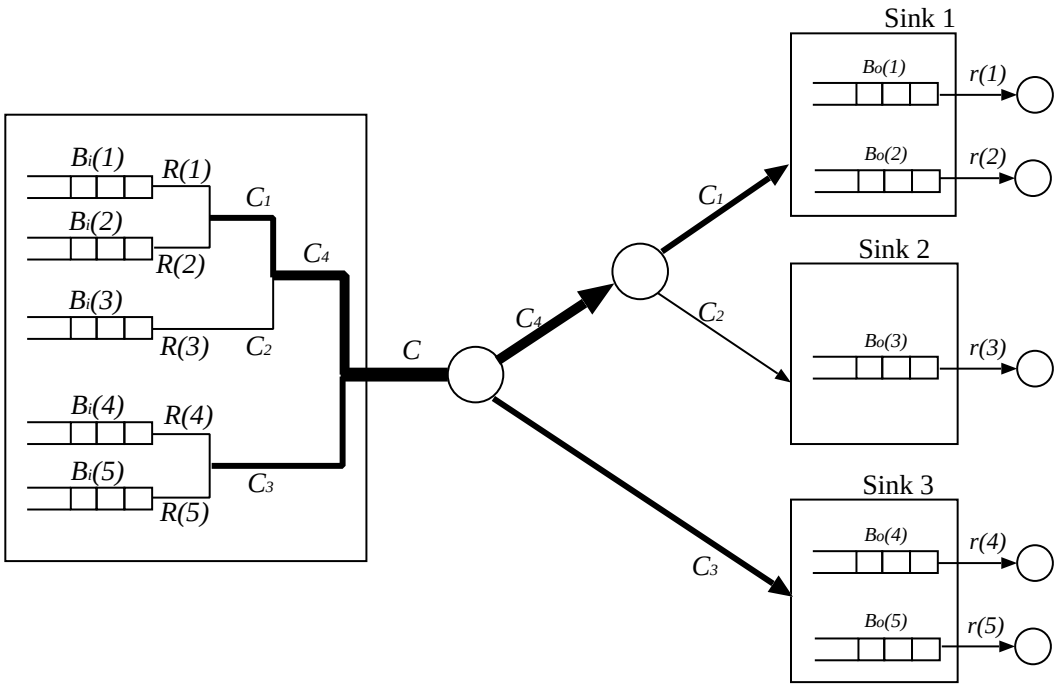


Figure 5.5: Per-connection traffic flow regulation.

bandwidth constraints. To enforce the reserved bandwidth constraint on all end-to-end paths, the underlying RDS topology along with the reserved bandwidth on all links within the RDS are kept at the source. The source keeps checking the total transmission rates on all links against the reserved bandwidth to ensure the transmission rates do not exceed the reserved bandwidth.

Figure 5.5 shows an example per-connection traffic flow regulation scenario. In this example, there are three sinks, and five end-to-end connections to five receivers (1 through 5) at different sinks. The total reserved bandwidth allocated for the source is C, and the reserved bandwidth on the other links are C_1, C_2, C_3, and C_4, as labeled on the diagram. Assume the source assigns a transmission rate $R(x)$ to each receiver x. These transmission rates are

subject to the reserved bandwidth constraints:

$$\begin{aligned} R(3) &\leq C_2 \\ R(1)+R(2) &\leq C_1 \\ R(4)+R(5) &\leq C_3 \\ R(1)+R(2)+R(3) &\leq C_4 \\ R(1)+R(2)+R(3)+R(4)+R(5) &\leq C \end{aligned}$$

If all the constraints above are enforced, none of the bandwidth reservation is exceeded.

The source traffic regulation algorithm determines the transmission rate on each active connection, based on the data backlogs on both server and sink sides as well the draining rates on individual connections. It always tries to allow the active connections with the shortest time to drain their output data backlogs to transmit first because these connections are not overloaded, and should not affected by other overloaded connections. In addition, when all the output backlogs are all relatively small, then the active connections with smaller input data backlogs will transmit first to avoid overloaded connections. Therefore, the active connections that are least likely overloaded are given the priority to transmit first, thus limiting the impacts of overloaded connections.

If we denote T as the scheduling interval (the interval between two updates of the output data backlog and drain rate information from the sinks), the source can regulate traffic at each interval T using the following algorithm:

For each active connection with receiver x,
  Output draining time $t(x) = B_0(x)/r(x)$
For all active connections with draining time $< T$,
  Sort these connections by input backlog B_is in an increasing order
For each active connection with receiver x (in sorted order)
  Transmit at $R(x) = \min\{B_i(x)/T, R_a(x)\}$,
    where $R_a(x)$ is the min. available bandwidth on the end-to-end path to x
If $R_a(x) > 0$
  For all active connections with draining time $\geq T$,
    Sort these connections by receiver draining time $t(x)$ in an increasing order

For each active connection with receiver x (in sorted order)
Transmit at $R(x) = min\{B_i(x)/T, R_a\}$,
end

This algorithm attempts to clear the data backlogs that can be drained quickly first by sorting the connections by the estimated drain time in an increasing order. For a connection with the shortest estimated drain time, compare its input data backlog with the traffic allowed by the remaining reserved bandwidth along the path from the source to the sink. The remaining reserved bandwidth is the difference between the original reserved bandwidth and the bandwidth already used by the connections with shorter estimated drain time. If the input data backlog is greater than the amount of data allowed by the remaining reserved bandwidth on the path, then send data to use up the remaining reserved bandwidth. If the remaining bandwidth allows for more data to be transmitted than the input data backlog, then let the input data backlog be cleaned out before the next update interval, and update the remaining reserved bandwidth along the path accordingly. If there is still remaining reserved bandwidth after cleaning out an input data backlog, then try to allocate remaining reserved bandwidth to the other connections to the same sink that have longer estimated drain time, starting with the connection with the shortest estimated drain time. Repeat the procedure until all input data backlogs are cleared, or all reserved bandwidth is consumed.

5.3.2 Aggregated Traffic Regulation

Although the additional per-connection information does not incur excessive amounts of memory, the per-connection traffic regulation may require excessive computation when there are many active flows. Therefore, we present an aggregated source traffic regulation algorithm that reduces the overhead and maintains the efficiency and effectiveness for large number of connections. In particular, a VSQ in the aggregate regulation algorithm is maintained for each *sink* instead of each connection, and each sink only maintains an aggregated output data backlog for *all* connections that pass through, instead of one backlog for each connection. Similarly, the sink measures the aggregate estimated drain rate for *all* connections. All connections to the same sink share the same input backlog, output backlog and drain rate. Besides these differences of data backlogs, the traffic regulation algorithm works the same way. Figure 5.6 shows a simplified diagram of aggregate traffic

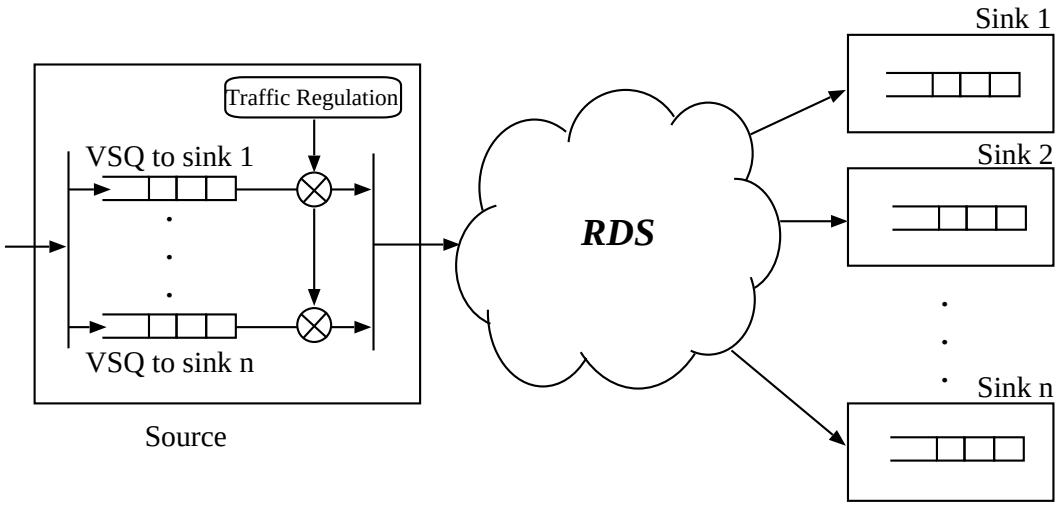


Figure 5.6: Aggregated traffic flow regulation.

flow regulation in contrast to the per-connection traffic flow regulation algorithm.

The aggregate traffic regulation algorithm works in the similar way as in the per-connection regulation algorithm. We first sort the sinks by their estimated drain time in increasing order. Then, we pick the VSQ with the shortest estimated drain time, and check if it is sufficient to drain the aggregate input data backlog. If not, transmit as much as allowed by the reserved bandwidth. Otherwise, clear out the input data backlog to that sink, and use the remaining bandwidth for the other sinks with the shortest draining time first.

Besides the per-connection and aggregate regulation, another intermediate option is to assign active flows to one of m queues per sink, using a hash function. The semi-aggregate traffic regulation algorithm treats each of the m queues individually as if each queue has one individual active connection. This intermediate solution has less overhead than the per-connection regulation, and has more precise regulation of individual flows than the total aggregate regulation.

It should be noted that source traffic regulation is coarse grained because rates are determined based on past information that lags by at least one round trip delay. The higher the frequency of the control messages, the more accurate is the regulation, but with a tradeoff of higher overhead.

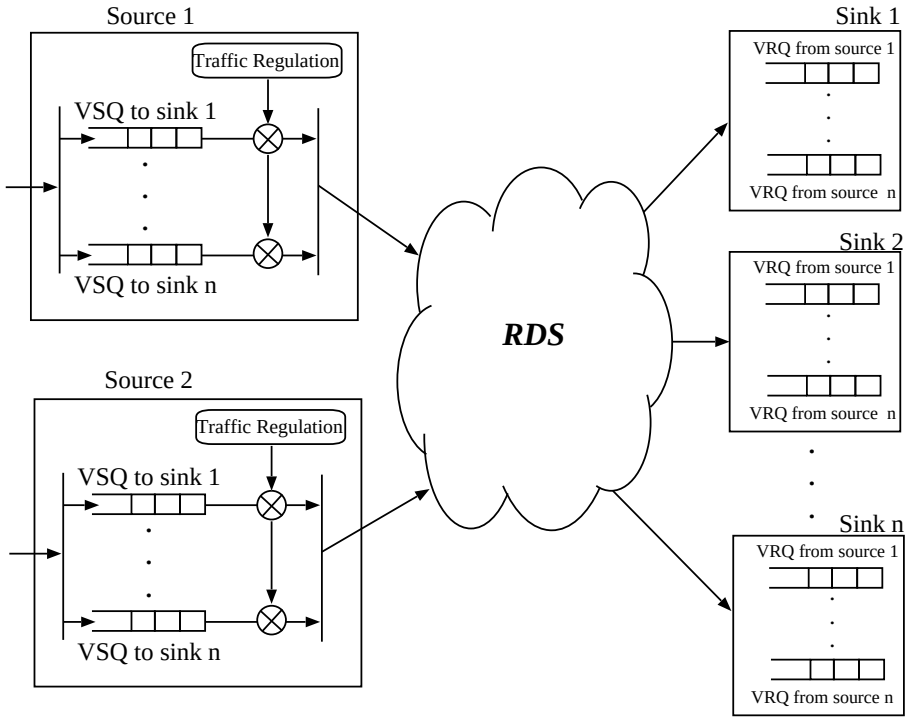
5.4 Source Traffic Regulation in a Multi-server RDS

In an RDS with multiple sources, traffic flows from different sources could compete for reserved link bandwidth. It is clear that as the number of source nodes increases, more nodes are likely to be affected by overloaded sink nodes. Note, if the reserved bandwidth is exclusively for an individual source node, the problem caused by an overloaded sink is limited to flows from the same source. In this case, the single traffic flow regulation is sufficient. However, in a multi-server RDS with shared reserved bandwidth among different sources, more complicated traffic flow regulation algorithms should be used to account for the additional sources. The major challenge is to coordinate the transmission rates from different sources to the same sink without causing unbalanced bandwidth utilization problems.

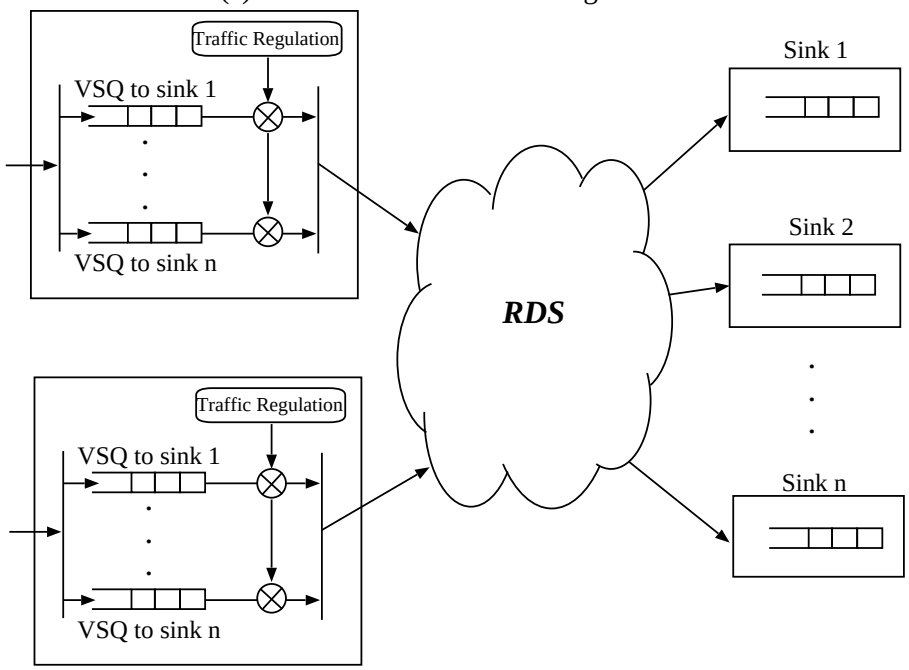
As shown in Figure 5.7(a), when there is more than one data source sending to a set of sinks, one source has to consider other sources to the same sink when determining its transmission rates. In particular, a source with a larger input backlog to a sink should get higher transmission rate than the ones with smaller input backlogs. The traffic regulation should also consider the output backlog and drain rates as for the single source case. In addition, the reserved bandwidth constraints along all paths from a source to a sink should be observed.

We need to modify our previous definitions to handle multiple source nodes. One tricky thing about multi-source traffic regulation is how a source can get information about input backlogs of other peer sources. One solution is to let the sink collect source data backlog information from all sources, and feeds it back to all the sources. Thus, a source not only receives data backlog information from its sinks, but also sends out its input backlog to all its sinks. A sink gathers this information and sends back to all sources. Another option is to fully connect all sources and regularly exchange input data backlog information among all sources. However, this requires another subnetwork among the peer servers, and does not take advantage of the existing RDS infrastructure.

In this basic multi-source regulation algorithm, because a source needs to multicast its input backlog to all its sinks or other sources, the control message overhead doubles, making it less effective and less efficient when there is a large number of flows. In this case, an aggregate multi-source traffic regulation algorithm is more favorable as it reduces the



(a) Basic multi-source traffic regulation.



(b) Aggregated multi-source traffic regulation.

Figure 5.7: Multi-source traffic regulation.

control overhead. In the aggregate regulation algorithm, if we use the sinks to "bounce" back the data backlog of the peer sources, a source sends its aggregate backlogs to all sinks. A sink sends back its own aggregate backlog and aggregate drain rate along with the source data backlog gathered from all the sources it connects back to these sources. If we use a dedicated subnetwork among sources to exchanges data backlog, the source aggregate the input data backlog information, and send it to the other sources. Sources can use this aggregate information to limit their use of shared resources to prevent overuse. Figure 5.7(b) shows an RDS with aggregate multiple traffic flow regulation. As in the single source case, a single queue is maintained for all flows from a sink node to all end hosts connected, and only aggregate drain rates are fed back to the source nodes.

5.5 Simulation Studies and Analysis

The evaluation of traffic regulation in an RDS was conducted through simulation studies using the network simulator (ns-2) [80]. New regulation classes are introduced in ns-2 to implement the source traffic regulation algorithms. We simulate the web traffic by assigning a traffic generator with CBR traffic. We choose CBR traffic because it shows the average bandwidth more clearly. Similar results can be obtained with bursty traffic flows, but the CBR results make the effects of source traffic regulation more apparent. In our simulations, each CBR flow has a rate of 1 Mbps. The link transmission latency of the links $(s, r_1), (r_1, a), (r_1, r_2), (r_2, b), (r_2, c)$ are 25 ms, 25 ms, 50 ms, 25 ms, 75 ms, respectively. Therefore, the RTT to sinks a, b, c are 100 ms, 200 ms, and 300 ms, respectively. All intermediate routers are using drop-tail queues, and the queue length is equal to the product of reserved bandwidth and maximum round trip delay. The regulation interval is set to the maximum round trip delay 300 ms.

5.5.1 Simulations

In our ns-2 simulations, we first implement the infrastructure of the basic RDS that enforces the bandwidth reservation on links in the subnetwork. Then, we implement the traffic regulation algorithm as a special regulator application that regulates the traffic generator at each traffic source. The traffic source is modified to allow the regulator to control the output.

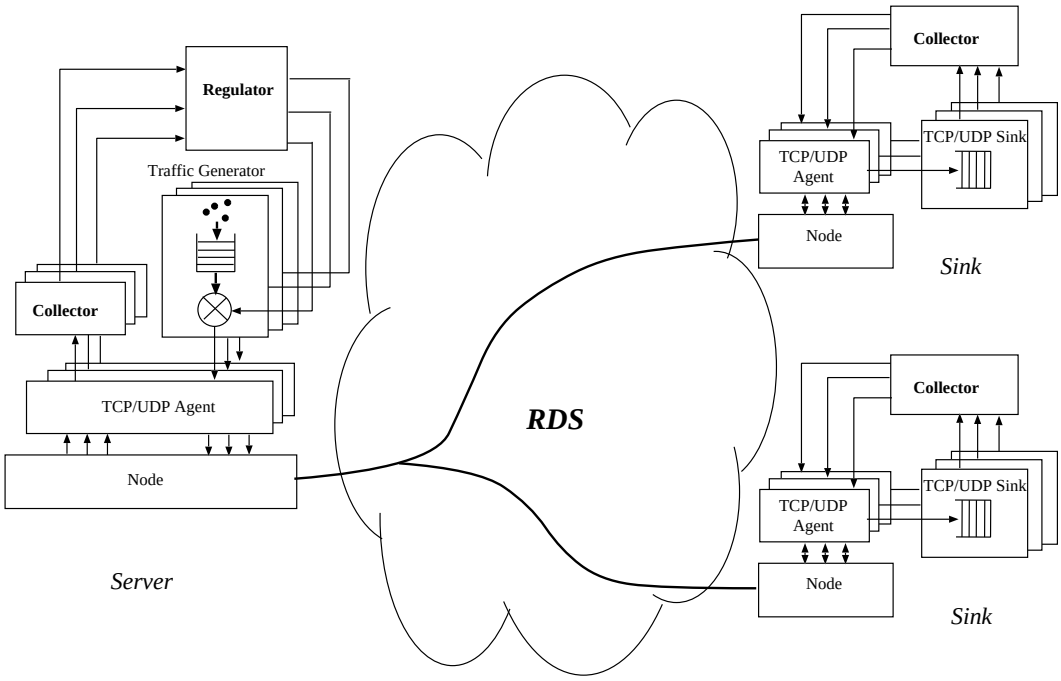


Figure 5.8: Source traffic regulation implementation in *ns-2*.

A collector is introduced as an application to gather the sink side data backlog information, and feed them back to the regulator. The implementation details are illustrated in Figure 5.8. As we can see, the traffic generator includes a new regulation control after the original packet queue. This regulation control is regularly updated after each regulation interval by the regulator to control the rate of a traffic source based on the regulation algorithm implemented in the regulator. The regulator has the data backlog information at both ends of each connection, as well as the topology and reserved bandwidth of the underlying RDS. At each regulation interval, the regulator updates a counter in the regulation control, and the attached TCP or UDP agent is only allowed to transmit this amount of data before the next regulation interval. The collector sends back the sink data backlog information also once every regulation interval.

5.5.2 Experimental Results and Analysis

We use the same simple network as in Figure 5.1 for our simulations for single-server RDS regulation. In particular, each of the three sinks initially has 100 connections, and each

connection with a CBR traffic generator at the source. After 30 seconds, sink c adds an additional 200 CBR flows, each with the same traffic generator. The total sink perceived bandwidth is measured and plotted with and without the source traffic regulation. We simulate both cases of all TCP flows as well as all UDP flows.

Figure 5.3 and Figure 5.4 in Section 5.2 show the results for all UDP and all TCP flows with no source traffic regulation. In the all-UDP flow simulation results in Figure 5.3, the total bandwidth to the overloaded sink c is limited (175 Mbps) by its access link reservation. Sinks a and b should not be affected, but their received bandwidth drops below their fair share of bandwidth (100 Mbps), resulting in reduced service quality. In the all-TCP flow simulation results in Figure 5.4, although all sinks get the optimal bandwidth allocation eventually, it takes more than 10 seconds to adapt to the optimal bandwidth, during which the sink bandwidth drops to below the fair share of reserved bandwidth. When we enable source traffic regulation, we expect a and b will not affected by c and get 100 Mbps along their path from s, and c will get 175 Mbps along its path from s.

Per-connection Regulation in a Single-server RDS

Figure 5.9 shows the effects of the source traffic regulation for the all UDP flow case. Because the source now determines the transmission rates based on the data backlogs on source and sink sides, the sinks with normal loads (a and b) are not affected by the overloaded sink c, and can still get their fair share of reserved bandwidth when the path from s to c becomes congested. Thus, even after 30 seconds, a and b still maintain about 100 Mbps bandwidth, while c is still limited to 175 Mbps by its access link. After 60 seconds, when traffic stops in all connections, input data backlogs to a and b are quickly cleared out, while c takes about 22 seconds to clear out its backlog.

Figure 5.10 shows the effects of the source traffic regulation for the all-TCP flow case. At the beginning, all flows try to find the maximum rates they are allowed during the TCP slow start phase, causing the surges. They soon get to their fair share of the reserved bandwidth of 100 Mbps. When c becomes overloaded after 30 seconds, the TCP congestion control mechanism causes all flows to reduce their transmission rates. Because we now have source traffic regulation, it takes less time (3 to 4 seconds, as compared to 12 seconds in Figure 5.4

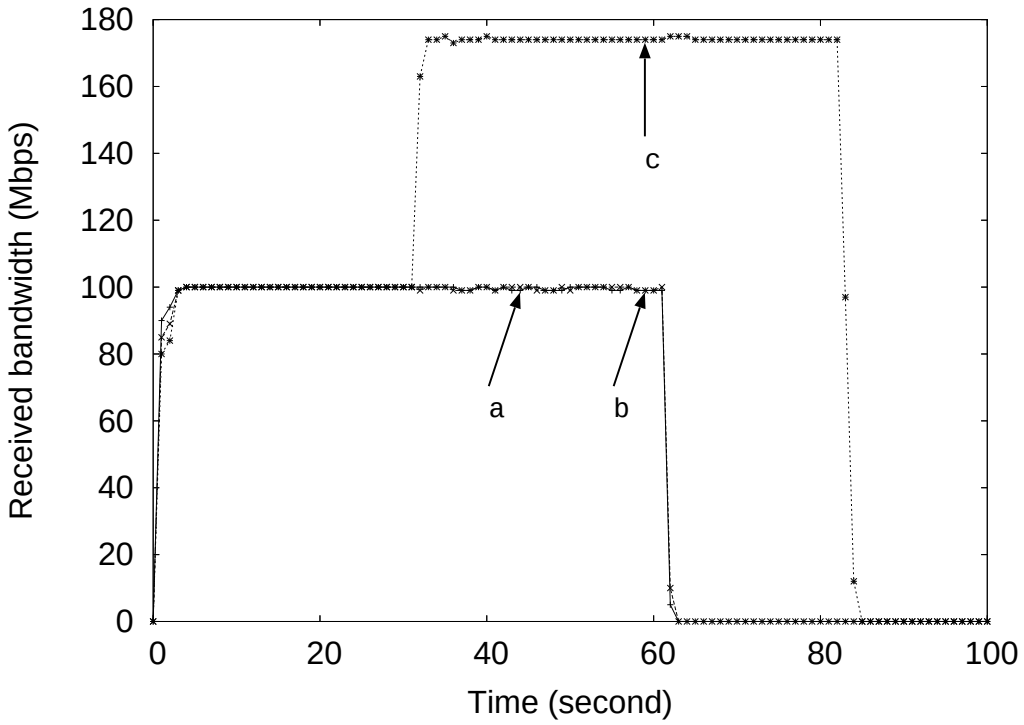


Figure 5.9: Simulation with per-connection source traffic regulation for all UDP traffic flows.

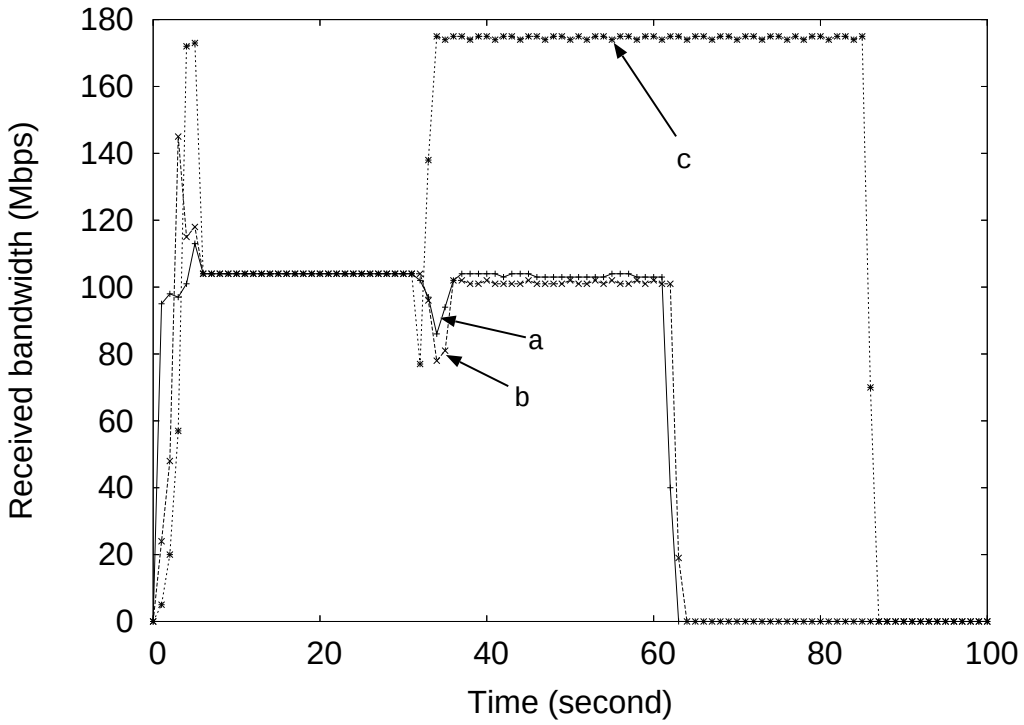


Figure 5.10: Simulation with per-connection source traffic regulation for all TCP traffic flows.

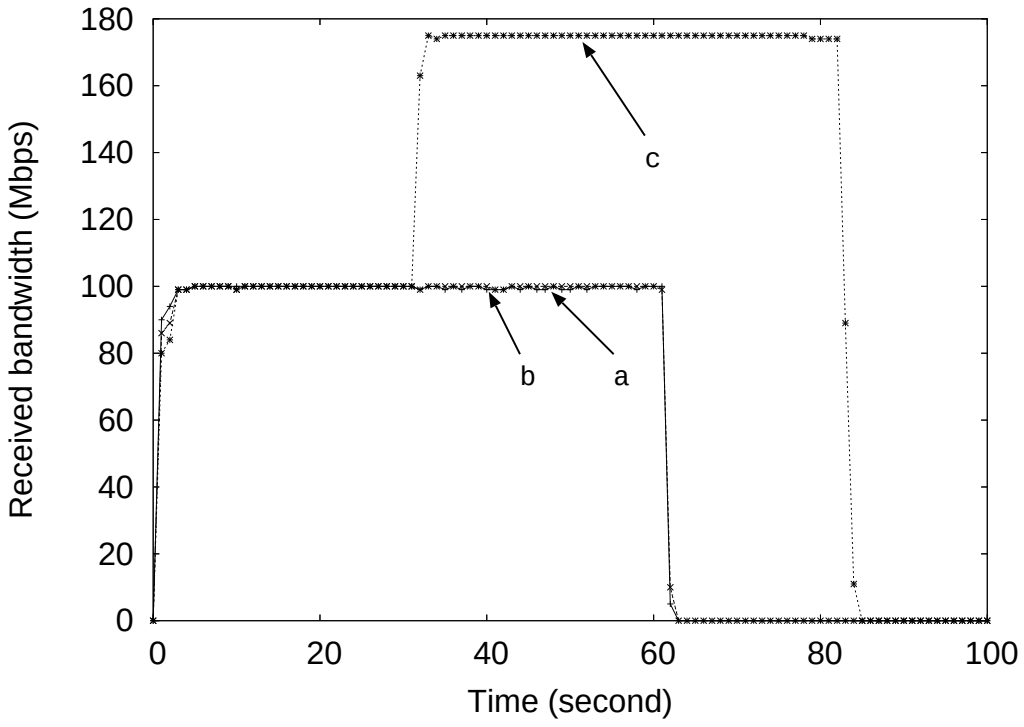


Figure 5.11: Simulation with aggregated source traffic regulation for UDP flows.

without source traffic regulation) for all the sinks to return to their fair shares of reserved bandwidth.

Aggregated Regulation in a Single-server RDS

Our simulations with aggregate source traffic regulation show that normally the aggregated source traffic regulation has the same effects on the total sink bandwidth results as the per-connection regulation, especially for all-UDP flows, as shown in Figure 5.11 (all-UDP flows) and Figure 5.12 (all-TCP flows). One major difference for all-TCP flows is that the received bandwidth of c drops to a lower level after c becomes overloaded, and that it takes a and b about 4 more seconds to return to 100 Mbps, and the bandwidth fluctuation is much smaller than the the case with no traffic regulation. In addition, the received bandwidth of c reduces gradually instead of sharply as in Figure 5.10.

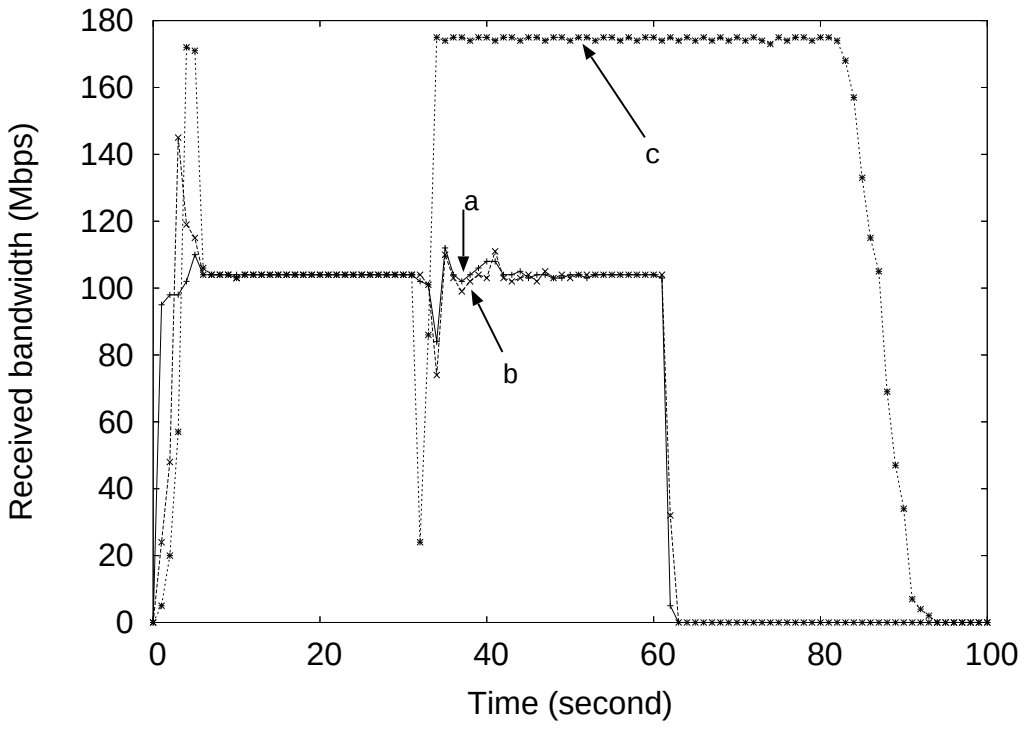


Figure 5.12: Simulation with aggregated source traffic regulation for TCP flows.

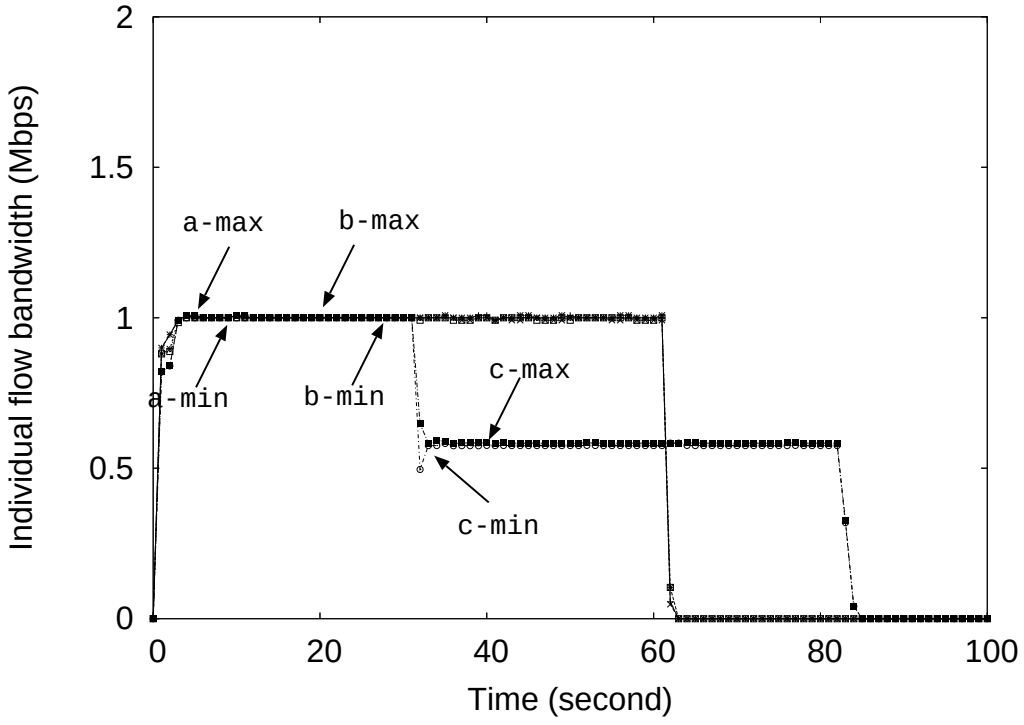


Figure 5.13: Maximum and minimum bandwidth in individual UDP flows to each sink with per-connection regulation.

These differences are caused by the different ways these two algorithms regulate the flow transmission rates. The per-connection regulation algorithm reduces the transmission rates on all connection when the end-to-end path gets congested, so the sources of all connections transmit at about the same lower rate. Thus, the overall fluctuation caused by the TCP congestion control mechanism on individual flows is smaller. In contrast, when congestion occurs, the aggregate regulation algorithm still allows an individual source to send as fast as it can as long as the total transmission rate is reduced. So, it essentially reduces the number of flows transmitting at high rates, and increases the number of flows that are "on hold" from transmission. Therefore, the overall fluctuation caused by TCP is higher.

The differences of the two regulation algorithms are more clearly depicted in Figure 5.13 through Figure 5.16. In these figures, we show the maximum and minimum individual *active* flow bandwidth among all flows to a sink measured every second with both regulation algorithms for both all-UDP and all-TCP flows. For example, the maximum individual flow bandwidth to sink a is labeled as "a-max", and the minimum individual flow bandwidth to

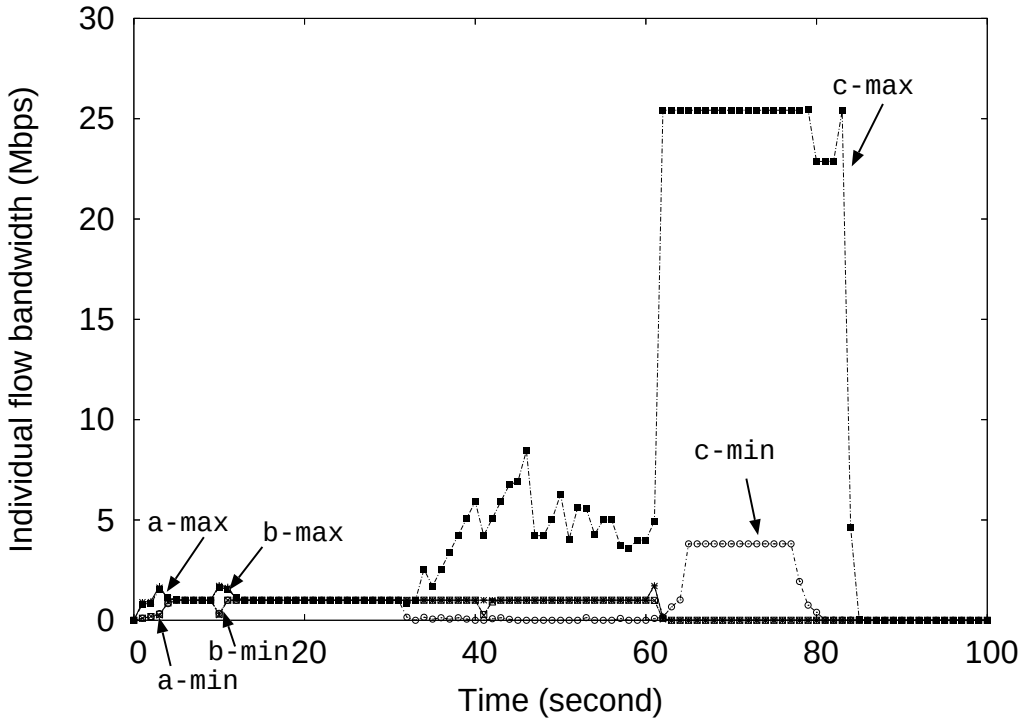


Figure 5.14: Maximum and minimum bandwidth in individual UDP flows to each sink with aggregate regulation.

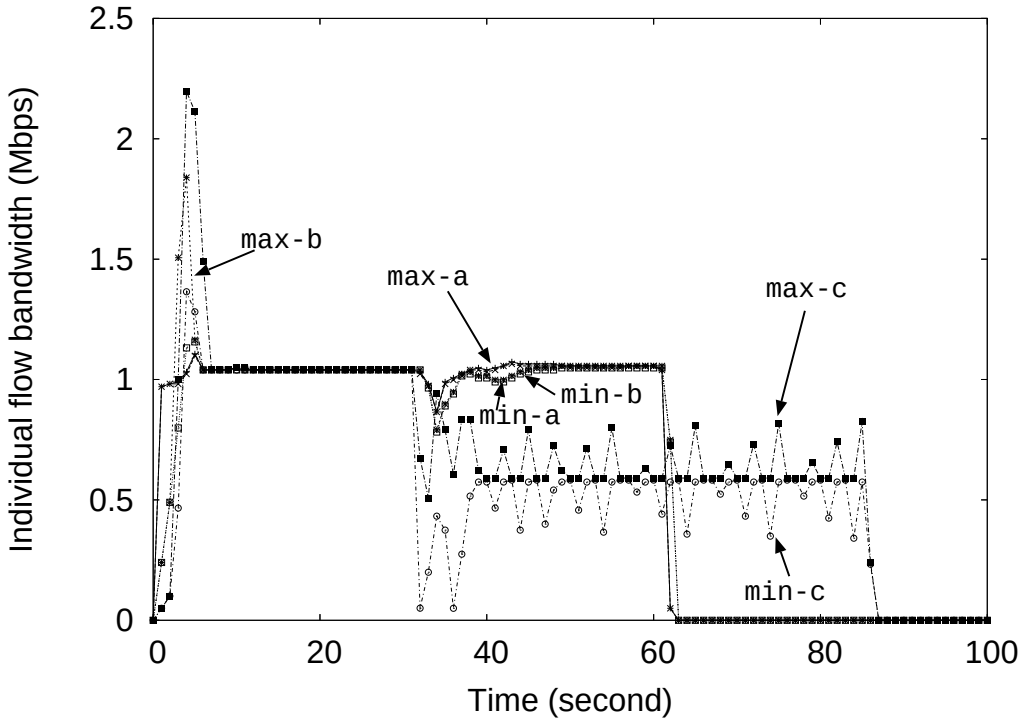


Figure 5.15: Maximum and minimum bandwidth in individual TCP flows to each sink with per-connection regulation.

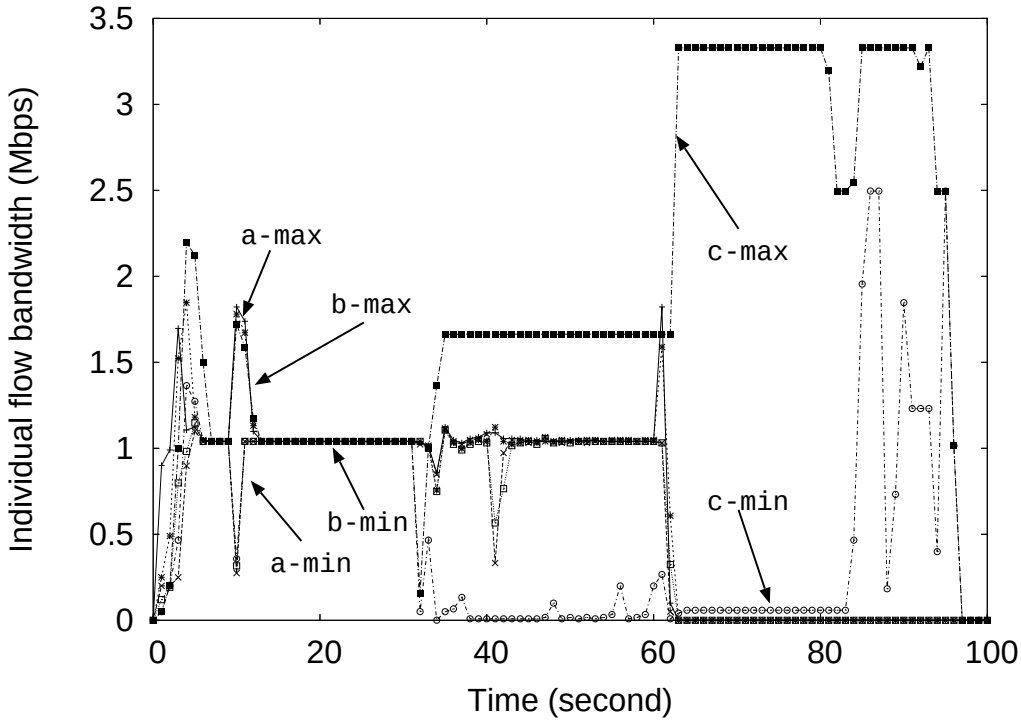


Figure 5.16: Maximum and minimum bandwidth in individual TCP flows to each sink with aggregate regulation.

a is labeled as "a-min". Figure 5.13 and Figure 5.15 use per-connection regulation, and Figure 5.14 and Figure 5.16 use aggregate regulation. Figure 5.13 and Figure 5.14 are for all-UDP flows, and Figure 5.15 and Figure 5.16 are for all-TCP flows.

Figure 5.13 shows that flows on connections to all sinks are almost the same. Before c becomes overloaded, the maximum and minimum individual flow bandwidth is about $1Mbps$, which means all flows are transmitting at the the maximum allowable rates. After c becomes overloaded after 30 seconds, the maximum and minimum individual flow bandwidth drops to about 0.6 Mbps. This is because when c becomes overloaded, all flows have 60% (roughly 175 Mbps out of 300 Mbps) of time transmitting each second, and all flows have equal opportunities to transmit.

In contrast, Figure 5.14 shows that the maximum and minimum individual bandwidth among flows to a and b is still about 1 Mbps, but the maximum individual flow bandwidth to c is much higher after c becomes overloaded. When a and b are still transmitting before the 62nd seconds, the individual flow maximum bandwidth goes up to 8.4 Mbps and has an average of about 5 Mbps, and the individual flow minimum bandwidth drops to close to 0. This indicates that some flows transmit more often than the others when using aggregate regulation because it does not attempt to regulate individual flows to avoid overloading each receiver. So, some flows are allowed to transmit as much as they can as long as the aggregate backlog at the sink is not too high and the aggregate received bandwidth is below its fair share of reserved bandwidth. Note that we only measure the active flows, which is less than 300. So the total bandwidth to c does not exceed the reserved bandwidth of 175 Mbps on the bottleneck link. After a and b stop their transmission at about 62th second, the individual flow maximum bandwidth jumps up to and stays at about 25 Mbps. The individual flow minimum bandwidth also increases to about 3.8 Mbps. This is because after a and b finish transmission, more reserved bandwidth is available to c, and c can transmit more data on the individual flows. So, the individual flow maximum bandwidth increases. in addition, because some flows get more chances to transmit earlier, some of these flows already cleared out their input backlog, and have no data to transmit. So, the total number of flows decreases, more flows with large accumulated backlog begin to transmit, and the individual flow minimum bandwidth increases.

Figure 5.15 shows that the individual flow maximum and minimum bandwidth to a and b is 1 Mbps. Among flows to c, the individual flow maximum and minimum bandwidth is

1 Mbps before C becomes overloaded at the 30th second; after c becomes overloaded, the individual flow maximum bandwidth drops to an fluctuating number no less than 0.6 Mbps, and the individual flow minimum bandwidth drops to an fluctuating number no greater than 0.6 Mbps. The fluctuation of the individual maximum and minimum bandwidth roughly complements each other. This indicates that there are active flows on all connection when c is overloaded. It also shows that not all flows get the same transmission rate every second; some flows get more data to transmit while other flows get about same less amount of data to transmit. These effects are caused the regulation of individual flows.

Figure 5.16 shows that although there are several "spikes" and "dips", the individual flow maximum and minimum bandwidth to a and b is about 1 Mbps. Among flows to c, the individual flow maximum and minimum bandwidth is 1 Mbps before it becomes overloaded at the 30th second. After c becomes overloaded and before a and b stop receiving data around the 62nd second, the individual flow maximum bandwidth increases to about 1.66 Mbps, and the minimum individual flow bandwidth drops to near 0. This shows that some flows are allowed to transmit more data while other flows only allowed to transmit very little data by the aggregate regulation algorithm. The difference is not as large as in the all-UDP flows though. After a and b clear out their data backlog and stop receiving data from the source, the maximum and minimum individual flow bandwidth both increases. The maximum individual flow bandwidth jumps to up to 3.3 Mbps and the minimum individual bandwidth jumps up to about 2.5 Mbps after 83 seconds. This is because more reserved bandwidth is available to c, and more flows can transmit more data to clear up their data backlog. As the data backlogs are cleared out on more connections, even more flows can transmit at a higher rate.

TCP Fairness

When two overloaded sinks have different round trip transmission delays to the server, the TCP congestion control mechanism will favor the sink with the shorter RTT. This bandwidth unfairness to different sinks is shown in Figure 5.17. This figure shows a simulation run on a simple network similar to the network we used previously, except that link (r_2, b) has a transmission delay of 100 ms in this network. This makes RTT from s to b 300 ms. Specifically, in this simulation, all three sinks start with 100 TCP flows, each with a CBR source with 1 Mbps bandwidth. After 30 seconds, a and b have an additional 50 flows, and

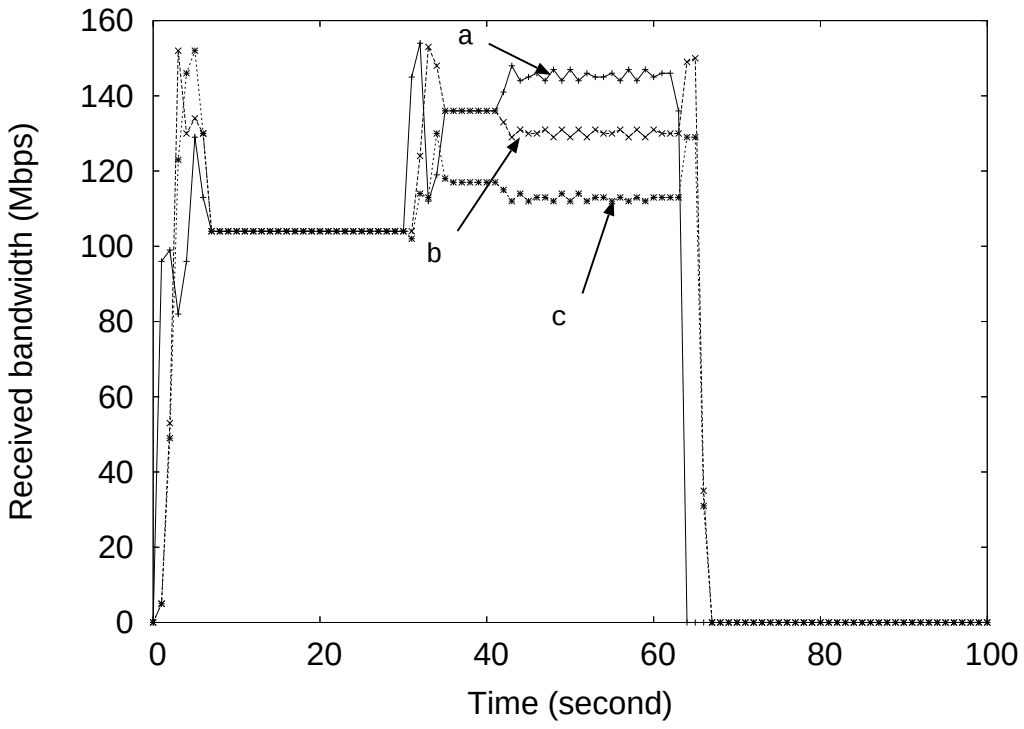


Figure 5.17: Bandwidth unfairness to congested sinks with different round trip delays.

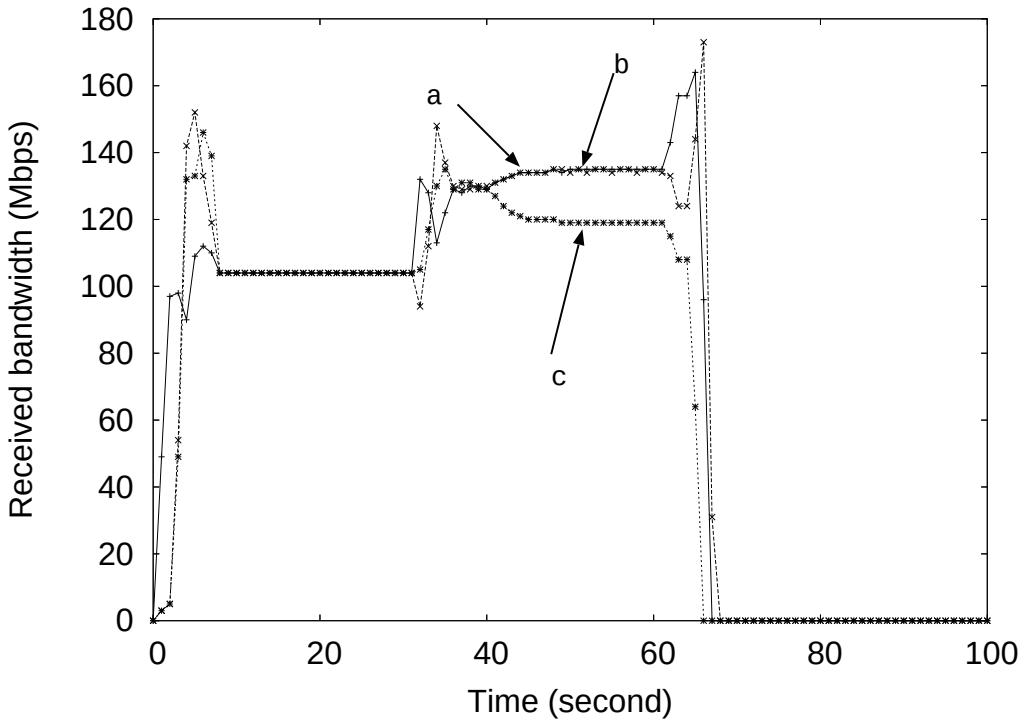


Figure 5.18: Improved TCP fairness with source traffic regulation.

c has an additional 30 flows. Thus, the (s, r_1) becomes the only congested link, and a and b are competing for the bandwidth on the congested link. Figure 5.17 shows that a receives about 145 Mbps of bandwidth and b only gets about 130 Mbps, although they both have the same total traffic demands.

Figure 5.18 shows the results of the total received sink bandwidth when we enable source traffic regulation. We only show the result of the per-connection regulation algorithm; the aggregated regulation produces similar results. Figure 5.18 clearly shows that when traffic regulation is enabled, all sinks adjust to their fair share of reserved bandwidth, independent of their RTT to the server. In particular, the competing overloaded sinks a and b both get roughly 135 Mbps even though b has a RTT that is three times that of a. These results show that source traffic regulation provides better fairness than TCP congestion control alone.

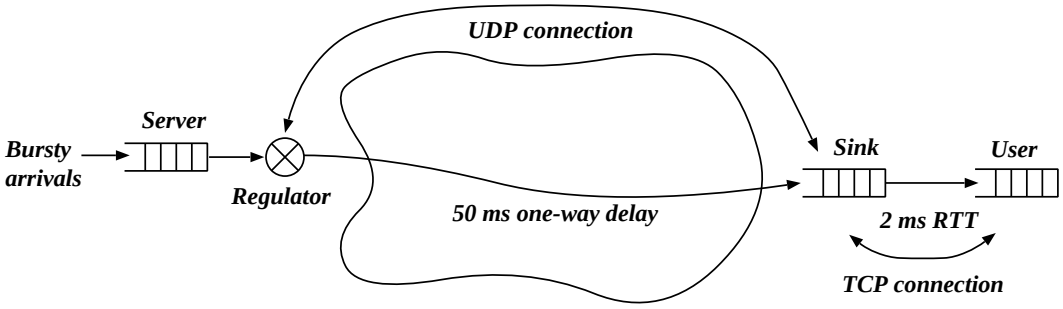


Figure 5.19: End-to-end burst delivery time simulation setup.

An Example Application

In addition to the bandwidth related simulations, we have also simulated an example application of source traffic regulation to improve end-to-end performance for delivery of large amounts of data across a wide area network, such as stream video image delivery.

Figure 5.19 shows the network setup for our simple simulation. There is a server that constantly has bursty traffic flows to the sink. The connection from the server to the sink is a UDP connection that has a one-way transmission delay of 50 ms. Overloading is prevented on this connection through source traffic regulation between the server side regulator and the sink. The connection from the sink to the user is a TCP connection with a round trip delay of 2 ms. The bursty traffic that arrives at the server is transmitted to the sink through the UDP connection, subject to the source traffic regulation by the regulator.

In our simulations, we measure the average and standard deviation of the end-to-end burst delivery time, which is the time between the moment when the first byte leaves the server and the moment when the last byte is received by the user. In comparison, we also measure the same data for regular end-to-end TCP connections from the server to the user. We use a packet size of 1500 bytes, and an average burst size of 100 packets. The average burst arrival rate is 100 per second. The burst transmission rate is 1Mbps. The connection between the server and the sink has a reserved bandwidth of 150 Mbps. It uses a Drop Tail queue, and the queue is set according to the delay bandwidth product.

Figure 5.20 shows the average burst delivery times for the two scenarios. The curve labeled TCP shows the average burst delivery time on a regular end-to-end TCP connection

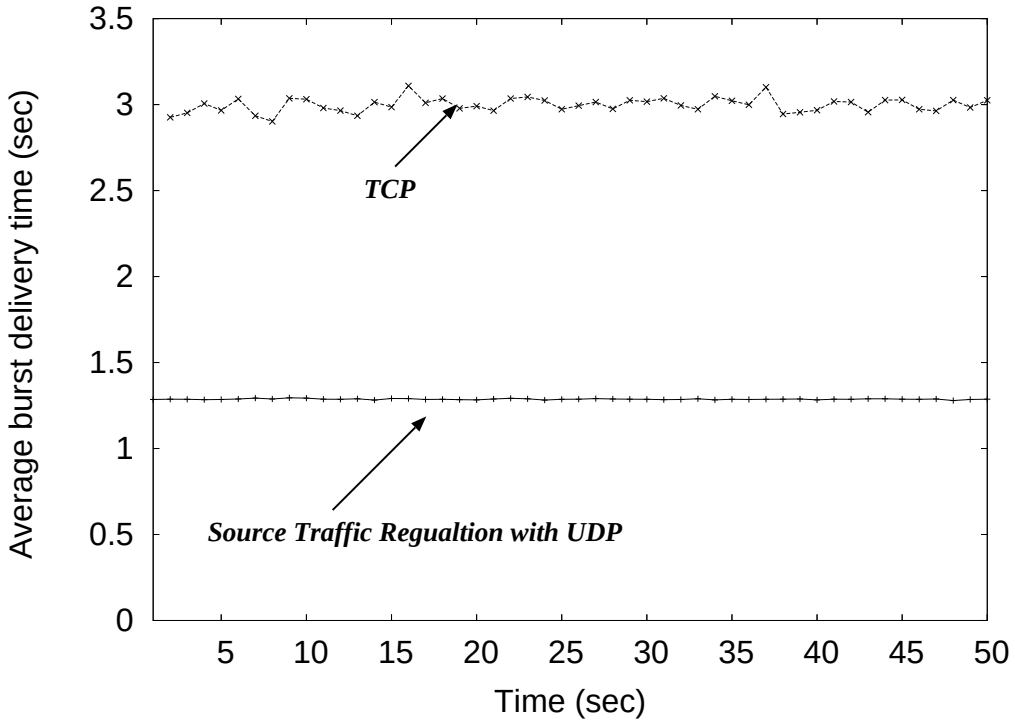


Figure 5.20: Average burst delivery time comparison.

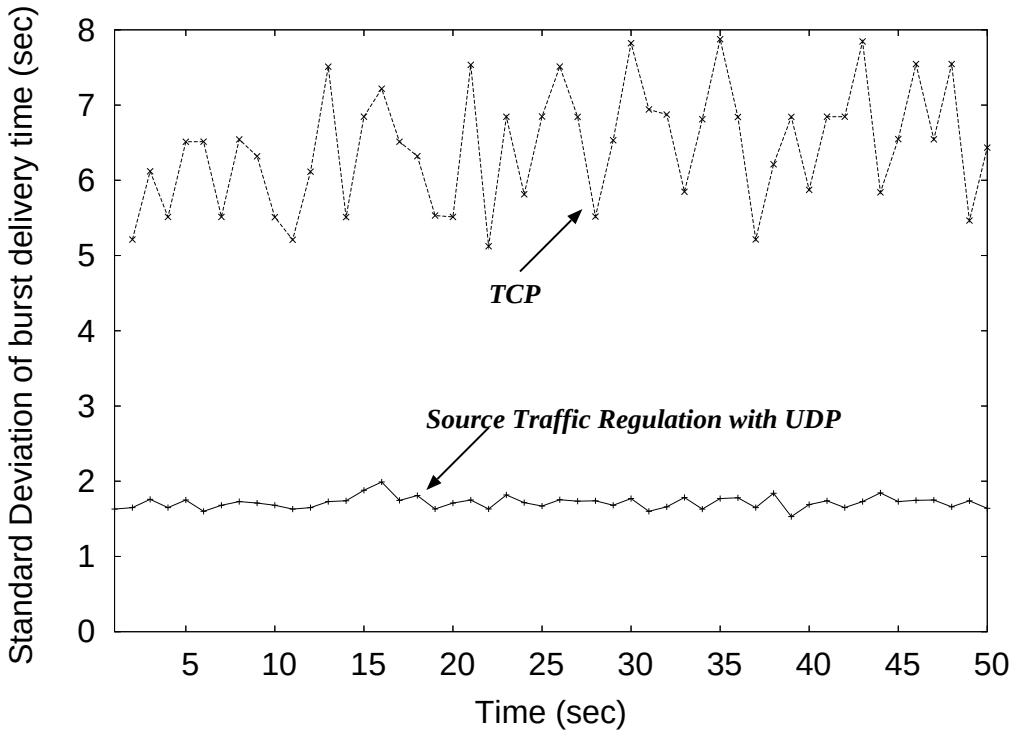


Figure 5.21: Standard deviation of burst delivery time comparison.

between the server and the user. The average burst delivery time is about 3 seconds. The other curve in the plot shows the average burst delivery time in an RDS with source traffic regulation enabled, as illustrated in Figure 5.19. The average burst delivery time is less than 1.5 seconds. It is clear that using RDS with source traffic regulation greatly improves the burst delivery time over regular end-to-end TCP connections.

Figure 5.21 compares the standard deviation of burst delivery time for the two scenarios. It shows that the standard deviation in a network with RDS and source traffic regulation is less than 2 seconds, while the regular end-to-end TCP has much greater standard deviation of 5 seconds or more. This result indicates that source traffic regulation with UDP connections makes the data delivery more smooth than a regular end-to-end TCP connection.

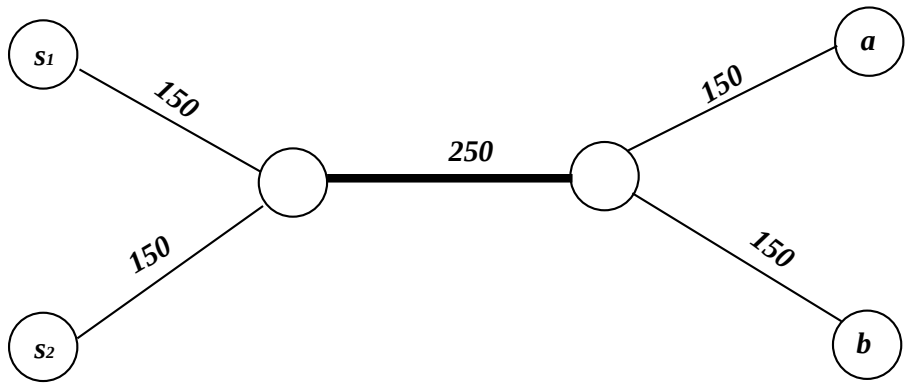


Figure 5.22: Simulation of multi-source traffic regulation in a simple multi-server RDS with two servers.

Regulation in a Multi-server RDS

We simulate the multi-source traffic regulation algorithm in a simple multi-server RDS as shown in Figure 5.22. In these simulations, sinks a and b normally have 50 TCP flows from each of the two servers s_1 and s_2 on average. Each flow is the same as in the single server RDS. a unexpectedly becomes overloaded, and demands 150 flows from s_1 and 300 flows from s_2. All flows start at 0 second, and stop after 50 seconds.

Ideally, a should receive twice as much data from s_2 as the data from s_1 under this overloading situation, such that neither server would get serious input backlog to a. Figure 5.23 shows the total received bandwidth on the sinks from both servers when we only enable single-source traffic regulation on s_1 and s_2 without any coordination between the two servers. Because both servers use source traffic regulation, b gets its average fair share of bandwidth from both s_1 and s_2, and are not affected by the overloaded a. However, because there is no coordination between the s_1 and s_2, the reserved bandwidth is not used efficiently. In particular, when the flows to b are still active, the flow from s_2 to a should get about twice as much bandwidth (100 Mbps on average) as the flow from s_2 to a (50 Mbps on average). However, because both servers do not have any information about the other peer server, they try to compete equally for the reserved bandwidth, and only get about 75 Mbps at the beginning each. The total bandwidth to a and b gradually changes afterwards and approaches the ideal allocation. After the backlog from s_1 to a is cleared after about

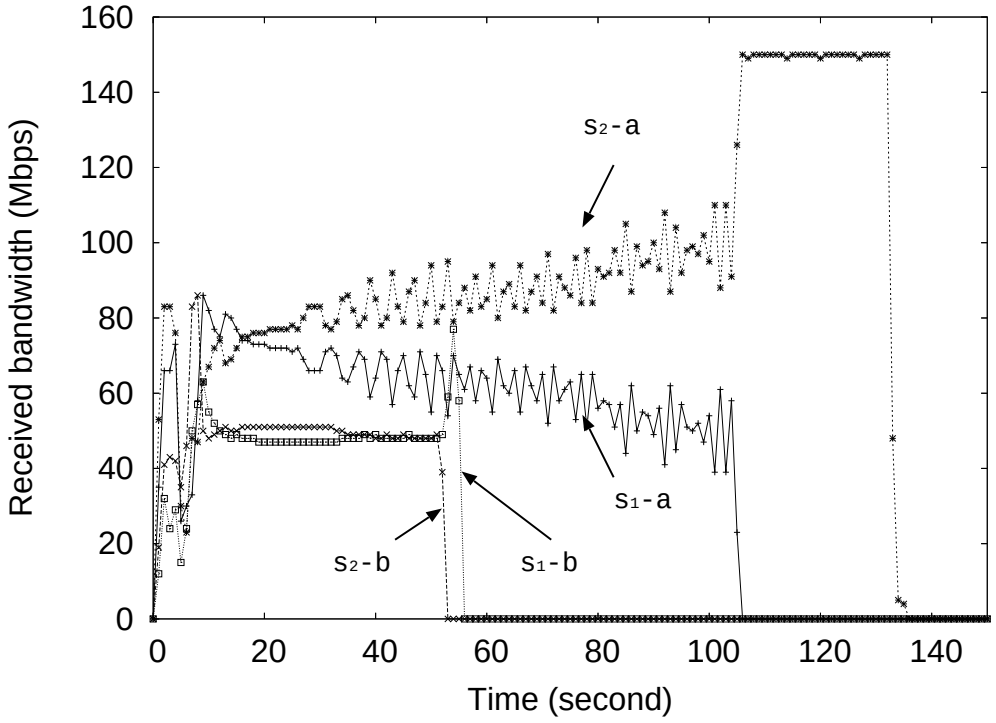


Figure 5.23: Lack of server coordination problem in multi-source traffic regulation (all TCP flows).

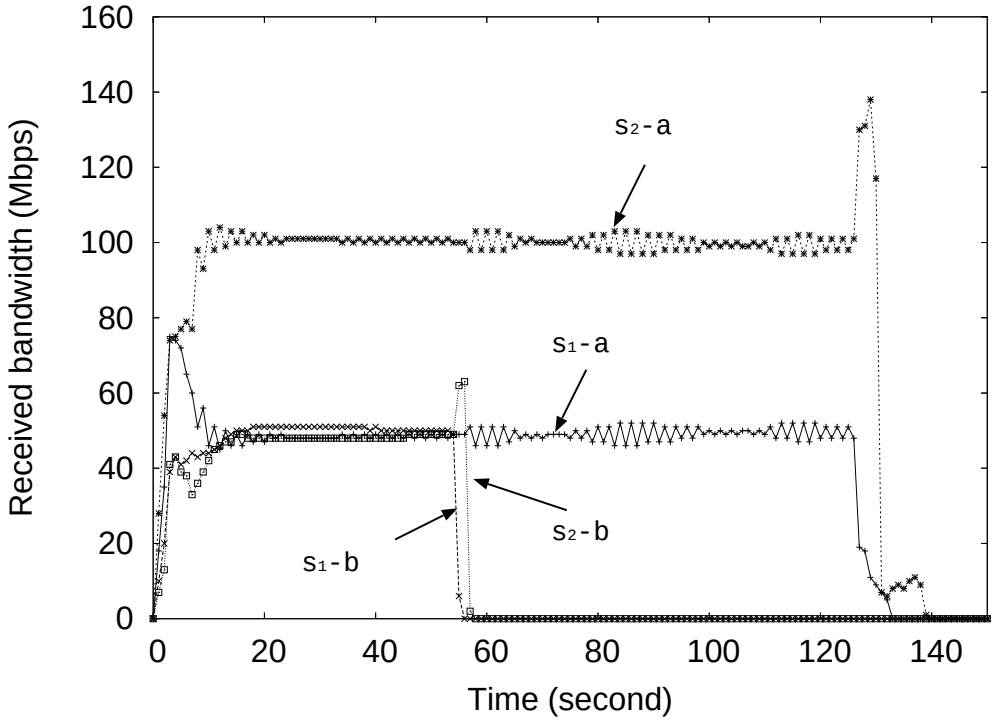


Figure 5.24: Simulation with per-connection multi-source traffic regulation (TCP flows).

108 seconds, flows from s_2 to a takes the maximum bandwidth allowed on its path to a (175 Mbps) and clears up its data backlog.

Figure 5.24 shows the total sink received bandwidth from each server when we enable per-connection multi-source traffic regulation. With the server coordination between the two servers, a receives roughly twice data from s_2 than from s_1, and the input backlogs at both servers are drained at about the same time. As it shows, the flows to b are not affected by the overloaded a, and both receive 50 Mbps average bandwidth. In addition, the $s_2 \rightarrow a$ flows get 100 Mbps average bandwidth, which is twice the bandwidth of $s_1 \rightarrow a$ flows. This is because the two servers exchange their data backlog information with each other, and regulate their flows accordingly to drain their data backlog at about equal rates. As a result, the flows from both s_1 and s_2 to a drain their input data backlog, and finish at about the same time after 130 seconds. The surge after about 128 seconds is caused by the termination of all $s_1 \rightarrow a$ flows.

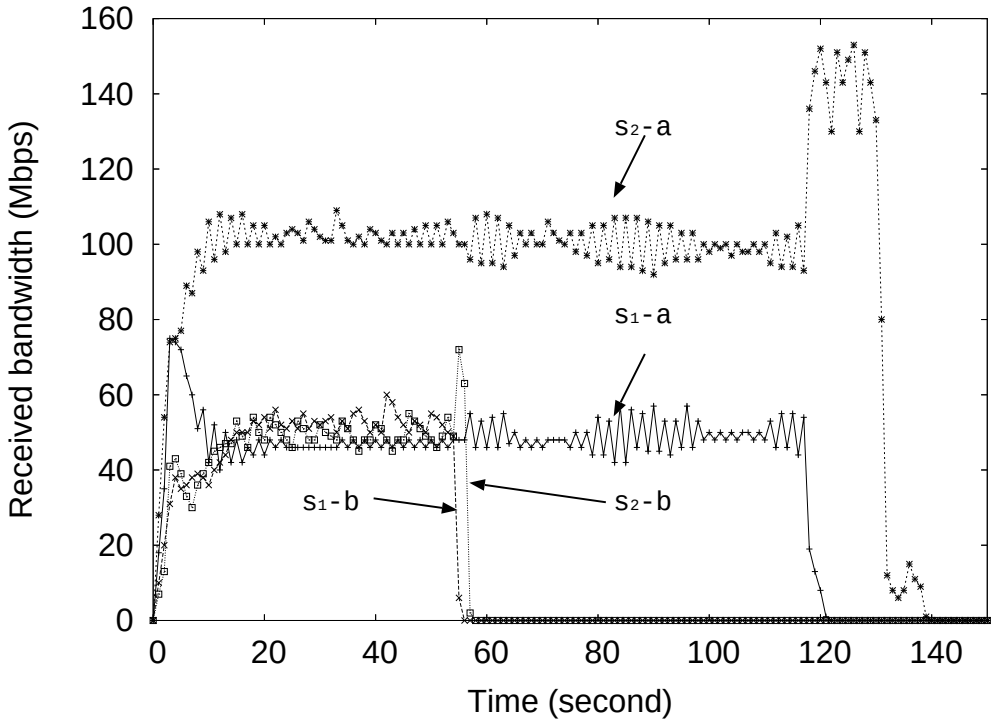


Figure 5.25: Simulation with aggregated multi-source traffic regulation (TCP flows).

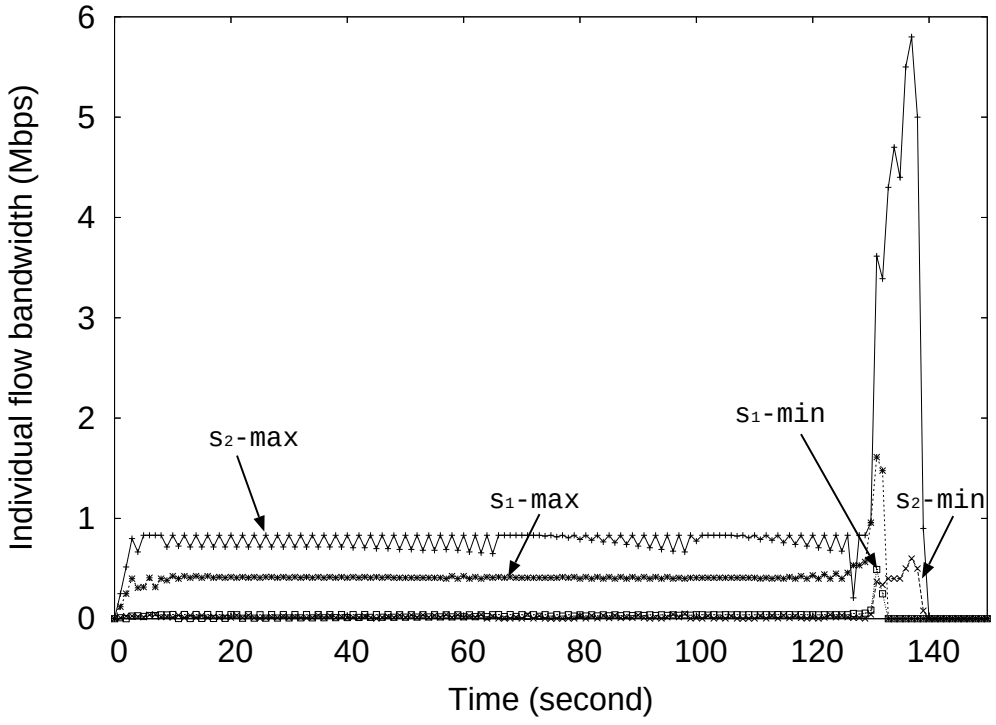


Figure 5.26: Maximum and minimum bandwidth in individual TCP flows from both servers to sink a with per-connection multi-source traffic regulation.

Figure 5.25 shows the total sink received bandwidth from each server when we enable aggregate multi-source traffic regulation. Although the aggregated regulation results in less smooth received bandwidth, it shows that similar results can be achieved with with less control overhead.

Figure 5.26 and Figure 5.27 show the maximum and minimum individual flow bandwidth from the two servers to the sink a in the multi-source traffic regulation simulations with per-connection and aggregate regulation algorithms, respectively.

Figure 5.26 shows that the maximum individual flow bandwidth from s_1 to a is about 0.4 Mbps on average, and that the maximum individual flow bandwidth from s_2 to a is about 0.8 Mbps before backlog to a at s_1 is first cleared out. The maximum individual flow bandwidth increases up to 5.8 Mbps before it quickly drops to 0. The minimum individual flow bandwidth from both servers stays close to 0 until the data backlogs are about to clear up, at which point the minimum individual flow bandwidth from s_1 goes up to 0.4 Mbps

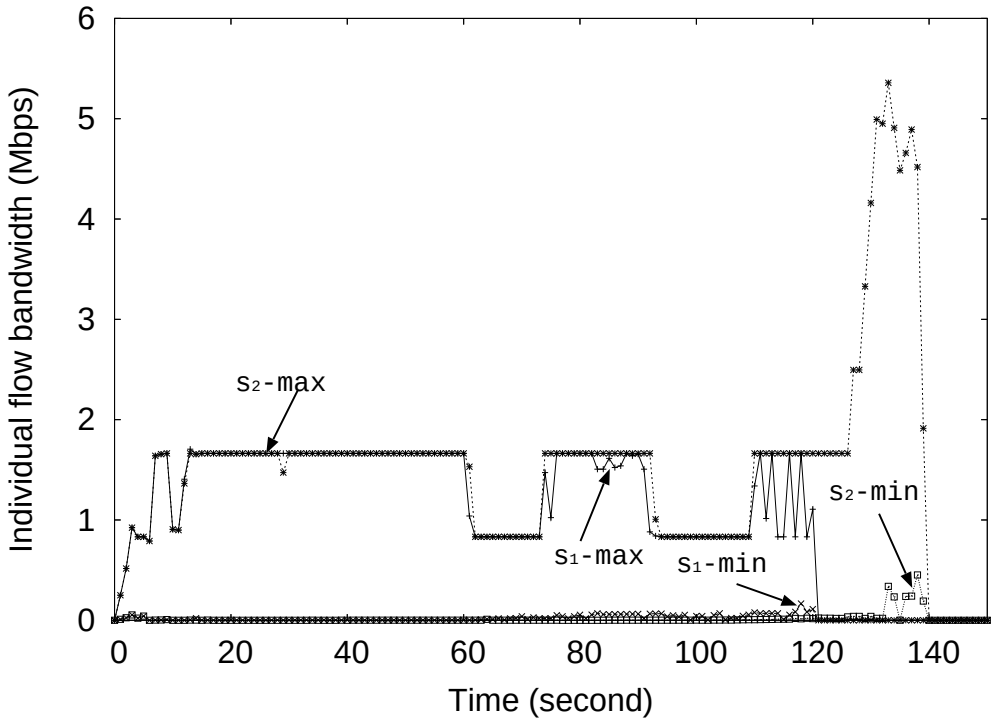


Figure 5.27: Maximum and minimum bandwidth in individual TCP flows from both servers to sink a with aggregate multi-source traffic regulation.

and that from s_2 goes up to about 0.5 Mbps. This shows that the per-connection regulation does not allow individual flows to have a high bandwidth, and that the maximum individual flow bandwidth from s_2 is about twice of that from s_1.

Figure 5.27 shows that the maximum individual flow bandwidth for aggregate regulation is higher than that of per-connection regulation. In particular, the maximum individual flow bandwidth from s_1 and s_2 stays about the same, that ranges from about 0.9 Mbps to 1.6 Mbps before data backlog from s_1 is cleared out. This is more than twice of the bandwidth in the per-connection regulation, because flows are not regulated on individual basis, and some flows get more chances to transmit.

5.6 Implementation on Various Platforms

In this section, we outline the possible implementations of the source traffic regulation algorithms on various platforms. The advantages and limitations of different implementations are discussed.

5.6.1 End Host Implementation

There are two ways to implement source traffic regulation on an end host: special user libraries and kernel modifications.

In the first implementation option, we can develop a set of "wrappers" in a user level library which adds the regulation functionality between the user applications and the network system calls. The major advantage of this approach is its portability. It implements the source traffic regulation without modifying the end host operating systems, and therefore should work on many platforms. However, it comes at the expense of performance due to the additional data operation overhead.

The second approach is to implement the source regulation algorithm at the socket (or an equivalent) layer in the end host operating system kernel to improve the performance. For example, on a BSD based operating system (such as NetBSD [78]), the regulation algorithm can be implemented in the socket layer with extensions to the *socket* structure and addition

of new traffic regulation socket options. In particular, the socket buffer and related system calls (for example, *recvmsg* and *sendmsg*) can be modified to include rate regulation such that user data stored in the socket buffer are sent out in a regulated fashion when the source traffic regulation option is enabled. In addition, such an implementation makes it efficient and convenient to exchange information between the regulator and the collectors with a reliable connection (such as TCP). The major drawback of this implementation approach is its complexity and poor portability because it must be implemented on end hosts at both ends of a connection.

The major advantage of an end host implementation is that it does not require changes to the access routers and proxies. However, it is not as easy to implement aggregated regulation as on the other platforms, and it needs operating systems support on both sides of connections.

5.6.2 Stand-alone Proxy Implementation

Another platform to implement the source regulation is the performance enhancing proxies [5] at source and sink sides.

The regulator can be implemented at the source side proxy, and the collector can be implemented on the sink side proxies, both as a performance improvement mechanism in addition to the other commonly used ones, such as ACK handling, compression, priority-based multiplexing, and protocol boosters. One of the most popular open source web cache proxy system, Squid [73], derived originally from the Harvest project [6], provides a good proxy platform to implement traffic regulation algorithms. In particular, the functions of a regulator or a collector can be added to the *ConnStateData* structure and the core data communication routine (*comm_select*()) with rate regulation using *DelayPool* classes.

The major advantages of a proxy implementation is the flexibility of this approach and ease of deployment. A stand-alone proxy is suitable to implement both the per-connection and aggregated regulation equally well. It does not require any changes on the routers or the end hosts. An end host can simply choose to go through a proxy in his connection setting to use the source traffic regulation. The major drawback of this implementation is the possible performance limitation, because of the extra hops to and from the proxy and the proxy processing overhead.

5.6.3 Extensible Router Plug-in Implementation

Extensible routers, such as the dynamically extensible router introduced in [13], provide another platform for source traffic regulation as well as other performance enhancing mechanisms that require moderate processing overheads.

Take the above dynamically extensible router for example. On each input and output port, there is a software-based packet processing smart port card (SPC) as well as a programmable hardware device called field programmable port extender (FPX). The SPC can use loadable modules to process data packets at a very high speed, and the FPX is capable of dynamically loading hardware modules onto the on-board FPGA for high speed hardware-based packet processing. By implementing the regulator and the collector as two SPC loadable modules, with the help of the FPX for the bulk of IP processing and buffering, the traffic regulation can be performed very efficiently. This platform is also a good choice if we want to handle large number of flows.

The major difficulty of this implementation approach is the possible resource limitation. In particular, when there are sustained overloading, the data backlog may increase to a point that the memory resource on a port is exhausted.

5.7 Summary

In this chapter, we show that besides the benefit of exclusive bandwidth access, the end-to-end performance can be further improved in an RDS by utilizing the knowledge about the underlying network. Specifically, we introduce source traffic regulation to resolve the unbalanced bandwidth utilization problem inside an RDS. The source traffic regulation ensures that all traffic flows are within the constraints of reserved bandwidth on the end-to-end path; in addition, it regulates the source transmission rates to different end hosts in such way that bandwidth utilization on all links is balanced to protect a sink from ill-behaved overloading sinks. We study our proposed per-connection and aggregated traffic regulation algorithms with simulations in the network simulator, and our simulation results demonstrate the improved end-to-end performance with source traffic regulation.

Chapter 6

Conclusions and Future Work

The Internet must provide services with a certain level of bandwidth assurance before it can become a more reliable and trustworthy information infrastructure. However, per-flow bandwidth reservation services have not been widely deployed as expected in today's Internet. Toward this end, we proposed a reserved delivery subnetwork (RDS) service that provisions aggregate bandwidth reservations for groups of users. An RDS is more easily deployed than per-flow reservation services, and provides more consistent quality of service than best-effort forwarding. In the preceding chapters of this dissertation, we study a number of design issues with the configuration, deployment, and operation of an RDS. Besides these topics we have covered in this dissertation, there are a number of related issues that can be further explored in future research.

6.1 Reserved Delivery Subnetworks

The reserved delivery subnetwork was introduced in Chapter 2 as an alternative way to provide more consistent quality of service within today's Internet infrastructure. Instead of deploying per-flow bandwidth reservation services, exclusive bandwidth is reserved for an aggregated group of customers of a service provider, to circumvent the deployment problem encountered by per-flow bandwidth reservation services. The deployment of such a service will benefit a number of network applications such as web content delivery, virtual private networks, and grid computing.

In this dissertation, we have focused on the configuration and deployment of a generic reserved delivery subnetwork. Less attention has been paid to the issues about how a specific

network application can benefit from the deployment of an RDS. For example, although a web content delivery service can naturally be deployed on a generic RDS, the deployment of a VPN service over an RDS may put extra requirements on the configuration of the underlying RDS because the asymmetric bandwidth assumption is no longer a constraint. In addition, data security and service stability are crucial in a VPN. Therefore, we may include security and stability considerations in the configuration and deployment of an RDS VPN. One possible strategy is to integrate the security and stability factors into the link cost function for the configuration of an RDS VPN.

Grid computing is another potential application of RDS. In particular, the RDS service can facilitate resource management in a grid computing application that uses the resources of a potential large number of computers connected by a network to solve a large-scale computation problem. Traditionally, the research focus has been put on the computational resource discovery and allocation of different nodes in a computational grid. Relatively little attention has been paid to the the management and allocation of bandwidth resources in the network used by the grid. We think this issue is equally important to the performance of a computational grid, and deserves more study. Similar techniques for configuring and deploying an RDS can apply to the bandwidth resource management problem in a grid computing application. In particular, link selection for a computational grid should also consider the economy of bandwidth aggregation so that communication cost is minimized.

6.2 RDS Configuration

The configuration of an RDS involves two tasks: selecting the subnetwork and determining the appropriate bandwidth reservation on links in the subnetwork. In Chapter 3, we start with configuration of the basic RDS with a single server. We formulate the configuration problem of such an RDS as a minimum concave cost network flow problem, where the per unit flow cost increments decrease as the current flow increases. This problem formulation takes the economy of bandwidth aggregation into consideration and is more practical, but it also makes the configuration problem a NP-hard problem. Traditional enumerative search-based exact algorithms are not practical even for a network with moderate size. An approximate heuristic (LDF) based on least cost augmentation algorithm has been presented to solve the problem efficiently. Our simulation results indicate that LDF creates

results that are within a constant factor of an estimated lower bound to the optimal solution. We used an easily computed lower bound and estimated lower bounds derived from this lower bound to evaluate the performance of our proposed algorithm. However, this lower bound is very loose. Thus, a better lower bound that is tighter than the lower bound we used and has comparable computational complexity would be worth studying.

To further improve the results from LDF, we apply local search heuristics on the LDF results. We started with a traditional negative cost cycle reduction algorithm first, and found that a special negative cost multi-cycle subnetwork structure can also be used to further reduce the cost of an RDS. We implemented and studied the performance of a local search algorithm based on negative cost bi-cycle reduction. Our simulation results show that although local search algorithms based on negative cost cycle and bi-cycle reduction can greatly improve the results of an arbitrary initial solution, the improvement to LDF is limited. We think this is a strong indication that LDF solutions are close to optimal.

We have only studied the simplest negative cost multi-cycles, bi-cycles, in our study. When we consider negative cost multi-cycle with more cycles, the computational complexity grows substantially. It would be interesting to study the tradeoff of computational complexity and performance improvements to find out a point of diminished returns.

In Chapter 4, we study the configuration of RDSs with multiple servers. We can transform this problem into a single server RDS configuration with an additional pseudo server, but there is a unique server placement issue for a multi-server RDS configuration that complicates the configuration. We have studied a variety of server placement algorithms, and our simulation results indicate that a class of greedy algorithms out-perform other server placement algorithms.

6.3 RDS Fault Tolerance

Also in Chapter 4, we have studied a method to improve the fault tolerance of a multi-server RDS. In particular, we study the problem of setting up redirection subnetworks for groups of up to four server in a multi-server RDS. The redirection subnetwork for a group of servers redirects traffic from a faulty or overloaded server to other “healthy” servers in

the group, utilizing the existing RDS links to the maximum extent. We use a recursive approach to build up the redirection subnetworks. In particular, we started with a simple problem of finding the optimal server pairs that traffic to one server in a server pair can be redirected to the other server in the pair. For each pair of servers, a redirection subnetwork is configured to allow traffic redirection from one server to the sinks of the other server. To generalize to groups of four servers, we start with the server pairs already obtained in the first step, and find optimal pairs of server pairs to form groups of four servers. For each group of four servers, identify a center redirection point and configure the redirection subnetwork to redirect traffic from the sinks of one server to the other three servers.

There are several possible studies we could pursue in the future. First, in some cases, not all sinks have to be covered by a redirection server, especially when the original server is only overloaded briefly. Instead, we can configure the redirection subnetwork to partially cover the sinks connected to a server. Second, we can also apply some simple local search heuristic to improve the solution quality. For example, we can try to adjust the server location locally, so that the total cost of the original RDS and the redirection subnetwork is lower, although this move may increase the cost of the original RDS.

6.4 RDS End-to-end Performance Improvements

In Chapter 5, we have investigated an option for potential performance gains of end-to-end applications in an RDS. By leveraging the knowledge about the underlying RDS network, we try to improve the end-to-end performance by solving the unbalanced bandwidth utilization problem with source traffic regulation. Without any traffic regulation at the server side, an overloaded sink would congest the upstream path to the server, reducing the bandwidth utilization and service quality at other sinks that share part of the congested path. By enabling source traffic regulation, the server controls its traffic to a specific sink according to the data backlogs at both ends of a connection and the traffic condition in the RDS. Our simulations have shown that all sinks get their fair share of reserved bandwidth and only the overloaded sinks are penalized. In addition, the traffic regulation mechanism improves the TCP fairness when the round trip delay to sinks is large.

Source traffic regulation was inspired by the distributed queueing techniques in high speed routers [58]. It would be interesting to study the work conservation property in the RDS context, and determine the speed-up factor needed to achieve work conservation. In this chapter, we also have outlined a number of implementations of source traffic regulation on three platforms. It would be interesting to implement them and evaluate RDS in a real network environment.

References

[1] E. Aarts and J. K. Lentra. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.

[2] Ravindra K. Ahuja, Thomas Magnanti, and James Orlin. *Network Flows*. Prentice Hall, 1993.

[3] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Proceedings of 18th ACM SOSP*, Banff, Canada, October 2001.

[4] Francisco Barahona and Eva Tardos. Note on Weintraub's minimum-cost circulation algorithm. *SIAM Journal of Computing*, 18(3):579–583, June 1989.

[5] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. RFC 3135: Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, June 2001.

[6] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28(1–2):119–126, 1995.

[7] R. Braden, D. Clark, and S. Shenker. Rfc 1633: Integrated services in the Internet architecture: an overview, 1994.

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

[9] J. Cao, D. Davis, S. Wiel, and B. Yu. Time-varying network tomography : Router link data. Technical report, Bell Labs Tech. Memo, 2000.

[10] Jin Cao, D. Davis, Scott Vander Wiel, Bin Yu, and Zhengyuan Zhu. A scalable method for estimating network traffic matrices from link counts. Technical report, Bell Labs Tech Report, 2001.

[11] Boris V. Cherkassky and Andrew V. Goldberg. Negative-cycle detection algorithms. In *European Symposium on Algorithms*, pages 349–363, Barcelona, Spain, March 1996.

[12] S. Choi and Y. Shavitt. Placing servers for session-oriented services. Technical report, Technical Report WUCS-01-41, Washington University at St. Louis, Dept. of Computer Science., 2001.

[13] Sumi Choi, John Dehart, Ralph Keller, Fred Kuhns, John Lockwood, Prashanth Pappu, Jyoti Parwatikar, W. David Richard, Ed Spitznagel, David Taylor, Jonathan Turner, and Ken Wong. Design of a high performance dynamically extensible router. In *Proceeding of DARPA Active Networks Conference and Exposition (DANCE)*, San Francisco, CA, USA, May 2002.

[14] Israel Cidon, Shay Kutten, and Ran Soffer. Optimal allocation of electronic content. In *INFOCOM*, pages 1773–1780, 2001.

[15] David Clark and William Lehr. Provisioning for bursty internet traffic: Implications for industry and internet structure. In *Proceedings of MIT ITC Workshop on Internet Quality of Service (MIT WISQ 1999)*, Boston, MA, USA, December 1999.

[16] CNN. Computer worm grounds flights, blocks ATMs. URL http://www.cnn.com/2003/TECH/internet/01/25/internet.attack/.

[17] M. B. Doar. A better model for generating test networks. In *Proceedings of Global Internet, Globecom '96*, November 1996.

[18] Zhenhai Duan, Zhi-Li Zhang, and Yiwei Thomas Hou. Service Overlay Networks: SLAs, QoS and Bandwidth Provisioning. In *Proceedings of 10th IEEE International Conference on Network Protocols (ICNP)*, Paris, France, Novmber 2002.

[19] Anja Feldmann, Albert G. Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving traffic demands for operational IP networks: methodology and experience. In *SIGCOMM*, pages 257–270, 2000.

[20] D.C. Feldmeier, A.J. McAuley, J.M. Smith, D.S. Bakin, W.S. Marcus, and T.M. Raleigh. Protocol boosters. *IEEE Journal on Selected Areas of Communication*, 16(3), April 1998.

[21] J. A. Fingerhut. *Approximation Algorithms for Configuring Nonblocking Communication Networks*. D. Sc. dissertation, Washington University, St. Louis, Missouri, May 1994.

[22] J. Andrew Fingerhut, Subhash Suri, and Jonathan S. Turner. Designing least-cost nonblocking broadband networks. *Journal of Algorithms*, 24(2):287–309, August 1997.

[23] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[24] Dalila B. M. M. Fontes, Eleni Hadjiconstantinou, and Nicos Christofides. A new branch-and-bound algorithm for network design using concave cost flows. Technical report, Imperial College, London, UK, 2002.

[25] Dalila B. M. M. Fontes, Eleni Hadjiconstantinou, and Nicos Christofides. Upper bounds for single-source uncapacitated concave minimum-cost network flow problems. *Networks*, 41(4):221–228, July 2003.

[26] Chuck Fraleigh, Fouad Tobagi, and Christophe Diot. Provisioning IP Backbone Networks to Support Latency Sensitive Traffic. In *Proceedings of IEEE InfoComm*, San Francisco, CA, USA, April 2003.

[27] G. Gallo and C. Sodini. Adjacent extreme flows and application to min concave cost flow problems. *Networks*, 9:95–121, 1979.

[28] GNU. GNU Scientific Library. URL http://www.gnu.org/software/gsl/.

[29] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *Journal of ACM*, 36:873–886, 1989.

[30] G. M. Guisewite and P. M. Pardalos. Minimum concave-cost network flow problems: Applications, complexity, and algorithms. *Annals of Operations Research*, 25:75–99, 1990.

[31] G. M. Guisewite and P. M. Pardalos. Algorithms for the single-source uncapacitated minimum concave-cost network flow problem. *Journal of Global Optimization*, 1:245–265, 1991.

[32] G. M. Guisewite and P. M. Pardalos. Global search algorithms for minimum concave-cost network flow problems. *Journal of Global Optimization*, 1:309–330, 1991.

[33] Dorit S. Hochbaum, editor. *Approximation Algorithms for HP-hard Problems*. PWS Publishing Company, 1993.

[34] R. Horst, P. M. Pardalos, and N. V. Thoai. *Introduction to Global Optimization*. Kluwer Academic Publishers, 1995.

[35] R. Horst and H. Tuy. *Global Optimization*. Springer-Verlag, 1993.

[36] Sundar Iyer, Supratik Bhattacharyya, Nina Taft, and Christophe Diot. An approach to alleviate link overload as observed on an IP backbone. In *Proceedings of IEEE InfoComm*, San Francisco, CA, USA, April 2003.

[37] Van Jacobson. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM*, pages 314–329, Palo Alto, CA, USA, August 1988.

[38] Sugih Jamin, Cheng Jin, Yixin Jin, Danny Raz, Yuval Shavitt, and Lixia Zhang. On the placement of internet instrumentation. In *IEEE INFOCOM*, pages 295–304, 2000.

[39] Sugih Jamin, Cheng Jin, Anthony R. Kurc, Danny Raz, and Yuval Shavitt. Constrained mirror placement on the internet. In *IEEE INFOCOM*, pages 31–40, 2001.

[40] Alpr Jttner, Istvn Szab, and ron Szentesi. On Bandwidth Efficiency of the Hose Resource Management Model in Virtual Private Networks. In *Proceedings of IEEE InfoComm*, San Francisco, CA, USA, April 2003.

[41] M. Klein. A primal method for minimal cost flows. *Management Science*, 14:205–220, 1967.

[42] Korupolu, Plaxton, and Rajaraman. Placement algorithms for hierarchical cooperative caching. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1999.

[43] Anshul Kothari, Subhash Suri, and Yunhong Zhou. Bandwidth constrained allocation in grid computing. In *Proceedings of Workshop on Algorithms and Data Structures (WADS'03)*, Ottawa, Canada, July 2003.

[44] Balachander Krishnamurthy and Jia Wang. On network-aware clustering of web clients. In *SIGCOMM*, pages 97–110, 2000.

[45] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed internet routing convergence. In *Proc. ACM SIGCOMM*, pages 175–187, Stockholm, Sweden, 2000.

[46] C. Labovitz, R. Wattenhofer, S. Venkatachary, and A. Ahuja. The impact of Internet policy and topology on delayed routing convergence. In *Proc. IEEE INFOCOM*, April 2001.

[47] Bruce W. Lamar. An imporved branch and bound algorithm for minimum concave cost network flow problems. *Journal of Global Optimization*, 3:261–287, 1993.

[48] Will E. Leland, Murad S. Taqq, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic. In Deepinder P. Sidhu, editor, *ACM SIGCOMM*, pages 183–193, San Francisco, California, 1993.

[49] Will E. Leland and Daniel V. Wilson. High time-resolution measurement and analysis of LAN traffic: Implications for LAN interconnection. In *INFOCOM (3)*, pages 1360–1366, 1991.

[50] Bo Li, Mordecai J. Golin, Giuseppe F. Italiano, Xin Deng, and Kazem Sohraby. On the optimal placement of web proxies in the internet. In *IEEE INFOCOM*, pages 1282–1290, 1999.

[51] Dong Lin and Robert Morris. Dynamics of random early detection. In *Proceedings of ACM SIGCOMM*, pages 127–137, Cannes, France, September 1997.

[52] Hongzhou Ma, Inderjeet Singh, and Jonathan S. Turner. Constraint based design of ATM networks, an experimental study. Technical Report WUCS-9715, Department of Computer Science, Washington University, 1997.

[53] R. Mahajan, S. Floyd, and D. Wetherall. Controlling high-bandwidth flows at the congested router. In *Proc. IEEE 9th International Conference on Network Protocols (ICNP)*, November 2001.

[54] A. Medina, N. Taft, S. Battacharya, C. Diot, and K. Salamatian. Traffic matrix estimation: Existing techniques compared and new directions. In *SIGCOMM*, Pittsburgh, PA, USA, August 2002.

[55] Alberto Medina, Anukool Lakhina, Ibrahim Matta, , and John Byers. BRITE: An approach to universal topology generation. In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS '01)*, Cincinnati, Ohio, USA, August 2001.

[56] Debasis Mitra and Qiong Wang. Stochastic Traffic Engineering, with Applications to Network Revenue Management. In *Proceedings of IEEE InfoComm*, San Francisco, CA, USA, April 2003.

[57] Rong Pan, Balaji Prabhakar, and Konstantinos Psounis. CHOKE, a stateless active queue management scheme for approximating fair bandwidth allocation. In *Proceedings of IEEE INFOCOM (2)*, pages 942–951, 2000.

[58] Prashanth Pappu, Jyoti Parwatikar, Jonathan Turner, and Ken Wong. Distributed queueing in scalable high performance routers. In *Proceeding of IEEE Infocom*, San Francisco, CA, USA, April 2003.

[59] Vern Paxson and Sally Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[60] L. Qiu, V. Padmanabham, and G. Voelker. On the placement of web server replicas. In *Proceedings of IEEE INFOCOM 2001*, Anchorage, AK, USA, April 2001.

[61] Ruibiao Qiu. Reserved delivery subnetworks configuration algorithm with the maximum sharing shortest path tree. In *SPIE Conference on Performance and Control of Next Generation Communication Networks, ITCom*, Orlando, FL, USA, September 2003.

[62] Ruibiao Qiu and Jonathan S. Turner. Configuration of reserved delivery subnetworks. In *Proceedings of IEEE Globecom*, Taipei, Taiwan, Novmber 2002.

[63] Ruibiao Qiu and Jonathan S. Turner. Approximation algorithm for reserved delivery subnetwork configuration. Technical Report WUCS-0352, Department of Computer Science and Engineering, Washington University, 2003.

[64] Ruibiao Qiu and Jonathan S. Turner. Improved local search algorithm with multi-cycle reduction for minimum concave cost network flow problems. Technical report, WUCS-04-74, Washington University at St. Louis, Department of Computer Science and Engineering, 2004.

[65] Ruibiao Qiu and Jonathan S. Turner. Configuring multi-server reserved delivery subnetworks. Technical report, WUCS-05-01, Washington University at St. Louis, Department of Computer Science and Engineering, 2005.

[66] P. Radoslavov, R. Govindan, and D. Estrin. Topology-informed internet replica placement. In *Proceedings of WCW'01: Web Caching and Content Distribution Workshop, Boston, MA*, June 2001.

[67] Pablo Rodriguez and Sandeep Sibal. SPREAD: Scalable platform for reliable and efficient automated distribution. *WWW9 / Computer Networks*, 33(1-6):33–49, 2000.

[68] M. Roughan, A. Greenberg, C. Kalmanek, M. Rumsewicz, J. Yates, and Y. Zhang. Experience in measuring backbone traffic variability: Models, metrics, measurements and meaning. In *ACM SIGCOMM Internet Measurement Workshop*, 2002.

[69] Stefan Savage, Tom Anderson, Amit Aggarwal, David Becker, Neal Cardwell, Andy Collins, Eric Hoffman, John Snell, Amin Vahdat, Geoff Voelker, and John Zahorjan. Detour: a Case for Informed Internet Routing and Transport. *IEEE Micro*, 19(1):50–59, January 1999.

[70] Stefan Savage, Neal Cardwell, and Tom Anderson. The case for informed transport protocols. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, USA, March 1999.

[71] Stefan Savage, Andy Collins, Eric Hoffman, John Snell, and Tom Anderson. The end-to-end effects of Internet path selection. In *Proceedings of the ACM SIGCOMM Conference*, pages 289–299, Cambridge, MA, USA, September 1999.

[72] Sherlia Shi and Jonathan S. Turner. Placing servers in overlay networks. In *Proc. Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, San Diego, CA, USA, July 2002.

[73] Squid Web Cache Proxy. URL http://www.squid-cache.org/.

[74] A. Steger, E. Mayr, and H. Prmel, editors. *Lectures on Proof Verification and Approximation Algorithms*, volume 1367. Springer, 1998.

[75] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Network topology generators: Degree-based vs structural. In *Proceedings of ACM SIGCOMM 2002*, August 2002.

[76] Robert Endre Tarjan. *Data Structure and Network Algorithms*, volume 44. Society for Industrial and Applied Mathematics, 1983.

[77] P. T. Thach. A decomposition method using a pricing mechanism for min concave cost flow problems with hierarchical structure. *Journal of Mathematical Programming*, 53:339–359, 1992.

[78] The NetBSD Foundation. NetBSD. URL http://www.netbsd.org/.

[79] U.S. Census Bureau. Census 2000. URL http://www.census.gov/population/www/cen2000/.

[80] VINT. Network Simulator. URL http://www.isi.edu/nsnam/ns/.

[81] B. M. Waxman. Routing of multipoint connections. *IEEE Jounral of Selected Areas in Communications*, 6(9):1617–1622, 1988.

[82] Jared Winick and Sugih Jamin. Inet-3.0: Internet topology generator. Technical Report UM-CSE-TR-456-02, Department of Computer Science and Engineering, University of Michigan, 2002.

[83] X. Xiao and L. M. Ni. Internet QoS: A big picture. *IEEE Network*, 13(2):8–18, March 1999.

[84] Guo-Liang Xue and Shang-Zhi Sun. *The shortest path network and its applications in bicriteria shortest path problems*, pages 355–362. World Scientific Publishing Co., 1993.

[85] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *IEEE Infocom*, volume 2, pages 594–602, San Francisco, CA, USA, March 1996.

[86] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. RSVP: A new resource reservation protocol. *IEEE Network Magazine*, September 1993.

[87] Weixiong Zhang. *State-Space Search*. Springer-Verlag New York Inc., 1999.

[88] Y. Zhang, M. Roughan, N. Duffield, and A. Greenberg. Fast accurate computation of large-scale ip traffic matrices from link loads. In *ACM SIGMETRICS*, San Diego, CA, USA, June 2003.

[89] Y. Zhang, M. Roughan, C. Lund, and D. Donoho. An information-theoretic approach to traffic matrix estimation. In *ACM SIGCOMM*, August 2003.

Vita

Ruibiao Qiu

Degrees

B.S. Beijing University of Posts and Telecommunications, Computer Engineering, July 1992
M.S. Beijing University of Posts and Telecommmunications, Computer Engineering, May 1995
M.S. Florida International University, Computer Science, December 1997
D.Sc. Washington University in St. Louis, Computer Science, May 2006

Professional Societies

Association for Computing Machines (ACM)
Institute of Electrical and Electronics Engineers (IEEE)
The NetBSD Foundation
The International Society for Optical Engineering (SPIE)
Institute of Electrical, Information and Communications Engineers (IEICE)

Selected Publications

Ruibiao Qiu, Jonathan S. Turner, ”Source Traffic Regulation in Reserved Delivery Subnetworks”. Proceedings of 25th IEEE International Performance Computing and Communications Conference (IPCCC), Phoenix, AZ, April, 2006.

Ruibiao Qiu, Jonathan S. Turner, ”Local Search Algorithms for Reserved Delivery Subnetwork Configuration Problems with Cycle and Bicycle Reduction”. Proceedings of Advances for Networks & Internet Symposium,IEEE Globecom 2005, St. Louis, MO, November, 2005.

Qiu, Ruibiao and Turner, Jonathan S. Configuration of Reserved Delivery Subnetworks. Proceedings of Service Infrastructure for Virtual Enterprises Symposium, IEEE Globecom 2002, Taipei, Taiwan, November 2002.

Qiu, Ruibiao, Cox, Jerome R., and Kuhns, Fred, A Conference Control Protocol for Highly Interactive Video-conferencing. Proceedings of IEEE Globecom 2002, Taipei, Taiwan, November 2002.

Qiu, Ruibiao, Kuhns, Fred, Cox, Jerome R. and Horn, Craig, Bringing Studio Quality Video-conferencing to Wide Area IP Networks with an Adaptation Layer Translator (ALX). Proceedings of IEEE International Conference on Multimedia and Expo (ICME 2002), Lausanne, Switzerland, August 2002.

Yu, Wei, Qiu, Ruibiao, Fritts, Jason, Motion-JPEG2000 Video Transmission over Active Networks. Proceedings of Image and Video Communications and Processing Conference at IS&T/SPIE Electronic Imaging 2003, Santa Clara, CA, January 2003.

Qiu, Ruibiao, Kuhns, Fred, Cox, Jerome, Horn, Craig, High Quality Videoconferencing System for Wide Area IP Networks. Proceedings of SPIE ITCom 2002, Boston, MA, 07/02.

Yu, Wei, Qiu, Ruibiao and Fritts, Jason, Advantages of Motion-JPEG2000 in Video Processing. Proceeding of SPIE Visual Communications and Image Processing (VCIP02), San Jose, CA, January 2002.

May 2006