修　　士　　論　　文

分散環境でのStableなブロードキャスト アルゴリズムの提案と実装

A Stable Broadcast Algorithm for Distributed Environments and Its Implementation

指導教員　**田浦 健次朗 准教授**

東京大学 情報理工学系研究科
電子情報学専攻

氏 名　**56428 高橋 慧**

提 出 日　**平成 20 年 2 月 4 日**

Abstract

Distributed environments have greatly contributed to many scientific fields by providing large scale computational resources. Data intensive applications, which treat much amount of data, such as documents on the web, are one of the applications requiring a large amount of computational resources. Since data transfer takes considerable part of the total execution time of data intensive applications, it is inevitable to efficiently perform data movements in order to effectively execute data intensive applications.

In particular, distributing large data to many nodes, known as a broadcast or a multicast, is an important operation in parallel and distributed computing. Most previous broadcast algorithms explicitly or implicitly try to deliver data to all nodes in the same rate. While being reasonable for homogeneous environments where all nodes have similar receiving capabilities, such algorithms may suffer from slowly receiving nodes in heterogeneous settings. In such settings, each node desires to receive data at its largest possible bandwidth and start computation as soon as it receives the data.

We propose to say a broadcast is *stable* when the bandwidth to a node is never sacrificed by the presence of other, possibly slow, receiving nodes, and proposes the stability as a desired property of broadcast algorithms. In addition, we show a simple and efficient stable broadcast when the topology among nodes is a tree and each link has a symmetric bandwidth. Our algorithm improves upon previously proposed algorithms such as FPFR and Balanced Multicast. For general graphs, it outperforms them when the network is heterogeneous and for trees, it is stable and optimal.

Our broadcast algorithm is implemented on a tool named *ucp*. With our stable broadcast algorithm, it efficiently delivers a large amount of data to many nodes. Since multiple hosts and files can easily be specified with regular expressions, it suits for interactive use from a terminal. In addition, NATs and firewalls are can be traversed by relaying technique.

In a real environment with 100 machines in 4 clusters, our scheme achieved 2.1 to 2.6 times aggregate bandwidth compared to the best result in the other algorithms. We also demonstrated the stability by adding a slow node to a broadcast. Some simulations also showed that our algorithm also performs well in many bandwidth distributions. In heterogeneous environments, our scheme yielded twice the aggregate bandwidth compared to a simpler algorithm using a single pipeline.

Acknowledgements

First of all, I would like to express my sincere gratitude and appreciation to my supervisors Professor Takashi Chikayama and Associate Professor Kenjiro Taura. Professor Chikayama's insightful advice and suggestions was indispensable for me. Associate Professor Taura's expert guidance always supported me throughout my research. This work would not been possible without his warm and constructive input. I also would like to thank him for providing me a chance to design the ICPC's poster and T-shirt, as well as many other websites.

I am deeply indebted to all the members in the laboratory, who supported my research life. Takeshi Shibata offered much-appreciated support especially with my first referred paper. His insight based on his theoretical background was paramount. I learned much from Yoshikazu Kamoshida about system programming and administration. He always guided us in all the difficult situations, such as kernel panics, disk crashes and other disasters. Hideo Saito always gave me coordial encouragement and support on my unskillful programs. He has been a great neighbor and a friend. His deep knowledge about Linux, English and trivial facts from Wikipedia were moral boosters at latenights. Tatsuya Shirai gave me much friendly and sensible advice, as a fellow and as an antecessor. His complaint about network and topology was informative and great motivation for my research. Even when my research does not go well, Takeshi Sekiya's laptop wallpaper, an image of cats, always relaxed me. His serious attitude for research was also impressive. Ken Hironaka's brilliant comments as well as his coffee were always stimulating. Not only in the field of computer science, he also helped me with English many times. I would like to show my gratitude again to those people and all the other members.

I thank my friends in Helsinki University of Technology, for my invaluable experience in Finland. I am also thankful for *Sodan* members, tutors in the educational computer system. I learned quite a few things from them, from Gentoo installation to *nabe*(hotpot) recipes. In addition, many thanks to my classmates in Komaba. I learned much from their attitude as a researcher.

Finally, I would like to thank everyone else for their support and encouragement. Thank you.

February 4, 2008

Contents

1 Introduction **1**

1.1 Background . . . 1

1.1.1 Distributed Environment . . . 1

1.1.2 Data Intensive Applications . . . 2

1.1.3 Data Transfer in Distributed Environments . . . 3

1.1.4 Broadcast Problem . . . 4

1.2 Contribution . . . 4

1.2.1 Stable Broadcast Algorithm . . . 5

1.2.2 UCP: Universal Broadcast Tool . . . 5

1.3 Organization of this Thesis . . . 6

2 Network Properties **7**

2.1 Network Devices . . . 7

2.2 Bandwidth and Latency . . . 7

2.3 Network Topology . . . 9

2.4 NAT and Firewall . . . 11

2.5 Network Measurement . . . 11

2.5.1 Topology Measurement . . . 12

2.5.2 Bandwidth Measurement . . . 12

3 Broadcast Algorithm **13**

3.1 Topology-unaware Broadcast . . . 13

3.2 Adaptive Broadcast . . . 14

3.3 Topology-aware Broadcast . . . 15

3.4 Multistream Broadcast . . . 16

4 Stable Broadcast Algorithm **18**
4.1 Stability and Optimality . . . 18
4.2 Our Broadcast Algorithm . . . 19
4.2.1 Basic Idea . . . 19
4.2.2 Construction of Transfer Trees . . . 21
4.2.3 Transfer Using Multiple Trees . . . 21
4.3 Stability and Optimality for Tree Topology . . . 24
4.3.1 Preparation: a Property from Symmetric Link . . . 24
4.3.2 Properties for Single Transfer . . . 25
4.3.3 Proof of Stability and Optimality . . . 25
4.4 Improvement in Graph Topology . . . 27

5 UCP - Universal Broadcast Tool **28**
5.1 Characteristics of ucp . . . 28
5.2 NATs and Firewalls . . . 29
5.2.1 Backward Connection . . . 29
5.2.2 Application-level Routing . . . 29
5.3 User Interface . . . 30
5.3.1 Process Startup . . . 30
5.3.2 Topology Specification . . . 31
5.3.3 Routing Specification . . . 31
5.3.4 Data Specification . . . 32

6 Evaluation **34**
6.1 Simulation . . . 34
6.2 Real Machine Experiment . . . 35

7 Conclusion and Future Work **39**
7.1 Conclusion . . . 39
7.2 Future Work . . . 39

List of Figures

1.1 Efficient Broadcast by Relaying 3
1.2 Traversing NAT by Relaying 4

2.1 Model of the Transmission Time 8
2.2 Model of the Transmission Time 9
2.3 Considering Topology 9
2.4 Switches 10
2.5 Machine and Link Throughput 10
2.6 NAT and Firewall 11

3.1 Flat Tree Broadcast 13
3.2 Pipeline Broadcast 14
3.3 Effect of Adding Nodes with Small Bandwidth 15
3.4 Tree Constructions in FPFR Algorithm 16

4.1 Stability and Optimality in Tree Topology 19
4.2 Counterexample in a Graph Topology 19
4.3 The Algorithm to Build Transfer Trees 22
4.4 Data Flow 23
4.5 Links in a Tree Symmetric Network 25

5.1 Backward Connection to Traverse NATs and Firewalls 29
5.2 Relaying to Traverse NATs and Firewalls 30
5.3 An Example Bandwidth-annotated Topology 31
5.4 Topology Expression 32
5.5 Routing Expression 32

6.1 The Real Environment Topology 36

6.2 The Relative Bandwidth to the *FlatTree* Algorithm in Simulations (Vertical axis: relative throughput to FlatTree) 37
6.3 Real Machine Experimets 38

List of Tables

6.1 Number of Nodes Used in the Experiments 35

Chapter 1

Introduction

1.1 Background

1.1.1 Distributed Environment

The spread of distributed environments has given a great impact on many scientific fields that require large amount of computations [13, 22]. While shared-memory parallel computers were traditionally used to achieve large computing resources, PC clusters, which consist of ordinary PCs connected to a fast network, has appeared to be a better alternative in many cases. They can achieve higher performance at the same cost as SMP machines, since their computational nodes as well as other network equipments are mass-produced,

Some computations require large memory spaces or computations. However, the size of one cluster is restricted by space, energy, heat and budget conditions. When clusters in different locations are connected together, larger memory space and computing resources are brought. Owing to the development of the wide-area high-throughput network, such distributed environment is becoming popular.

The main benefit of distributed computing is its vast computational resources. When 1000 nodes with 1 GFlops CPU and 2 GB memory are gathered, the environment potentially holds 1 TFlops processing power and 2TB memory. This environment enables large-scale computations, and the processing time of existing programs may also be dramatically shortened. Additionally, surpass computational resources are effectively utilized. If nodes are not used by local users, they are provided to other users who need them.

Some distributed computing projects are ongoing, such as TeraGrid [4] in the United States, *European DataGrid* [9] in European Union and *InTrigger* [26] project in Japan. For example, *TeraGrid* holds 750 teraflops of computing capability and more than 30 petabytes of online data storage connected with 40Gbps networks. This environment is used by thousands of researchers in the field of temperature simulations and gene analyses.

Although distributed computing has many advantages, it has some problems as well. One of the most important problems is heterogeneity, in terms of CPU performance, memory, storage and network speed. Since a distributed environment consists of various nodes, task assignments to nodes need to be carefully planned. We especially focus on network heterogeneity, which greatly affects data transfer time. A real network has a complicated topology, whose links have various latency and bandwidth values. Therefore, a transfer time of the same file greately affected by peer positions, relay planning and existence of other transfers. For example, when the bandwidth between two nodes varies from some Gbps to Mbps, a transfer in the latter case takes 1000 times as long time as in the former case.

This heterogeneity problem greatly affects some applications that need to synchronize all the processors. This applies to many traditional applications for high-performance computing, such as matrix simulations. While the performance of these applications is limited by the slowest node in the computation, a distributed environment possibly has nodes with high latency, nodes with slow CPU and nodes that lack bandwidth. As a result, they cannot achieve high performance without devising the algorithm. In addition, high latency also degrades the performance of these applications. Since the speed of data transmission cannot exceed the speed of light, latency is essentially high in a widely distributed environment.

1.1.2 Data Intensive Applications

Recently, high-performance computing greatly contributes to new fields of applications, such as data mining from the web, natural language processing and gene analyses. By processing vast amounts of data, these applications extract general knowledge, achieving great success. Especially, there are growing demands to analyze on the web. Since more data are accumulated on the web, larger storage, memory and computational power are required every day.

These applications are called data intensive applications, and the environment for data-intensive applications is called data grid [22]. Nowadays, data intensive applications are one of the most important applications executed on distributed environments.

Many data intensive application suits to be executed on the distributed environment, since they have high *locality*. An application is called to have high locality when the application can be divided into small parts that require only limited data. In these applications, data transfers do not frequently happen, so latency is not a large factor in the overall performance. When nodes are connected with sufficient bandwidth, every node in a distributed environment can be fully utilized.

While data intensive applications require high computational power, many of these applications can be split into small tasks that can be individually executed without communicating with each other. Such application can easily be executed with a task scheduler [16, 8, 2], which distributes serial tasks onto available nodes. A programmer only needs to write a serial program, and a task scheduler executes these tasks in parallel. Some task schedulers also care about data transfer as *staging* functions [14, 15, 19]. When a task needs large input data, the data are transferred in advance of the execution. The output data are transferred to a certain node as specified before. As different amount of tasks are assigned to different nodes according to its processing performance, the overall processing power is effectively utilized.

For example, Kawahara et al. [13] performed a large scale experiment in natural language processing. They used 350 nodes in a multi-cluster environment to perform tasks called case frame compilation

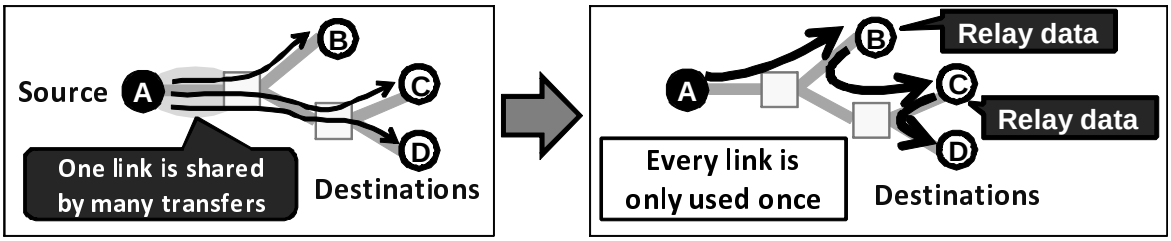


Figure 1.1: Efficient Broadcast by Relaying

to analyze each word in a sentence. While it takes 10 months to process with one node, they processed 500 million sentences in a day with those machines. Despite terabytes of data transfers, high scalability was achieved in this computation.

1.1.3 Data Transfer in Distributed Environments

Execution of data intensive applications requires a large-size data movement. Since data transfer occupies large part of the total execution time [3], data transfer plan largely affects the total processing time. However, data transfer planning is difficult because of several factors shown below:

Effects by link sharing with other transfers: When one link is shared by many large transfers, each transfer can only utilize part of the link bandwidth. In the worst case, when N transfers share one link, the throughput of each transfer is decreased to $1/N$. This situation easily happens on the outgoing link from a source node from which multiple nodes require data.

Existence of NATs and Firewalls: A distributed environment typically has some firewalls and NATs. While a firewall prevent attacks from the outside by filtering packets between local network and WAN by some rules, a NAT separates the local network from the internet to provide many private addresses to local nodes. As a result, a direct connection passing through NATs and Firewalls may not be possible.

These two problems are partly solved by using relaying technique. When many destination nodes require the same data from a source node, some of the destinations can relay data to the rest of the destinations in order to avoid the traffic concentration on the outgoing link from the source. Figure 1.1 illustrates an example that traffic concentration is avoided by efficiently relaying data. Three destination nodes B, C and D require the same data from the source node A and let every link have the same bandwidth. Every destination node receives data directly from A in the left picture, which causes traffic concentration on the outgoing link from A. In this case, the throughput of each link is limited to one third of the link bandwidth at most, because three transfers share one link.

In contrast, no traffic concentration occurs in the right picture of Figure 1.1. While only node B directly receives data from node A, C and D receives data from B and C, respectively. Since one link is only used once, transfer throughput arriving at each destination is the same as the link bandwidth, which is three times more than the previous case.

The latter problem, a connection through a NAT, is also solved by relaying. When a node has both private and global addresses, it can relay transfers traversing NAT. Figure 1.1(b) depicts an example.

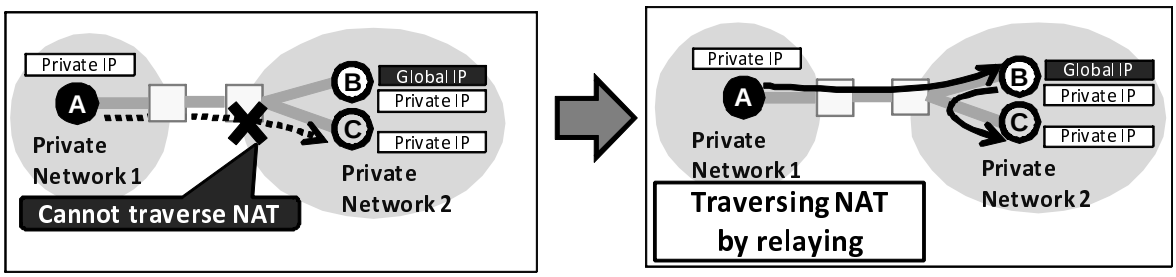


Figure 1.2: Traversing NAT by Relaying

Node A and C cannot establish a direct connection since they only have a private address in a different network, whereas node B with a global address can establish a connection to both A and C. When node B intermediate transfers, a connection can be established from A to C.

However, relay planning is a difficult problem especially in a large distributed environment, since it has a factorial search domain [27]. When there are 100 nodes concerned in a transfer, there are possibly 10^{157} kinds of relay plans.

1.1.4 Broadcast Problem

We especially focus on broadcast, one common, practical problem. In a broadcast, one or more *source* nodes have some data that need to be copied to some *destination* nodes. Various algorithms have been proposed to optimize broadcasts according to various criteria, such as completion time and aggregate bandwidth (the sum of the bandwidths of all nodes) [5, 6, 7, 11, 12, 17].

Such existing broadcast algorithms have a few shortcomings, especially for data intensive applications executed in distributed, wide-area environments. One is that they are not *stable*, in that the addition of a new node to an existing broadcast can potentially lower the bandwidth of a node already participating in the broadcast. This complicates task scheduling, because earlier scheduling decisions may have depended on a particular node receiving data at a particular rate. Another shortcoming of existing broadcast algorithms is that they are not *optimal* in terms of aggregate bandwidth. Even though many existing algorithms do optimize for aggregate bandwidth, they do so under the assumption that all nodes have the same incoming bandwidth. As a result, they do not explore the possibility of different nodes receiving data at different rates, missing the opportunity of having nodes that receive data early start computation without waiting for other nodes. In wide-area environments where link speeds can vary a great amount, it is especially important to prevent nodes with slow links from slowing down nodes with faster links.

1.2 Contribution

The contribution of our work consists of two points. First, we propose a stable broadcast algorithm that outperforms existing broadcast algorithms in terms of aggregate bandwidth. Second, we implemented a broadcast tool for distributed environments to demonstrate the usefulness of our algorithm.

1.2.1 Stable Broadcast Algorithm

We propose a broadcast algorithm that uses bandwidth-annotated topology information to optimize for aggregate bandwidth. Our algorithm improves an existing algorithm called FPFR [11], and performs broadcasts with higher aggregate bandwidth than FPFR for any graph topology. Moreover, for tree topologies, we prove that our algorithm satisfies the following two properties:

- *Stability:* Adding a node to an existing broadcast does not lower the incoming bandwidth of any node already participating in the broadcast.
- *Optimality:* A broadcast performed using our algorithm achieves the maximum possible aggregate incoming bandwidth.

While our algorithm works well with any graph topology, we use tree topologies for simplicity in the exposition. Additionally, broadcast planning time with our algorithm takes very short time: only 2 milliseconds for 400 nodes. To evaluate our algorithm, we used the topology inference tool developed by Shirai et al. [18] to obtain a tree topology, and performed both simulations and real-machine experiments.

1.2.2 UCP: Universal Broadcast Tool

Based on our broadcast algorithm, we propose a universal broadcast tool called *ucp* (Universal CoPy) for a distributed environment. It has three major characteristics:

Efficient Broadcast
: When multiple destinations are specified, it constructs multiple pipelines with our *stable broadcast* algorithm, and performs broadcast with them. In this broadcast algorithm, even if some nodes with small bandwidth join the broadcast, other nodes do not suffer from them. In a tree topology, our algorithm is proved to be optimal as well.

NAT/Firewall traversal
: When direct connections are restricted because of NATs and firewalls, *ucp* traverses them by backward connections and relaying. The ucp tool uses relaying technique only when it is needed, since relaying method has a larger overhead than the direct socket transfer.

High Usability
: We designed *ucp* for interactive use from a command line. Similar to *scp* command, it takes as arguments URI-like syntax to specify nodes and files. It uses a cluster middleware called *GXP* [21] to spawn processes in many nodes. With *GXP*, users can easily execute commands in remote nodes. As a result, a broadcast is performed by inputting a few commands.

Because of these characteristics, *ucp* can transfer data seamlessly in a distributed environment having NAT and firewalls. Its broadcast plan is efficient, and the aggregate bandwidth is maximized.

1.3 Organization of this Thesis

The rest of this thesis is organized as follows. In Chapter 2, we show a model of message transmission and network topology. Some work about network measurement is also introduced. In Chapter 3, we discuss work on previous broadcast techniques and data transfer tools. Our broadcast algorithm as well as proof of optimality and stability is shown in Chapter 4. Chapter 5 illustrates our broadcast tool *ucp*, on which our broadcast algorithm is implemented. The Chapter 6 depicts the experiments and evaluation of our algorithm in the real environment. Finally, we conclude the thesis in Chapter 7.

Chapter 2

Network Properties

A message transmission time in a network is affected by many factors. In this chapter, we show some factors concerned to a message transmission and introduce some existing network measurement techniques. Section 2.1 shows basic assumtions of our discussion, and Section 2.2 shows a message transmission model. Our model of network topology is described in Section 2.3, and Section 2.4 discusses about NATs and firewalls. Finally, existing network measurement techniques are introduced in Section 2.5.

2.1 Network Devices

A network physically consists of computational nodes, switches, routers and intermediate links. Typically these devices use Ethernet and TCP/IP protocol, and the network is connected to the internet. Each device has some limitations with data procession: a device can process only a certain amount of data at a time, and the procession takes some time. When the communication is performed within a small local environment, only a few devices are concerned a transfer. However, in a wide-area network, quite a few devices intermediate the transfer.

Among many network layers, we will discuss the application layer, in which each node is identified with an IP address or hostname. Since routing is automatically planned, a message reaches the destination through multiple devices. While some IP addresses are unique in the internet, some are only valid in a local environment. The former class of addresses are called *global address*, and the latter is called *private address*.

2.2 Bandwidth and Latency

Transmission time of a message is dominated by many network properties. Although the real network behavior is complicated, a simplified model well explains real network behaviors. In the model, two factors of a network are concerned in message transmission time.

Bandwidth: The amount of data passing through a link in a degree of time. Typical link speed in Ethernet is 1Gbps. Although the maximum bandwidth, or capacity, is unique to each link, the available bandwidth changes when other transfers simultaneously occur on the link. When several transfers share a link, the total of their bandwidth is limited by the link capacity.

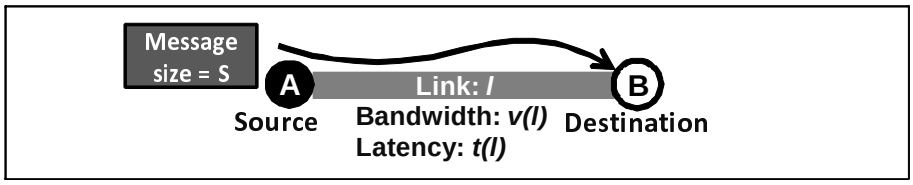


Figure 2.1: Model of the Transmission Time

When two nodes are connected with two links in parallel, bandwidth is calculated by the sum of that of each link. When links are connected in a chain, a transfer can utilize only the smallest bandwidth among them. Let $v(l)$ denote the bandwidth of link l.

Latency : Time that a packet of data reaches from one node to another. It contains physical signal transmission and overhead on both end nodes. When a small packet is sent from one node to another, the message is copied to special memory, processed by network devices and transmit through a physical link. Although many factors play a role, the total of this delay is unique to a link and devices and nearly independent to message size.

A latency value varies from nanoseconds to hundreds of milliseconds. A long-distance link inevitably has a large latency value, since physical signal transmission takes long time. Even when the signal is conveyed in the speed of light, it takes 30 milliseconds to transmit 1000 km.

When several links are serially connected, latencies are simply accumulated. Let $t(l)$ denote the latency of link l.

Figure 2.1 shows an example that two nodes are connected with one link l, whose bandwidth and latency are $v(l)$ and $t(l)$, respectively. Transmission time T of a message with size S satisfies the following equation:

$$T = t(l) + \frac{S}{v(l)} \tag{2.1}$$

While the former term $t(l)$ is independent of the message size, the latter term $\frac{S}{v(l)}$ is proportional to the message size. Therefore, although latency is dominant for short messages, bandwidth is more important for long messages. In particular, since our focus is on very large messages, we only need to consider the latter term.

For example, if a link has 1Gbps bandwidth and 1 millisecond latency, the bandwidth term is greater than the latency term for $S >$ 1MB. Since our focus is on long messages, we can assume S is greater than 100MB. In this case, the latency term is less than one percent in the total transmission time, being virtually negligible.

In a real situation, usually several switches stand between the source and destination nodes. Figure 2.2 illustrates an example that node A and B are connected with three links. Let $\{L_k | 0 \leq k \leq 2\}$ denote the three links. Link l_k has v_k bandwidth and l_k latency. The transmission time is expressed in the following equation:

$$T = \sum_{0 \leq k < 3} t(l_k) + \frac{S}{min(v_0, v_1, v_2))} \tag{2.2}$$

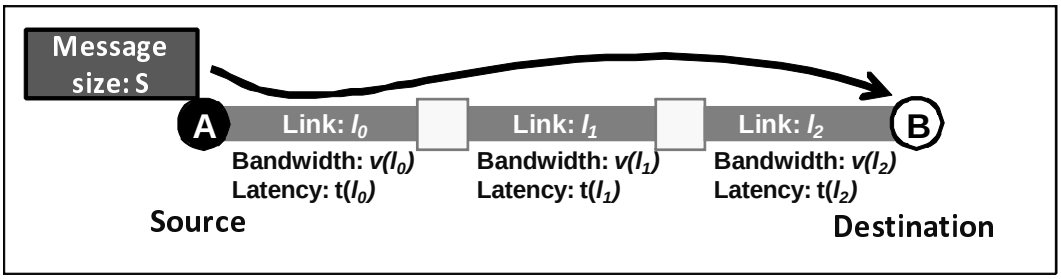


Figure 2.2: Model of the Transmission Time

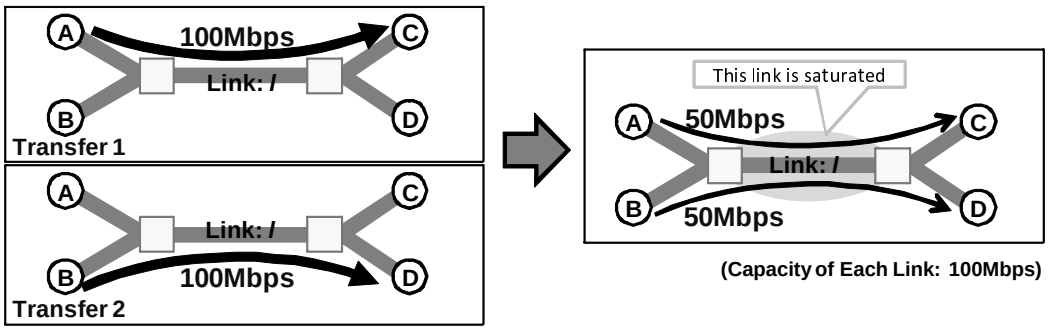


Figure 2.3: Considering Topology

It shows that only the narrowest link bandwidth in the route matters in the total transmission time for a long message broadcast. When three links with 10Mbps, 100Mbps and 1Gbps bandwidth are connected in a line, the overall bandwidth is limited to 10Mbps.

2.3 Network Topology

While a link bandwidth has a physical limitation, sharing a link by multiple transfers also causes decay in transfer throughput. Figure 2.3 illustrates an example that two transfers, $T1$ from A to C and $T2$ from B to D, occur in parallel. Let every link capacity be 100Mbps. When only $T1$ exists, it achieves 100Mbps, so as $T2$. However, when they simultaneously occur, their throughput drop to 50Mbps each because of the saturation of link l.

As a result, it is inevitable to consider link sharing during transfer planning in order to estimate available bandwidth. Link sharing is detected by using topology information, a connection map of nodes and switches. Although wide-area network contains many intermediate devices such as routers and switches, we use a simplified topology which only contains computational nodes and switches with branches. Our topology consists of the following three elements:

Link: A link in our model corresponds to one or more serially-connected physical links. It may also contain intermediate devices, such as routers and switches without branches. Its bandwidth shows the minimum throughput of these links and devices.

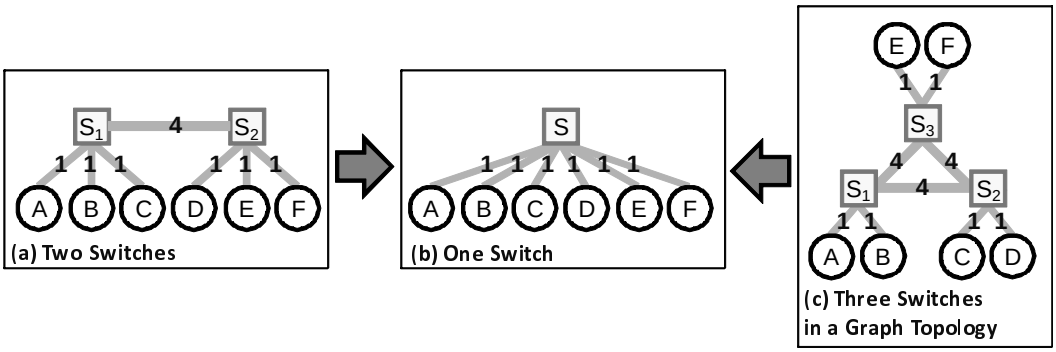


Figure 2.4: Switches

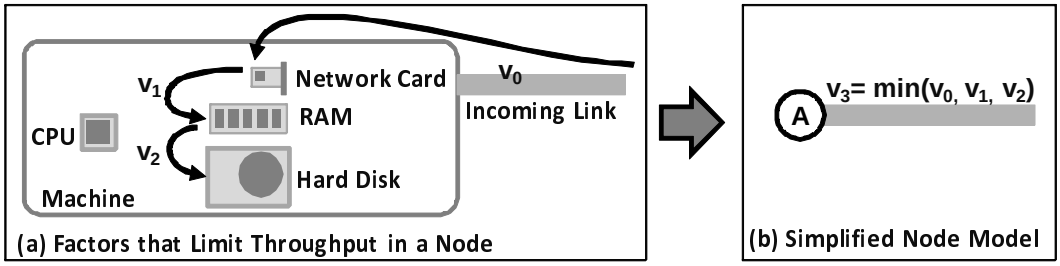


Figure 2.5: Machine and Link Throughput

Switch: A switch in our model corresponds to one or more physical switches. Although a switch has some processing throughput limitation called *backplane*, it does not affect transfer throughput when the backplane is larger than the total amount of all the passing traffic. When a switch has N number of ports, and each port has throughput value v, the traffic passing through this switch is $(v \cdot N \cdot (N-1))$ at most. Thus, the switch never limit any traffic passing through it if its backplane is larger than $(v \cdot N \cdot (N-1))$. We assume this condition is satisfied in most cases.

Similarly, switches connected with enough bandwidth link can be treated as one switch. Figure 2.4 illustrates an example with 6 nodes and some switches. In Figure 2.4(a), two switches S_1 and S_2 is connected with link bandwidth 4. When we only consider bandwidth, this topology can be simplified to Figure 2.4(b), since the link between S_1 and S_2 is never saturated. Similarly, Figure 2.4(c) can also be simplified to Figure 2.4(b), since links among S_1, S_2 and S_3 are never be saturated. Note that in the latter case, a graph topology is simplified to a tree topology.

Computational Node: A node in our model corresponds to one computational node. Although data throughput in a node is limited by its processing power and disk access performances, these throughput limitation can be merged to that of the outgoing link from the node. Figure 2.5 illustrates an example, where data coming from a link is written to the disk. While the throughput is limited by one of v_0, v_1 and v_2 in (a), the node and link are simplified in (b), where the link has the bandwidth value $min(v_0, v_1, v_2)$.

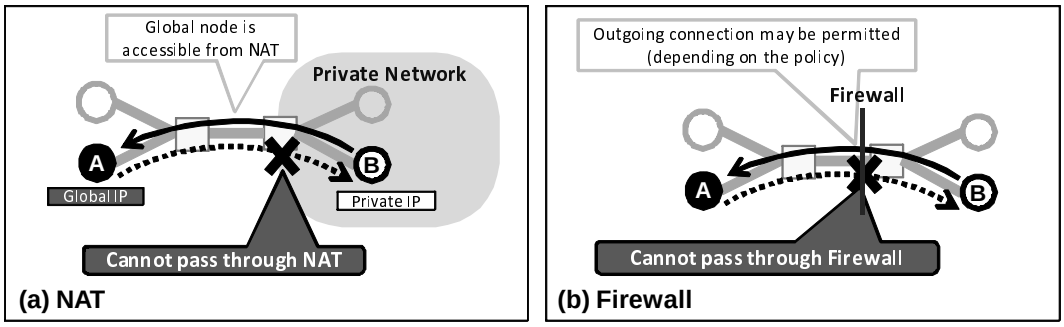


Figure 2.6: NAT and Firewall

2.4 NAT and Firewall

A distributed environment typically contains some NATs and firewalls. Although a NAT and a firewall have different functions, their behaviors are similar.

A NAT intermediates two networks, typically global and private. Since it converts global and private addresses, a node only having a private address can establish a connection to a global node. It is especially useful when the number of global addresses is limited.

While NATs have many advantages, some connections passing a NAT is blocked. Figure 2.6(a) illustrates an example of two nodes and one NAT. While node A has a global address, node B behind a NAT only has a private address. As a result, although B can connect to A by specifying the global address of B, connection from A to B is not possible. Generally, a node cannot establish a connection to a node only having a private address in a different network.

On the other hand, a firewall is installed between a local network and a global network to prevent nodes from unsolicited network accesses. It does so by only permitting incoming and outgoing packets that match against specified rules, and drops any other packets. Although various policies are possible, one typical setting is as follows:

- Permit any outgoing connections
- Permit packets of already established connections

Figure 2.6(b) illustrates an example that one of two global nodes is behind a firewall, which has the above mentioned policy. While a connection from A to B is denied, that from B to A is allowed.

2.5 Network Measurement

As we have seen in Section 2.3, network topology information and link bandwidth are important factors for efficient data transfers. Recently, much research has done in the field of dynamic network measurement. This section introduces some work to infer topology and to measure bandwidth in a large-scale network.

2.5.1 Topology Measurement

Although topology information is useful for efficient transfer planning, it is not easy, or sometimes impossible, to physically trace a topology of a distributed environment. Unlike in the case of a small local network, nodes and switches are located in various places. In addition, since network topology frequently changes in a WAN environment, dynamic topology measure is inevitable.

One approach is directly collecting information from routers by using administrative protocols, such as SNMP. Since routers provide information about mappings of nodes and ports, a complete topology is obtained by gathering and analyzing this information. While exact information is obtained by this approach, administrative privileges are required to perform it. This requirement is virtually impossible in a distributed environment, where various networks are connected in it.

Shirai et al. [18] introduced a method to estimate network topology by only using network latency information among two nodes. In this approach, a topology is gradually constructed by adding a node to a certain branch. Although the software can only work in a tree topology, it can be executed without administrative privileges. The estimation is significantly fast by reducing slow and redundant measurements: they reported that it took 27 seconds for 256 nodes in 4 clusters. This topology inference tool is used in our broadcast planning.

2.5.2 Bandwidth Measurement

Bandwidth between two nodes is induced from Equation 2.1. After sending a large message, bandwidth is calculated by dividing message size by elapsed time. *Iperf [1]* is one of the most widespread tools using this method. While they can measure exact available bandwidth, they affect other ongoing transfers in the network because of a large number of packets.

Some other tools, such as *Pathchar*, *NetTracer* [25] and *Spruce* [10], measure bandwidth from much smaller amount of packets than *Iperf*. They model the behavior of each packet to estimate bandwidth [20]. For example, when two packets are sent in a rapid sequence, the latter packet reaches the destination delays to the former packet because of simultaneous traffics. By measuring this delay, available bandwidth is inferred. Unfortunately, when we tested these tools in a real environment, none has performed satisfying result. Since these techniques are still in progress, bandwidth measurement becomes easier soon.

Although above mentioned tools only measure bandwidth among two peers, bandwidth of each link in a topology can also be inferred by combining many measurements. Naganuma et al. [28] has invented a tool that dynamically measures each link bandwidth of a topology in short time. This tool is used in our implementation of the broadcast algorithm.

Chapter 3

Broadcast Algorithm

In this chapter, we show previous broadcast algorithms for large messages. We classify them in two criteria: one is topology unaware and aware, the other is single-stream and multi-stream. Although much research has done to improve aggregate bandwidth, none has been evaluated in terms of stability.

3.1 Topology-unaware Broadcast

The simplest broadcast algorithm is *flat-tree*, in which the source node directly sends data to all the destinations. Figure 3.1 describes an example of flat-tree broadcast, where (a) shows a graph topology with one source S and four destinations $D_0 \ldots D_3$. The bandwidth value of each link is labeled on it. In Figure 3.1(b), direct connections between the source and all the destination are established. Since the outgoing link from S is saturated and the aggregate bandwidth, the total amount of incoming bandwidth in each destination, is limited to 10.

This method is easy to implement, and its performance does not matter for short message size and a small number of nodes. However, for long message broadcast to a large number of nodes, this simple algorithm causes serious performance degradation. Since the outgoing link from the source is shared by all the transfers, it takes proportional time to the number of destinations.

FastReplica [5] improve the performance by avoiding traffic concentration on the outgoing link from the source. First, the source splits data into small pieces and sends each piece to a different destination.

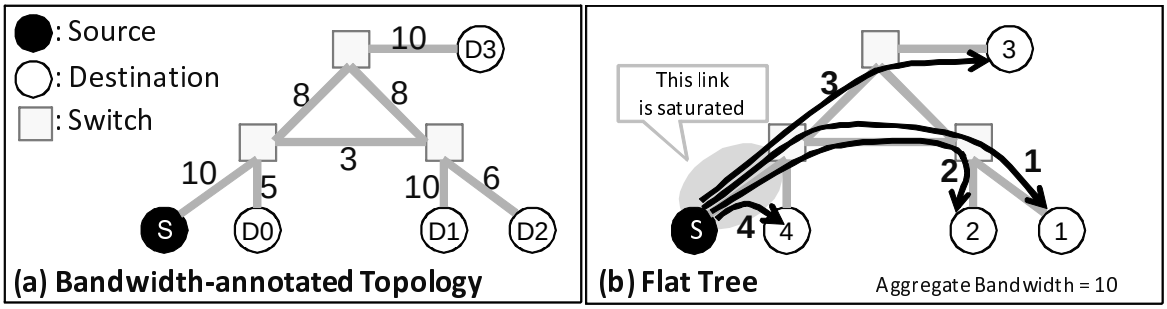


Figure 3.1: Flat Tree Broadcast

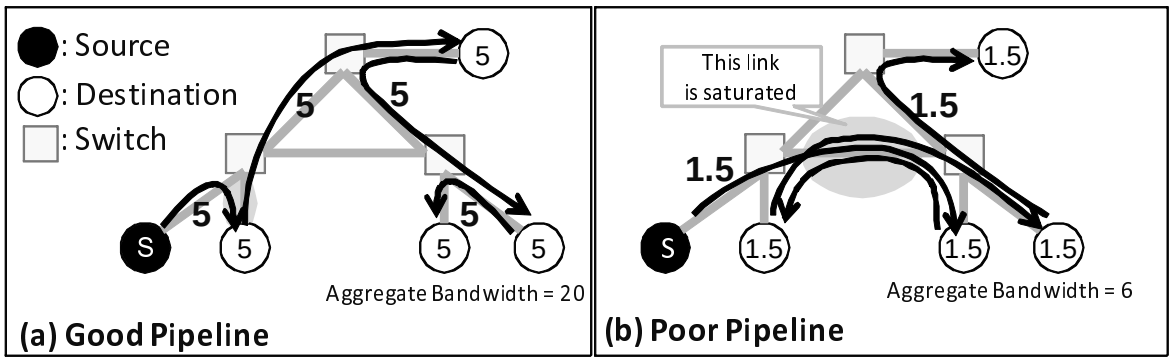


Figure 3.2: Pipeline Broadcast

Second, the destinations construct a ring connection to exchange pieces, and finally every destination obtains the entire file. Since all the data pass through the outgoing link from the source only once, its aggregate bandwidth is improved than in the case of *flat-tree*.

However, since the method does not consider network topologies, traffic concentration may occur not in the outgoing link from the source, but in other links in the ring connection. Even only two links share a link, each transfer bandwidth is halved.

Pipeline transfer is another well-known broadcast technique. In this method, data are sent in a line from the source to every destination. Since each destination relays data to only one other node, each node is possibly perform maximum possible bandwidth when every link has the same capacity.

However, the link sharing problem also occurs on this method. Figure 3.2 illustrates two examples of pipeline broadcast. Picture (a) successfully avoids link sharing and achieves large aggregate bandwidth. On the other hand, in (b), one link used many times degrades the overall performance. To construct a well-performing pipeline, topology information need to be considered.

3.2 Adaptive Broadcast

BitTorrent [17, 24] and some other work [6] adaptively improve the transfer graph, by dynamically changing the relay network.

In *BitTorrent*, data are divided into small pieces called *chunks*. The size of a chunk is typically set to 64KB to 1MB. Each node is allowed to receive chunks in an arbitrary order, and a transfer finishes when it has received all the chunks. In addition, all the nodes including source and destinations construct a transfer group.

When two nodes establish a connection, they exchange already received chunks each other. A node that has just joined a transfer chooses 5 nodes from the transfer group and establishes connections with them. Since the joined node does not have any chunks, it only receives chunks in the first stage. A connection may be refused when the peer already has many connections. In that case, it chooses another peer and try to establish a connection.

The connections are periodically changed to improve the overall throughput. A connection between two nodes is closed 30 seconds after the establishment. Then, they randomly select new peers, establish

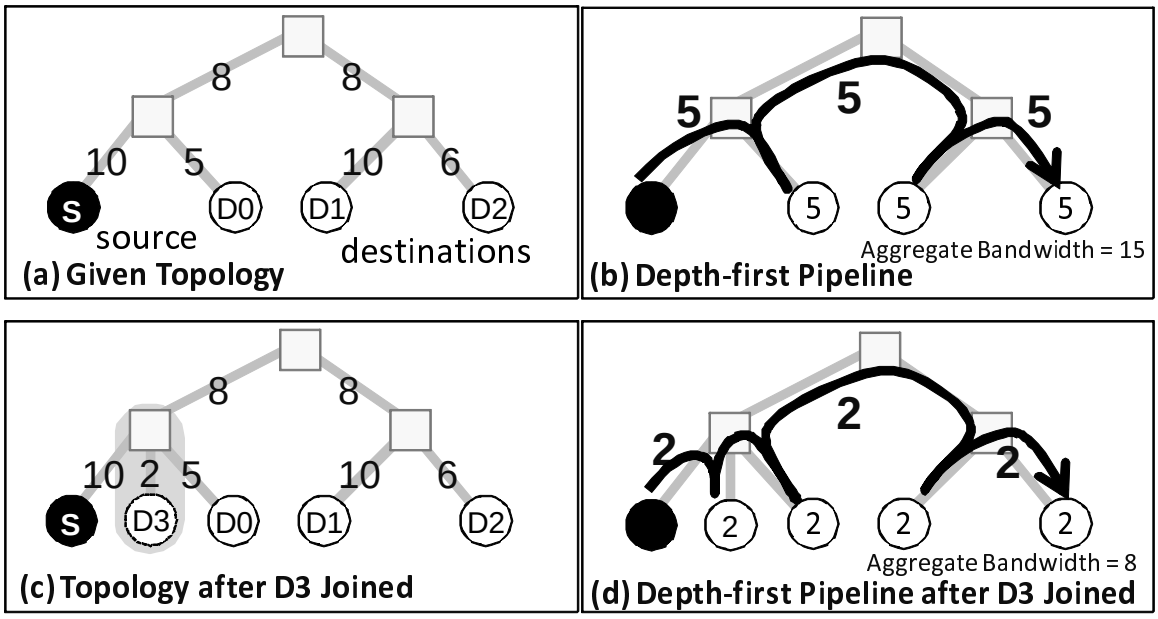


Figure 3.3: Effect of Adding Nodes with Small Bandwidth

a connection and again start exchanging between newly-selected peers. As mentioned before, a node can only accept 5 connections at a time. When it receives more than 5 connection requests, it chooses 5 nodes in the order of *download rate*, the bandwidth it has offered to other nodes. Since a node offering more *download rate* is likely to receive more amounts of data, their bandwidth is not degraded by nodes with small bandwidth. As a result, the aggregate bandwidth is improved.

However, these algorithms cannot always reach a schedule that avoids the link sharing. In addition, it takes time to reach a better schedule since these methods improve the transfer tree by try-and-error.

3.3 Topology-aware Broadcast

Some methods consider network topology information to avoid link sharing and to obtain a high-throughput broadcast schedule. Karonis et al. [12] proposed to use a hierarchical tree that describes the real network topology. A broadcast is performed by tracing the tree: a node relays data to all its children. Since a link is only used once for the same data, it can avoid self-induced congestions. This algorithm well works for a small message broadcast. For long messages, however, the aggregate bandwidth decreases since the tree has many branches. If a node has N children, the bandwidth of lower-level nodes is limited to $1/N$.

In a topology-aware pipeline transfer [18, 23], each node relays data to only one other node. By tracing a topology in a depth-first manner, a pipeline is obtained. In this pipeline, each directional link is used only once, so the throughput is limited by the narrowest link in it. If the bandwidth of every link is the same, every node can receive the same amount of data as the link bandwidth.

However, the depth-first pipeline cannot achieve good performance in terms of aggregate bandwidth when some nodes lack bandwidth. Figure 3.3 illustrates a situation in which new node D_3 with

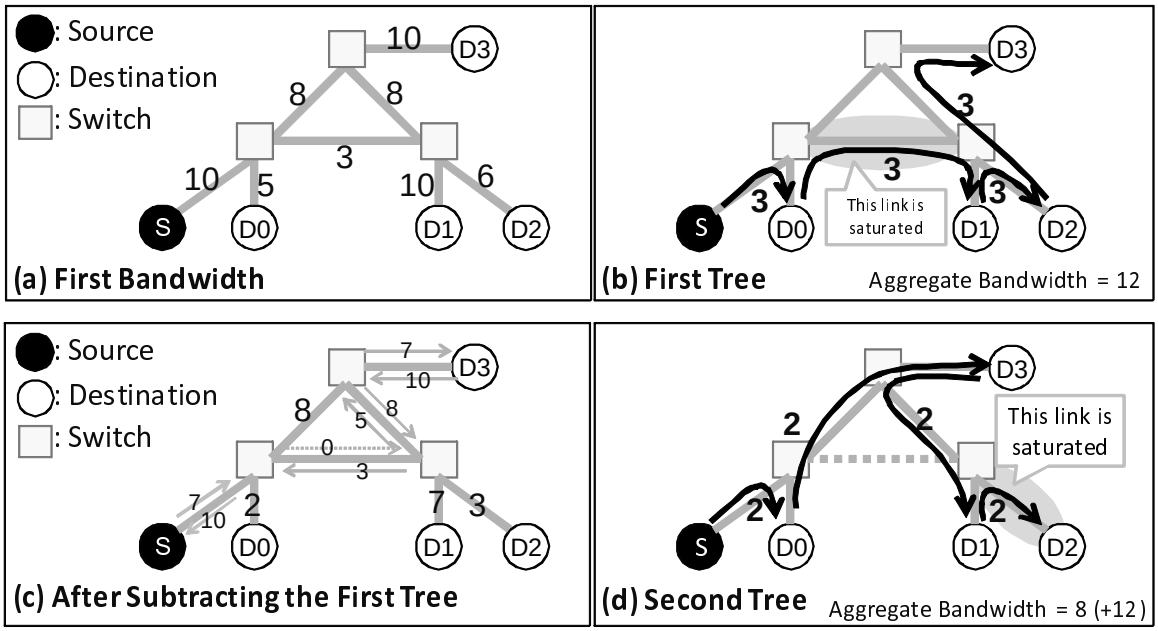


Figure 3.4: Tree Constructions in FPFR Algorithm

small bandwidth joins the broadcast. In (a), one source S holds the original data, three destinations D_0, D_1, D_2 need them, where bidirectional bandwidth is labeled on each link. Picture (b) shows the original broadcast in which the aggregate bandwidth of D_0, D_1 and D_2 is 15. Then, a new node D_3 joined the broadcast in (c). As a result, in (d), the aggregate bandwidth has dropped to 8, even it is for 4 nodes. We call this situation *lack of stability*.

3.4 Multistream Broadcast

The Fast Parallel File Replication (FPFR) tool [11] uses bandwidth-annotated network topology information to perform efficient broadcasts. It iteratively constructs multiple spanning trees, and concurrently uses them. By utilizing available link bandwidth more effectively, FPFR achieves higher aggregate bandwidth than methods that only use a single spanning tree.

Let the network topology and available bandwidth of each link be given in advance. The first tree is built based on the original bandwidth values. They have tried several ways to construct spanning trees: *depth-first*, *breadth-first* and *Dijkstra*. Among them, *depth-first* method, which traces the destinations from the source in a depth-first manner, achieved the best performance. After the first tree was formed, the bandwidth value for the tree is subtracted from the original bandwidth of all the links. The following trees are repeatedly built with the subtracted bandwidth values until no spanning tree can be created.

The details to build a tree are as follows. The *depth-first* method traces the destinations from the source in a depth-first manner. The only difference in *breadth-first* method is that destinations are traced in breadth-first style. In the *Dijkstra* method, the source and destinations already connected

to the source form the *source set*. A destination that can be reached from the source set in the largest bandwidth are picked and put into the source set. This operation is continued until every destination is connected to the source.

These multiple spanning trees are concurrently used with the planned bandwidth. Each tree sends different pieces of the file, and finally every destination receives the whole file.

Figure 3.4 depicts an example of planning of FPFR broadcast. In this example, the initial bandwidth is given in (a). The first tree traces all the destinations from S in a depth-first manner, resulting a tree (b). The bandwidth of the tree (b) is 3, which is determined by the narrowest edge in it. After the first tree has obtained, the bandwidth of the first tree is subtracted from all the links used by the first tree. It is shown in (c). For the bandwidth-annotated topology in (c), the second tree is similary constructed in a depth-first manner. As a result, the tree described in (d) is obtained. After constructing these two trees, no more trees can be constructed since the incoming link to $D0$ is saturated.

While *FPFR* has achieved better performance than methods that use only one tree, *Balanced Multicasting* [7] has further improved FPFR by using linear programming to maximize the aggregate bandwidth. In *FPFR*, the bandwidth assigned to each tree is decided in a greedy manner: the first tree takes as much bandwidth as possible, and the following trees can only use the rest of the bandwidth. Although *Balanced Multicasting* also uses the same set of trees as in *FPFR*, it maximizes the aggregate bandwidth by applying linear programming to the bandwidth of these trees. As a result, the aggregate bandwidth is improved.

However, same as single pipeline transfer, FPFR and Balanced Multicasting has problems with stability. Since they only use spanning trees, a node with small bandwidth limits the aggregate bandwidth of the other nodes. Since the throughput of a pipeline is limited by the narrowest link in it, one node that lacks bandwidth limits incoming bandwidth of all the nodes.

Chapter 4

Stable Broadcast Algorithm

In this chapter, we show an algorithm to distribute a large message to many hosts in a heterogeneous network. Similar to *FPFR* algorithm, it constructs multiple pipelines and uses them simultaneously. Section 4.1 introduces criteria *stability* and *optimality*, and the detail of our broadcast algorithm is explained in Section 4.2. Section 4.3 discusses the stability and optimality in a tree topology. Finally, Section 4.4 shows the improvement in a common graph topology.

4.1 Stability and Optimality

In the execution of data-intensive applications, each node is desired to have as much incoming bandwidth as possible. However, in most algorithm mentioned in Chapter 3, incoming bandwidth of nodes already in a broadcast is degraded by adding a node that lacks bandwidth.

To evaluate this situation, we propose a notion of *stability*. A broadcast algorithm is *stable* when adding more nodes to destinations never decreases bandwidths delivered to the original nodes. Formally, a stable broadcast satisfies the following condition: for any pair of destination sets D and E such that $D \subset E$, and for any node $n \in D$, the bandwidth to n generated with destinations E is never smaller than that generated with designations D.

In addition, we introduce another criterion *optimality*. A broadcast is *optimal* when it achieves the maximum possible aggregate incoming bandwidth. While a broadcast algorithm is commonly evaluated by the aggregate bandwidth, an optimal algorithm maximizes the aggregate bandwidth.

Figure 4.1 illustrates an exmple that stability and optimality are achieved in our broadcast algorithm, which is described in the following sections. The topology and bandwidth for each link is shown in (a). Picture (b) illustrates an optimal transfer for node $D2$, where the incoming bandwidth of node $D2$ is 8. In other words, node $D2$ never receives more bandwidth than 8 in any cases. On the other hand, picture (c) depicts a broadcast plan obtained by our algorithm, in which node $D2$ can also receives the same bandwidth 8, optimality and stability holds for $D2$. Since these properties are similarly satisfied for all the nodes, we can conclude this broadcast plan is stable and optimal.

In a common graph structure, however, no algorithm can achieve both stability and optimality at the same time. Figure 4.2 shows one counter example. Picture (b) shows that node A possibly receives 6 bandwidth by receiving both from S and B, where B relays data from S. Similarly, the incoming bandwidth of node C reaches 6 in (c). However, in a broadcast to both A and C, both nodes

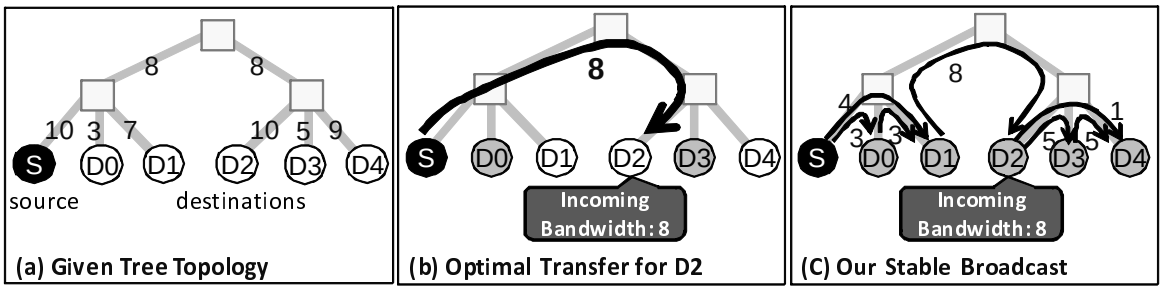


Figure 4.1: Stability and Optimality in Tree Topology

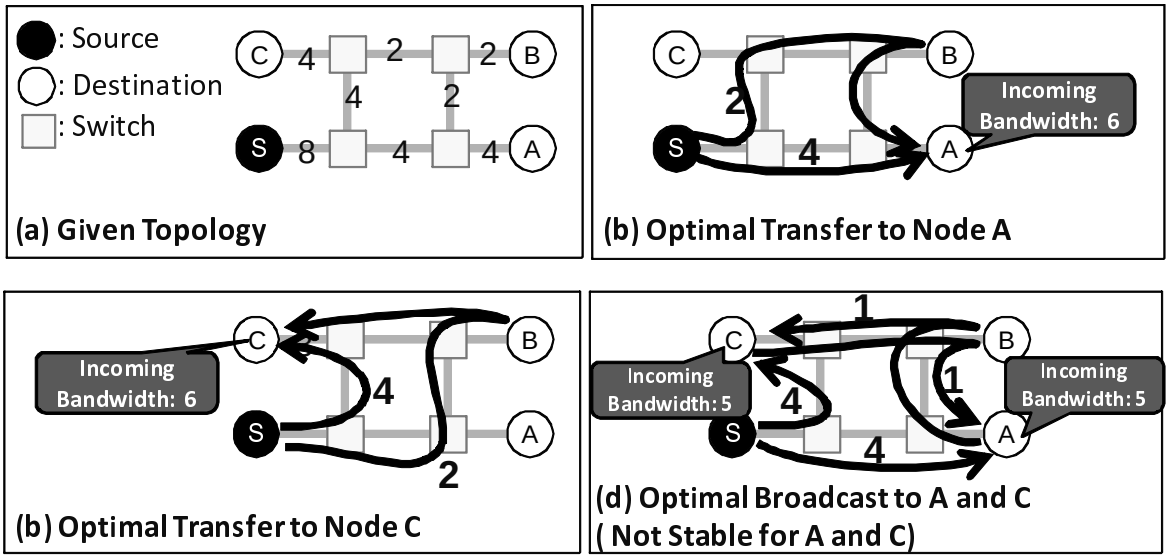


Figure 4.2: Counterexample in a Graph Topology

cannot simultaneously achieve the bandwidth 6. As a result, no algorithm can achieve stability and optimality in this graph topology.

4.2 Our Broadcast Algorithm

In this section, our broadcast algorithm is explained. The basic idea of this algorithm is shown in Subsection 4.2.1, and Subsection 4.2.2 describes the detailed algorithm to construct trees. The actual transfer with these trees are shown in Subsection 4.2.3.

4.2.1 Basic Idea

Like many broadcast algorithms referred in Chapter 3, our algorithm takes as input a bandwidth-annotated network topology, a single source node, and multiple destination nodes, to generate a set of *transfer trees* each of which is labeled with its throughput (the bandwidth it consumes). In this

model, a set of transfer trees is said *feasible* when the total throughput used on each link is within its capacity. In practical terms, we model the problem assuming that each switch has a sufficiently strong backplane, so the transfer rate is limited only by capacities of hosts or individual ports of switches, not by internal switching capacities.

We propose a broadcast algorithm which achieves more aggregate bandwidth than FPFR and Balanced Multicast. Besides, the algorithm is shown to be stable and optimal with respect to aggregate bandwidth when the network satisfies the following conditions.

1. The topology is a tree.

2. Each link is bidirectional and has a symmetric capacity in both directions.

The basic idea of the algorithm is similar to FPFR and the balanced multicast, in that it builds multiple transfer trees and sends data fragments through them. The main difference is that while FPFR stops building transfer trees as soon as the narrowest link saturates, our algorithm continues to make *partial trees* that involve only a subset of the destinations. Such trees play an important role to guarantee that our algorithm is stable, or that nodes connected to the source with higher bandwidth will receive data with their highest bandwidths.

Transfer scheduling becomes slightly more complex than FPFR to guarantee that all nodes receive all data despite that some transfer trees only contain a subset of the destinations. Its details are shown in Subsection 4.2.3.

Algorithm 1 Constructing Transfer Trees

Require: $\mathcal{T}$ is a network topology. $B_0(e)$ is the bandwidth for link e. s is the source node and D_0 is a set of destinations.
  $B := B_0$; $T := \varnothing$;
  while TRUE **do**
    $t :=$ the tree obtained by tracing destinations in depth-first manner from s;
    if t contains no destination **then**
      break
    end if
    $u :=$ the bandwidth of the narrowest link in t;
    $T := T \cup \{\langle t, u\rangle\}$;
    for each link e in t **do**
      $B(e) := B(e) - u$;
    end for
  end while
  return T

Algorithm 2 Sending Data via Transfer Trees

Require: $t_1, \ldots, t_n$ are the transfer trees returned by the construction algorithm (t_i is made by the i-th iteration). u_i is the throughput of t_i. D is the data to broadcast.
  for $k := n$ downto 1 **do**
    $R :=$ data nodes in t_k have not yet received;
    send R through $t_1, \ldots, t_k$, allocating to t_i the amount of data proportional to u_i;
  end for

4.2.2 Construction of Transfer Trees

The algorithm to create transfer trees is shown in Algorithm 1. The network topology $\mathcal{T}$ is given in advance as a directed graph. Computation nodes and switches are expressed with its leaf and intermediate nodes, respectively.

Each link is bidirectional and modeled as two separate edges of the graph. In addition, the bandwidth (capacity) $B_0(e)$ for each link e is given.

Like FPFR, the algorithm repeats building transfer trees. The first tree is constructed by visiting all destinations from the source node in a depth-first manner. As a result, it creates a tree that connects the source and every destination. The throughput of this tree is set to the capacity of the narrowest link in it.

After obtaining the first tree, bandwidth of each link available for subsequent trees is calculated by subtracting the throughput allocated to the first tree. Edges whose capacities become zero are removed. Note that at least one edge is removed. The second tree is constructed with this reduced bandwidth map using the same depth-first traversal. Note however that in this time, it may not be possible to reach every destination from the source as some edges have been removed. If that happens we connect only destinations reachable from the source. Note that if the network is a tree, in particular, at least one node becomes unreachable. The throughput of the second tree is again set to the capacity of its narrowest link, and is then subtracted to each link used by the tree. We repeat this until no nodes become reachable from the source.

Figure 4.3 shows the tree construction process in our algorithm. The overall topology as well as bandwidth of each link is described in (a), where four destinations $D_0 ... D_3$ need data from one source S. Based on this topology, the first tree is constructed in a depth-first traversal, which is described in (b). Bandwidth 3 is reserved for this tree, and it is subtracted from the original tree, as in (c). Next, the second tree is constructed in the same manner. These two trees trace all the destinations. In (e), however, $D0$ cannot receive more data since the incoming link to D_0 is saturated. As a result, the third tree, described in (f), only contains part of the destinations: D_1, D_2 and D_3. The fourth tree is similarly obtained. After obtaining four trees, no more data can flow from S in (i), so the tree construction terminates. After all, four trees are obtained, and these trees are simultaneously used, as in (j).

The idea introduced in Balanced Multicasting may also be applied for this in the graph case: the bandwidth may be improved by applying linear programming for the four trees, to maximize the aggregate bandwidth. However, in the tree case, the trees and their bandwidth obtained by this procedure is always optimal. It is proved in Section 4.3.

This algorithm can also be applied to multi-source broadcast. For each source, the algorithm shown above is performed. After some trees originated from a source are constructed, the bandwidth used in those trees is subtracted. With this available bandwidth, trees starting from the other source are structured.

4.2.3 Transfer Using Multiple Trees

Once the transfer trees have been constructed, we send data fragments with them. To achieve the claimed stability and optimality, all trees must be effectively utilized. Algorithm 2 shows how the transfer is done.

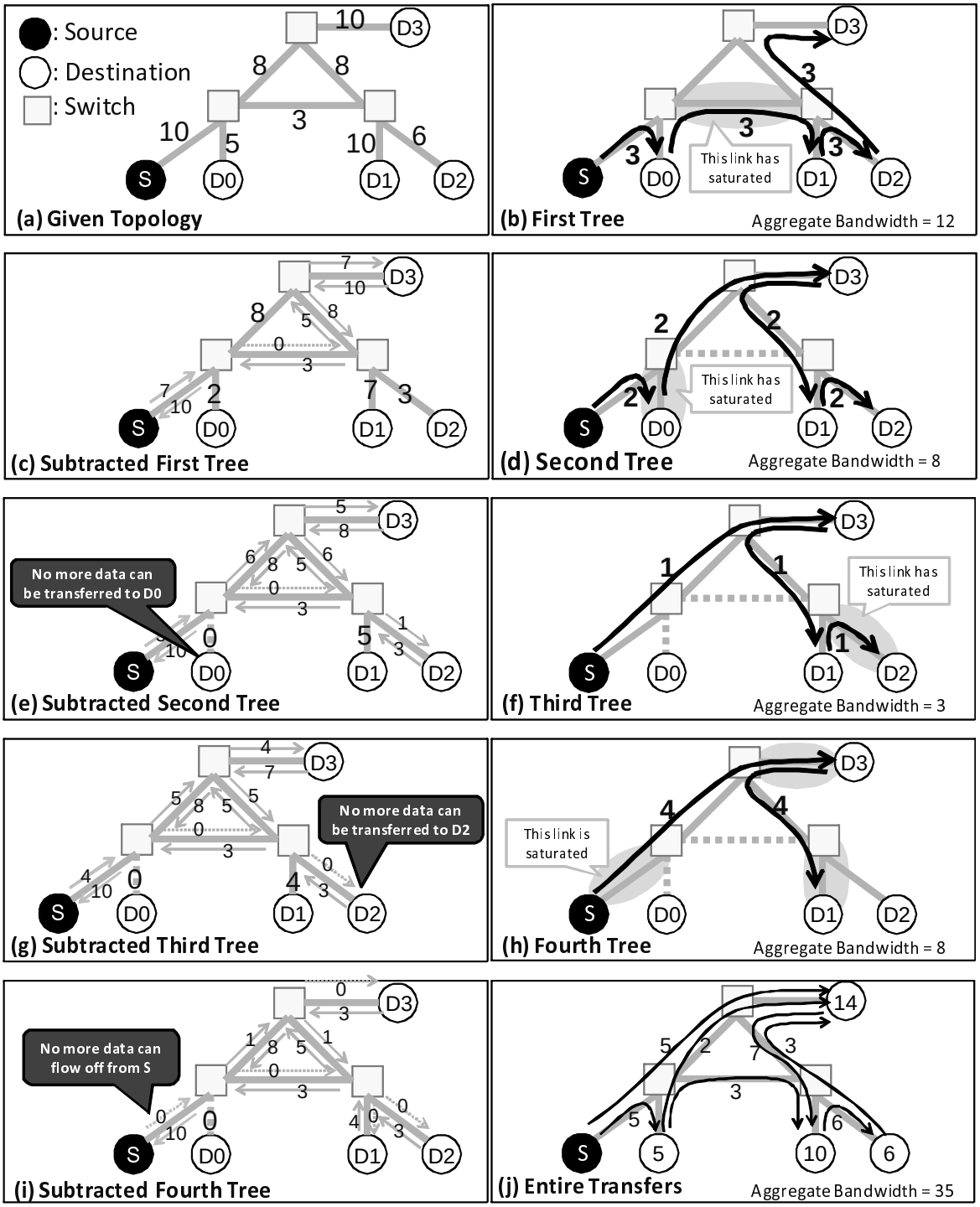


Figure 4.3: The Algorithm to Build Transfer Trees

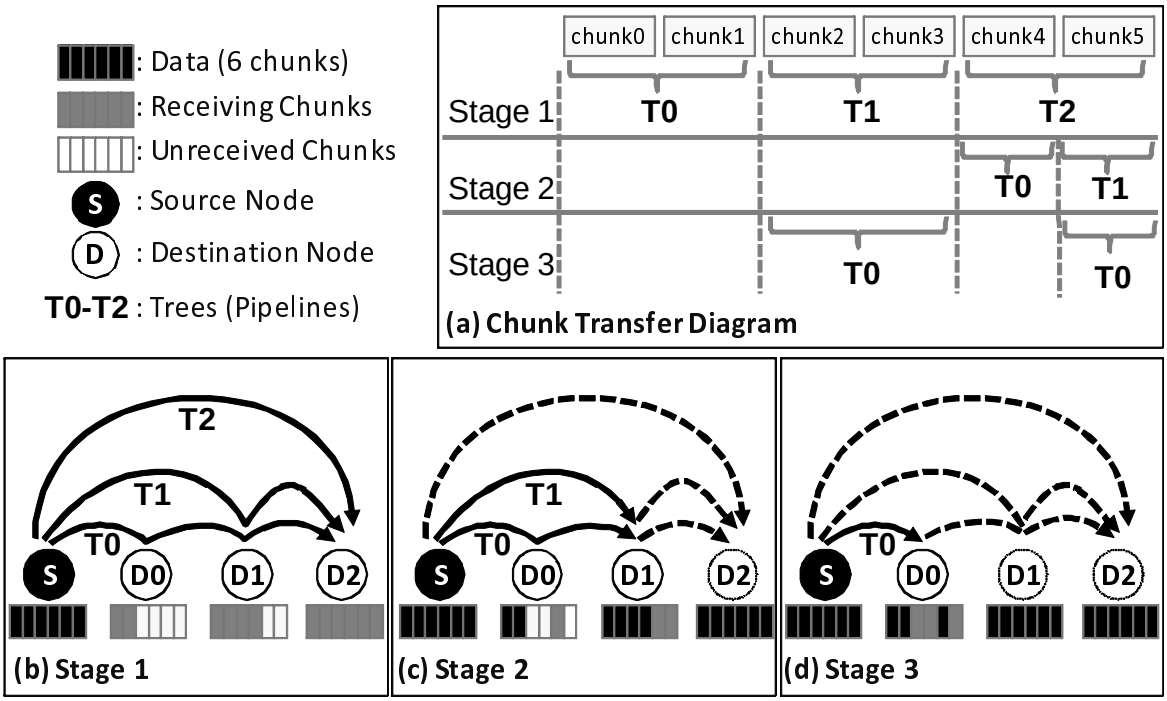


Figure 4.4: Data Flow

Data are sent in multiple *stages*, with each stage using different set of transfer trees. Let t_i be the transfer tree made by the i-th iteration of the construction algorithm ($i = 1, \ldots, n$) and denote by $N(t)$ the set of destinations involved in tree t. Note that we have $N(t_n) \subset N(t_{n-1}) \subset \cdots \subset N(t_1)$.

In the first stage, *the entire data* is partitioned into fragments and sent using *all* transfer trees. Data must be allocated to trees so that they finish the transfer at the same time. In principle this can be achieved by allocating to each tree the amount of data proportional to its throughput. In the actual implementation, we achieve this by dividing data into small chunks and by allocating data dynamically to trees. Each transfer edge is implemented by a TCP connection and we send a chunk via a connection only when the connection is writable (i.e., writing to it does not block). In the end of the first stage, nodes in $N(t_n)$ will have obtained the entire data and need not to receive data any more. Other nodes only get a part of the data, which will be delivered in later stages.

In the second stage, we send data that the nodes in $N(t_{n-1})$ have just missed in the first stage (i.e., data sent through t_n) using $t_1, \ldots, t_{n-1}$, with the same policy to allocate data to individual trees. The second stage will deliver all data to nodes involved in t_{n-1}. Similarly in the third stage, data that the nodes in $N(t_{n-2})$ have not yet received will be sent via $t_1, \ldots, t_{n-2}$ so they will have all data in the end of the stage. We repeat this until all destination nodes receive all data. Note that nodes that already have received all the chunks may need to stay the transfer, and continue relaying data to other nodes.

An example of stages are shown in Figure 4.4. In this example, three destinations $D_0 ... D_3$ requires data from S, and three trees (or pipelines) T_0, T_1 and T_2 is constructed. While T_0 connects all the

destinations, T_1 and T_2 includes only part of the destinations. Assume these three trees have the same bandwidth. Data are split into 6 chunks, and the delivering schedule is show in (a).

Picture (b) describes the first stage, where all three trees are used in parallel. As can be seen in (a), the three trees deliver two chunks each. The boxes under each node correspond to six chunks, and their color shows the status: black for received, gray for receiving and white for undeceived. While every chunk in S is black, other destinations only have gray and white chunks.

When each tree has delivered two chunks, the second stage starts. Now node D_2 has the entire data, and D_0 and D_1 have part of the data. Picture (c) shows the second stage, where only D_0 and D_1 continues receiving data. Note that the transfer through T_2 is no longer needed. Finally, in the third stage in picture (d), only D_0 has chunks that have not received. In this stage, D_1 and D_2 have the entire data.

4.3 Stability and Optimality for Tree Topology

We show that the algorithm introduced in the Section 4.2 is stable and optimal for a tree topology, in the sense of aggregate bandwidth. The proof consists of three parts. First, we show that we can treat each link as an undirected link during tree construction. After that, we show some behavior about the set of links used in direct transfers. Finally, we prove that the set of trees deliver to a node the maximum amount of data that can be received by the node.

4.3.1 Preparation: a Property from Symmetric Link

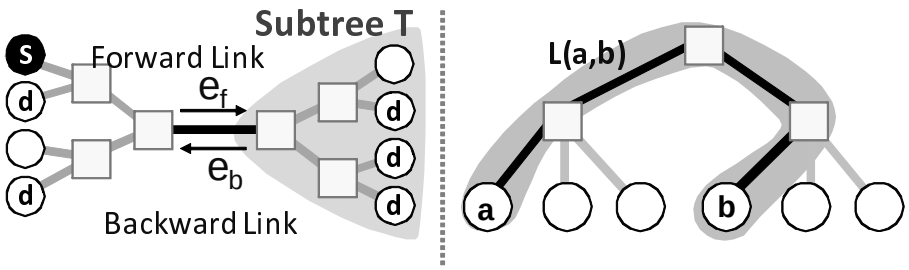
Basically, we need to treat each link as a directed link during the construction of trees. The bandwidths of the two directed links on the same path are separately calculated, and the link is added separately. However, when each link has the same bandwidth in the two directions, we can treat the two directed links as one link, whose bandwidth is given by the smaller bandwidth of the two directional links. The reason is shown as follows.

We have a tree topology that contains a source s and some destinations. A bidirectional link e in the network has two subtrees connected to both ends. Since the network is a tree, the source node s is contained in either of the two subtrees. Assume that the subtree T does not contain the source node s.

The link e can be split into two directional links: one heading for T and the other away from it. Let the term *forward link* e_f and *backward link* e_b denote the former and the latter link, respectively. This is shown in Figure 4.5(a). In a tree network, the removal of e from the topology disconnects any node in T and s. Therefore, if e_f is removed, we can create no route from s to any node in T.

Here, assume the backward link e_b is adopted by a tree. Since the tree uses e_b, it has reached one of the destinations in T. Without e_f, we cannot create a route from s to any node in T. Thus, when a tree contains e_b, it also employs the corresponding forward link e_f. Therefore, $B_k(e_b) - B_{k-1}(e_b) \leq B_k(e_f) - B_{k-1}(e_f)$ holds, where $B_k(e)$ denotes the available bandwidth used in the construction of the (k)th tree.

Since the bandwidth of a link is symmetric at first, $B_0(e_b) = B_0(e_f)$. As a result, $B_k(e_b) \leq B_k(e_f)$ is lead.



(a) Forward/Backward Link (b) An example of $L(a, b)$

Figure 4.5: Links in a Tree Symmetric Network

Thus, the bottleneck in the tree always lies on the forward link. Consequently, during the construction of the trees, we only need to use the bandwidth of the forward link.

4.3.2 Properties for Single Transfer

Since the topology is a tree, there is exactly one route that connects two nodes a and b using the minimum number of links. Let $L(a, b)$ denote this set of links. This is shown in Figure 4.5(b). The removal of any link in $L(a, b)$ from the tree disconnects a and b. Thus, any route from a to b contains all the links in $L(a, b)$. The transfer throughput is maximized in a route that uses each link in $L(a, b)$ once. Let $v_k(a, b)$ denote this throughput when the bandwidth is B_k:

$$v_k(a, b) = \min_{e \in L(a,b)} (B_k(e)) \tag{4.1}$$

With a set of links $(L(a, b) \cup L(b, c))$, we can construct a path from a to c via b. Therefore, the following relation holds:

$$(L(a, b) \cup L(b, c)) \supset L(a, c) \tag{4.2}$$

Assume we iteratively construct a total of n trees. From the source s, a tree traces some destinations in a depth-first path. Let D_k denote the set of destinations included in the (k)th depth-first tree, and P_k denote the set of links used in the (k)th tree.

Using the above notations, the following equations (4.3) and (4.4) hold. Because the graph is a tree,

$$P_k = \bigcup_{d \in D_k} L(s, d). \tag{4.3}$$

Because the algorithm traces the path from s,

$$d \notin D_k \text{ if and only if } v_k(s, d) = 0. \tag{4.4}$$

4.3.3 Proof of Stability and Optimality

Let $d_{k,0} \ldots d_{k,n-1}$ denote the nodes in the destination set D_k, where $v_k(s, d_{k,i}) \leq v_k(s, d_{k,i+1})$. Let u_k denote the throughput of the tree P_k. We get the following:

$$u_k = \min_{e \in P_k}(B(e)) = \min_{0 \leq i < n} (v_k(s, d_{k,i})) = v_k(s, d_{k,0}). \tag{4.5}$$

The following equation holds.

$$D_{k+1} = D_k \setminus A_k, \tag{4.6}$$

where $A_k = \{d \in D_k | v_k(s,d) = v_k(s,d_{k,0})\}$.

Proof. (Proof of (4.6))

From the definition of B_k,

$$B_{k+1}(e) = \begin{cases} B_k(e) - u_k & \text{if } e \in P_k, \\ B_k(e) & \text{otherwise.} \end{cases}$$

Thus, from (4.4),

$$v_{k+1}(s,d) = \begin{cases} v_k(s,d) - u_k & \text{if } d \in D_k, \\ 0 & \text{otherwise.} \end{cases} \tag{4.7}$$

If $d \in A_k$, namely $v_k(s,d) = v_k(s,d_{k,0})$, then $v_{k+1}(s,d) = u_k - u_k = 0$ from (4.5). If $d \in D_k \setminus A_k$, namely $v_k(s,d) > v_k(s,d_{k,0})$, then $v_{k+1}(s,d) = v_k(s,d) - u_k > 0$.

From (4.4), $D_{k+1} = \{d \in D_0 | v_{k+1}(s,d) \neq 0\} = D_k \setminus A_k$. □

From (4.6), a destination d in A_k receives data from the (0)th to the (k)th trees. Let $w(d)$ denote the total bandwidth d receives from the trees, that is,

$$w(d) = \sum_{0 \leq i \leq k} u_k. \tag{4.8}$$

From (4.7), the following holds:

$$\begin{aligned} u_k = v_k(s,d_{k,0}) &= v_{k-1}(s,d_{k,0}) - u_{k-1} \\ &= v_0(s,d_{k,0}) - \sum_{0 \leq i < k} u_i. \end{aligned}$$

Thus

$$v_0(s,d_{k,0}) = \sum_{0 \leq i \leq k} u_i. \tag{4.9}$$

From (4.8) and (4.9), we get the following:

$$\forall d \in A_k, w(d) = v_0(s,d_{k,0}) \tag{4.10}$$

This means the trees deliver to d the same amount of data as in the direct transfer from s to d in the condition that there is no other traffic. From 4.1, this is the maximum amount of data d can receive.

From the definition of A_k and (4.6), every destination is included in one of $A_0, \ldots, A_{n-1}$, where n is the number of the trees.

Consequently, it is proved that our trees deliver to a node the maximum amount of data that can be received by the node. It is also stable because the receiving bandwidth does not depend on other destinations.

4.4 Improvement in Graph Topology

Although no algorithm can achieve both stability and optimality together, our a broadcast algorithm improves aggregate bandwidth than previously proposed algorithm such as FPFR. Since trees that our algorithm constructed are the superset of trees used by FPFR and Balanced Multicasting, the aggregate bandwidth of these algorithms is never greater than ours.

For example, as we have seen in Subsection 4.2.2, Figure 4.3 shows the tree construction process in our algorithm. While our algorithm uses four trees, FPFR and Balanced Multicasting only uses the first two trees. While the aggregate bandwidth of the first two trees is 20, our algorithm achieves 35 aggregate bandwidth by using four trees.

Chapter 5

UCP - Universal Broadcast Tool

In this chapter, we introduce a broadcast tool named *ucp*, which is based on our broadcast algorithm explained in Chapter 4. Its characteristics are shown in Section 5.1, and Section 5.2 illustrates our approach to traverse NAT and firewalls. Finally, the user interface of *ucp* is shown in Section 5.3.

5.1 Characteristics of ucp

Three major characteristics of *ucp* are described below:

Efficient Broadcast
: Since a transfer plan is constructed with the broadcast algorithm explained in Chapter 4, *ucp* receives benefit from it. In a tree topology whose link has a symmetric bandwidth, the broadcast algorithm is *stable* and optimal: a node that lacks bandwidth does not degrade bandwidth of other nodes, and each node receives as much amount of data as possible at a time. For general graphs, it outperforms other broadcast algorithms in terms of aggregate bandwidth, In addition, it takes only 2 milliseconds to obtain a broadcast schedule for 400 nodes.

NATs/Firewall Traversal
: A distributed environment often contains NATs and firewalls, which block some connections among nodes in the environment. Our *ucp* tool traverses NATs and firewalls by using relay technique, Routing information is specified in advance. When a connection cannot be established from the source to the destination, it tries to establish a connection from the destination to the source. If both connections are not possible, some intermediate nodes are used to relay the transfer.

High Usability
: Since our *ucp* tool is designed for interactive use, it achieves high usability. It uses a cluster middleware called *GXP* [21] to spawn processes in many nodes. With our URI-like expression, multiple nodes and files can easily be specified. As a result, users can dynamically specify a broadcast job with just a few commands.

The following sections will write about the details.

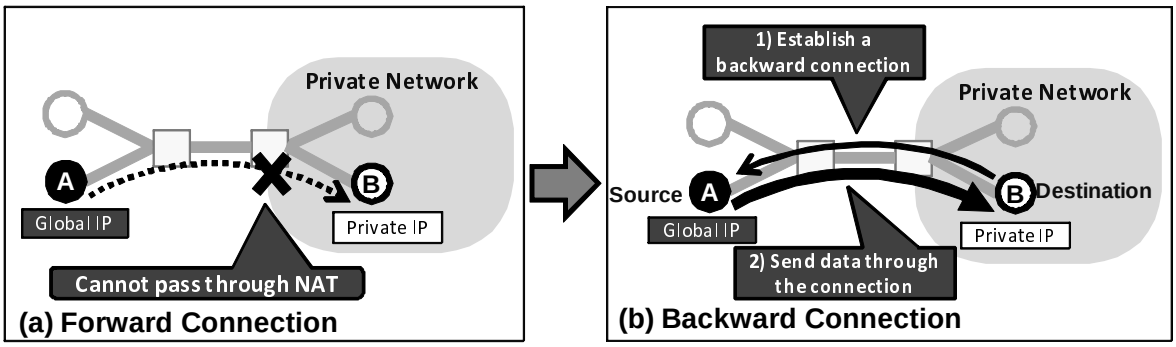


Figure 5.1: Backward Connection to Traverse NATs and Firewalls

5.2 NATs and Firewalls

This section describes our approach to traverse NATs and firewalls in *ucp*. Although a NAT and firewall connections forbids some connections traversing it as shown in Section 2.4, we use some techniques to traverse them. Thus, users do not need to care about it.

We use two ideas for that, one is backward connection technique, the other is relaying. They are shown below:

5.2.1 Backward Connection

Even when one node A cannot connect to another node B via socket, the connection from B to A may be possible. We call the former connection as *forward*, latter as *backward* connection. This situation happens between a node having a global address and a node only having a private address. To traverse NATs and firewalls, *ucp* utilizes backward connection when forward connection is blocked.

An example is shown in Figure 5.1, where node A has a global address and node B behind a NAT does not. While node A is accessible from anywhere, node B is only accessible from the private network it belongs to. When node A sends data to node B, A cannot establish a forward connection to B. Instead, first B connects to A and a connection is established. After that, A sends data to B.

With this technique, an ordinary connection is established through NATs and firewalls. This technique can be used when one peer has a global address.

5.2.2 Application-level Routing

When two peers cannot establish a connection in both directions, *ucp* employs the application-level routing technique. This happens when two peers are in different private networks or behind different firewalls. Although the previous backward connection technique cannot be applied in these cases, the two peers can transmit a message when another node relays data. This intermediate node needs to be accessible from both peers.

Figure 5.2 illustrates an example, where node B transfers data to C. Since nodes B and C are in different private networks, they cannot establish a direct connection either in forward or in backward. This situation is solved when node A, which is accessible from anywhere, relays data transfer. Figure

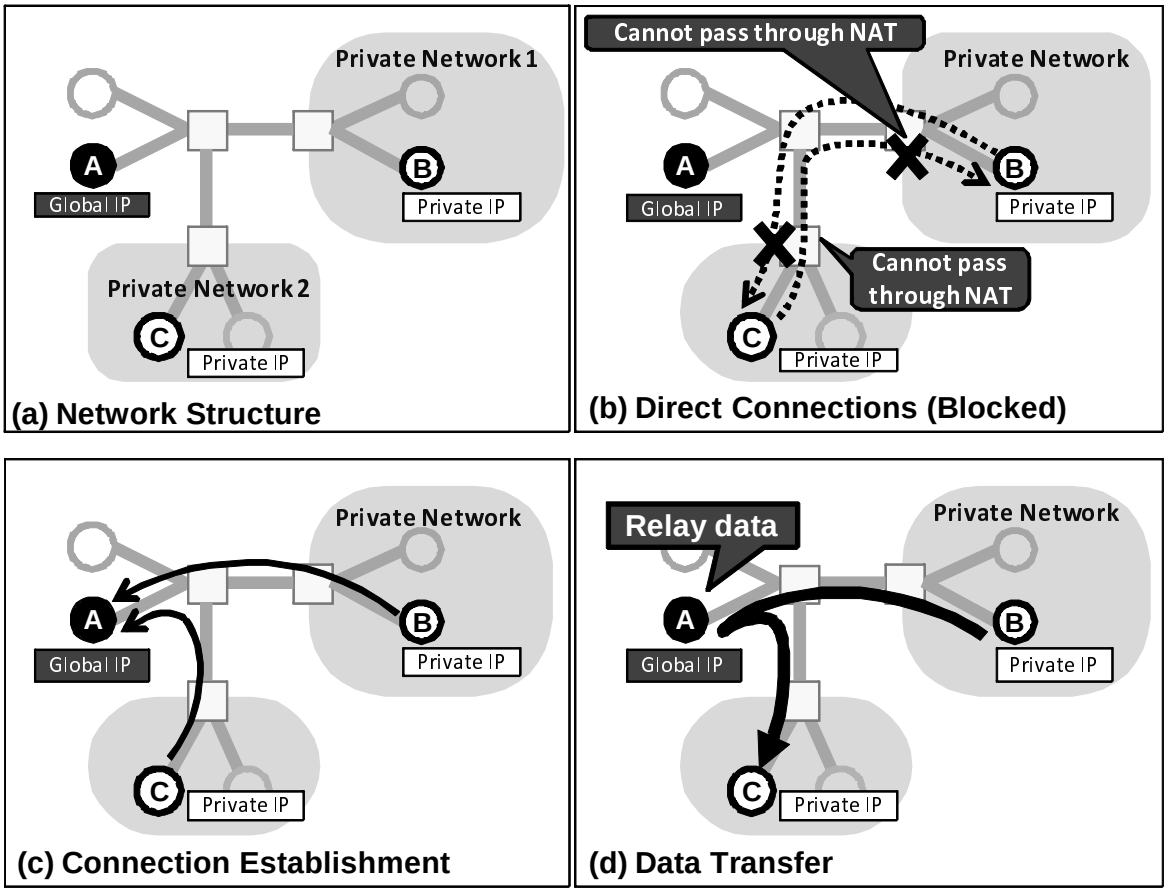


Figure 5.2: Relaying to Traverse NATs and Firewalls

5.2(c) shows the connection establishment phase, in which node B and C individually connect to node A. In Figure 5.2(d), data transfer is performed from B to C via A.

Since relaying method takes larger overhead than the direct socket transfer, the ucp tool uses relaying technique only when backward connection is not possible.

5.3 User Interface

Unlike many other remote data transfer tools, *ucp* is designed for interactive use. This section discusses about its interface for users.

5.3.1 Process Startup

To spawn processes in remote nodes, *ucp* uses a middleware called *GXP* [21]. With *GXP*, remote machines in a distributed environment can interactively be used. It accesses nodes with rsh, ssh, and batchqueue, spawning a daemon on each node. After logging in to remote nodes, users can run any command allowed on that machine. It also provides basic communication among remote processes.

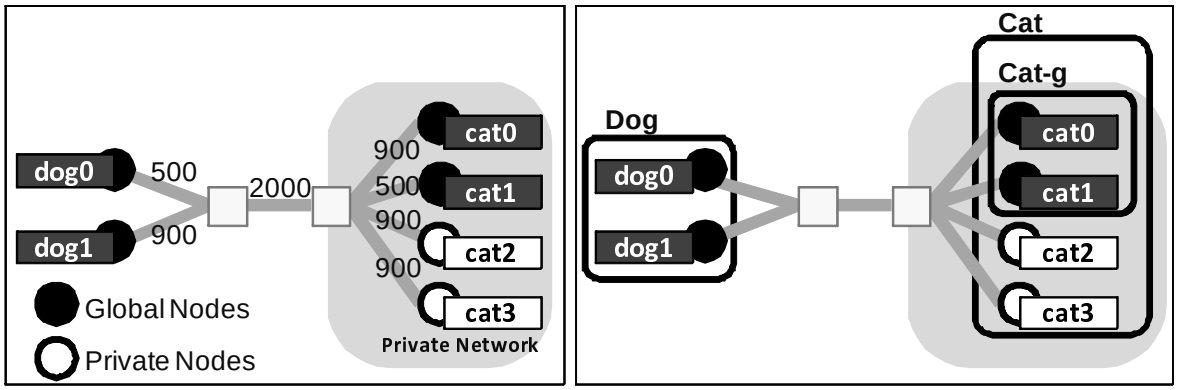


Figure 5.3: An Example Bandwidth-annotated Topology

One node works as a master, which plans broadcast, negotiates connections and order nodes to send, receive and relay data.

A typical startup routine is as follows:

```
$ gxpc use ssh .
$ gxpc explore [hostnames]
$ gxpc ucp dog000:/data/some/path/ cat:/data/dest/path
```

In the first line, it is specified to log in to any node via ssh. The second line actually spawns daemons of *GXP* on nodes specified by `[hostnames]`. A *ucp* job has actually started in the third line. In this example, all the files in `/data/some/path/` on a node `dog000` are broadcasted into `/data/dest/path/` on every node whose hostname matches against `cat`, such as `cat000`.

5.3.2 Topology Specification

As mentioned before, *ucp* requires bandwidth-annotated topology information to plan an efficient broadcast. It imports a topology generated by the topology inference tool implemented by Shiraiet al. [18]. The bandwidth of each link is measured by a tool implemented by [28]. An XML file is used to convey topology and bandwidth information. Figure 5.4 describes a topology shown in Figure 5.3.

A node and its child nodes has a physical connection, whose bandwidth is given by the `bandwidth` attribute. For example, a node `dog000` is connected to a switch in the bandwidth value 500.

5.3.3 Routing Specification

For a environment containing NATs and firewalls in it, *ucp* requires routing information. It is specified in a configuration file, which describes a group of nodes and connectivity among groups.

A host group consists of a label and regular expression. In a configuration file, one group is described as follows:

```
label host_pattern
```

```
<CLUSTER>
  <SWITCH>
    <SWITCH bandwidth="2000">
      <NODE bandwidth="500"><HOSTNAME>dog000</HOSTNAME></NODE>
      <NODE bandwidth="900"><HOSTNAME>dog001</HOSTNAME></NODE>
    </SWITCH>
    <NODE bandwidth="900"><HOSTNAME>cat000</HOSTNAME></NODE>
    <NODE bandwidth="500"><HOSTNAME>cat001</HOSTNAME></NODE>
    <NODE bandwidth="900"><HOSTNAME>cat002</HOSTNAME></NODE>
    <NODE bandwidth="900"><HOSTNAME>cat003</HOSTNAME></NODE>
  </SWITCH>
</CLUSTER>
```

Figure 5.4: Topology Expression

```
dog dog
cat-global cat00[01]
cat cat

socket dog > dog
socket cat > cat
socket dog cat > dog cat-global
```

Figure 5.5: Routing Expression

where the group consists of nodes whose hostnames are matched against `host_pattern`. and `label` denotes this group. For example, the first three lines in Figure 5.5 describes three host groups. Among nodes appeared in Figure 5.3, the first line matches node `dog000, dog001`. A node may belong to multiple groups.

After every group is defined, connectivity definition follows. The next expression describes an expression for the connectivity:

```
method source_group > destination_group
```

This line denotes that nodes in `source_group` are able to establish a connection between nodes in `destination_group` by `method`. The `source_group` and `desination_group` may consist of multiple group labels. The latter three lines in Figure 5.5 describes an environment illustrated in Figure 5.3. For example, the last line denote that nodes in `dog` and `cat` group can establish a connection between nodes in `dog` and `cat-global` group via socket.

5.3.4 Data Specification

A URI-like expression has commonly been used in interactive data transfer tools such as *scp* and *rsync*. In this expression, a file in a node is expressed in the following expression:

```
hostname:path
```

where `hostname` denotes a node and `path` denotes the location of a file or a directory. While this expression contains necessary and sufficient condition to specify a path on a certain node, it cannot express multiple nodes.

In common unix shells, multiple files are easily specified by using regular expressions. Similarly, *ucp* extended the URI-like expression to specify multiple nodes and files. It uses to specify node a hostname pattern, instead of hostname itself. The expression has almost the same syntax:

```
hostname_pattern:path_pattern
```

While the former `hostname_pattern` denotes the pattern of hostnames, the latter `path_pattern` expresses the pattern of files. Since hostnames in one cluster environment often have the same suffix and serial numbers, a simple regular expression matches nodes in one cluster. Consequently, users can easily specify a broadcast task to many nodes.

Another benefit of hostname pattern is that users need not use FQDN, fully qualified domain name. Under the assumption that every node concerning in the transfer has a different hostname, a node can be identified with a short hostname. Nodes that only have a private address can also be specified in this manner. Note that *ucp* has already access to every node via *GXP*.

Chapter 6

Evaluation

We implemented the algorithm, and evaluated it in both a simulation and a real environment. Five algorithms are compared:

Ours
: This is the algorithm we proposed. Multiple partial trees are iteratively constructed in a depth-first manner, and transfers are performed in parallel.

Depth-First (FPFR [11]-like)
: The algorithm constructs multiple spanning trees in a depth-first manner [7, 11].

Dijkstra
: This algorithm iteratively builds a tree in a greedy manner. Pick one unreached destination that can be reached in the maximum possible bandwidth from reached nodes. The method is explained in [11], and a similar method is proposed in [23].

Random Pipeline
: This algorithm randomly creates a tree. One node is randomly chosen from all the unreached destinations at a time. For each condition, we generated 100 candidates and chose the candidate with the largest aggregate bandwidth.

Flat Tree
: The source sends the data to all the destinations.

Except for *Flat-tree*, every algorithm requires a bandwidth-annotated topology.

6.1 Simulation

A simulator has been implemented to evaluate broadcast algorithms. We make the throughputs of the machines and switches large enough compared to the link bandwidths, so that they do not become bottlenecks. We used a tree topology with 400 nodes, taken from the real environment. During the experiment, three different bandwidth distributions were tested:

Table 6.1: Number of Nodes Used in the Experiments

Number of Destinations	A	B	C	D
105	59	9	35	2
47	30	1	16	1
11	6	1	3	1

Uniform Random

A uniform random value is assigned to each link to see general behavior of these algorithms. Both *low-variance* (from 500 to 1000) and *high-variance* (from 100 to 1000) conditions are tested.

Mixed Fast and Slow Links

While 80% of the links have broad bandwidth (1000), the others have narrow one (100). It describes a situation that some nodes have a slower NIC, or some switches are slow.

Random Inter-Cluster

In this condition, inter-switch links are assigned random distribution (from 100 to 1000).

The simulation is performed 10 times for the same algorithm, bandwidth variance and number of destinations. Both symmetric and asymmetric conditions are tested for each bandwidth distribution.

The result is shown in Figure 6.2. The vertical axis shows the relative aggregate bandwidth of the *FlatTree algorithm*. Our algorithm took only 2 milliseconds to induce the schedule. Our method achieved the best performance in every symmetric condition. As shown in Figure 6.2(c), the improvement is especially notable when fast and slow links are mixed. In this case, the performance of the other algorithms is dropped because of the lack of stability. In an asymmetric network, the performance of our algorithm is the best except for one case, Figure 6.2(d). Even in this case, the difference is only 3%. As a result, the superiority of our algorithm is confirmed.

We have also performed an experiment that simulates multiple source broadcast. It is shown in Figure 6.2 (i). While we changed the number of destinations from 10 to 50, the number of sources are also changed from 1 to 5. Figure 6.2(i) shows that the performance compared to the *FlatTree* is not changed in the multi-source multicast.

6.2 Real Machine Experiment

We also performed some experiments using a real machine environment. This environment had the tree topology shown in Figure 6.1, with 105 nodes in 4 clusters. The bandwidth of the links are also shown in Figure 6.1. We performed broadcasts among 10, 47 and 105 nodes. Table 6.1 shows the number of nodes from each cluster. Each condition was tested four times, and the best aggregate bandwidth was taken.

Figure 6.3 illustrates the results. Since the environment is heterogeneous, the performance was improved significantly by our algorithm. The aggregate bandwidth increased by 2.1 to 2.6 times compared to the best result in the other algorithms. However, the bandwidth was 30% to 45% worse than the optimal value predicted by simulation. An investigation revealed that our assumption that each link is bidirectional and thus has an independent (full-duplex) bandwidth in each direction

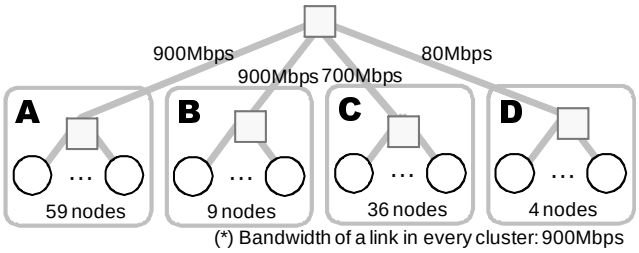


Figure 6.1: The Real Environment Topology

was subtly violated, because transferring data in both directions saturated CPUs. While each node could send or receive 900Mbps independently, it could only send and receive 750Mbps when it was simultaneously sending and receiving. In the simulation, the link adjoining such a node was modeled as having 900Mbps in each direction, but in reality each only had 750Mbps when it was used to relay data.

To demonstrate the stability of our broadcast, another test was performed. In this experiment, a node with small bandwidth joined a 9-node broadcast. Figure 6.3 (d) shows the change in the aggregate bandwidth of the older 9 nodes for four broadcast algorithms. While the aggregate bandwidth dropped to 52% in *depth-first*, which works the same as *FPFR* for a tree, the deterioration in our algorithm was only 2.5%. Although the drop in the other two algorithms were also small, our algorithm yielded 1.5 times aggregate bandwidth of them.

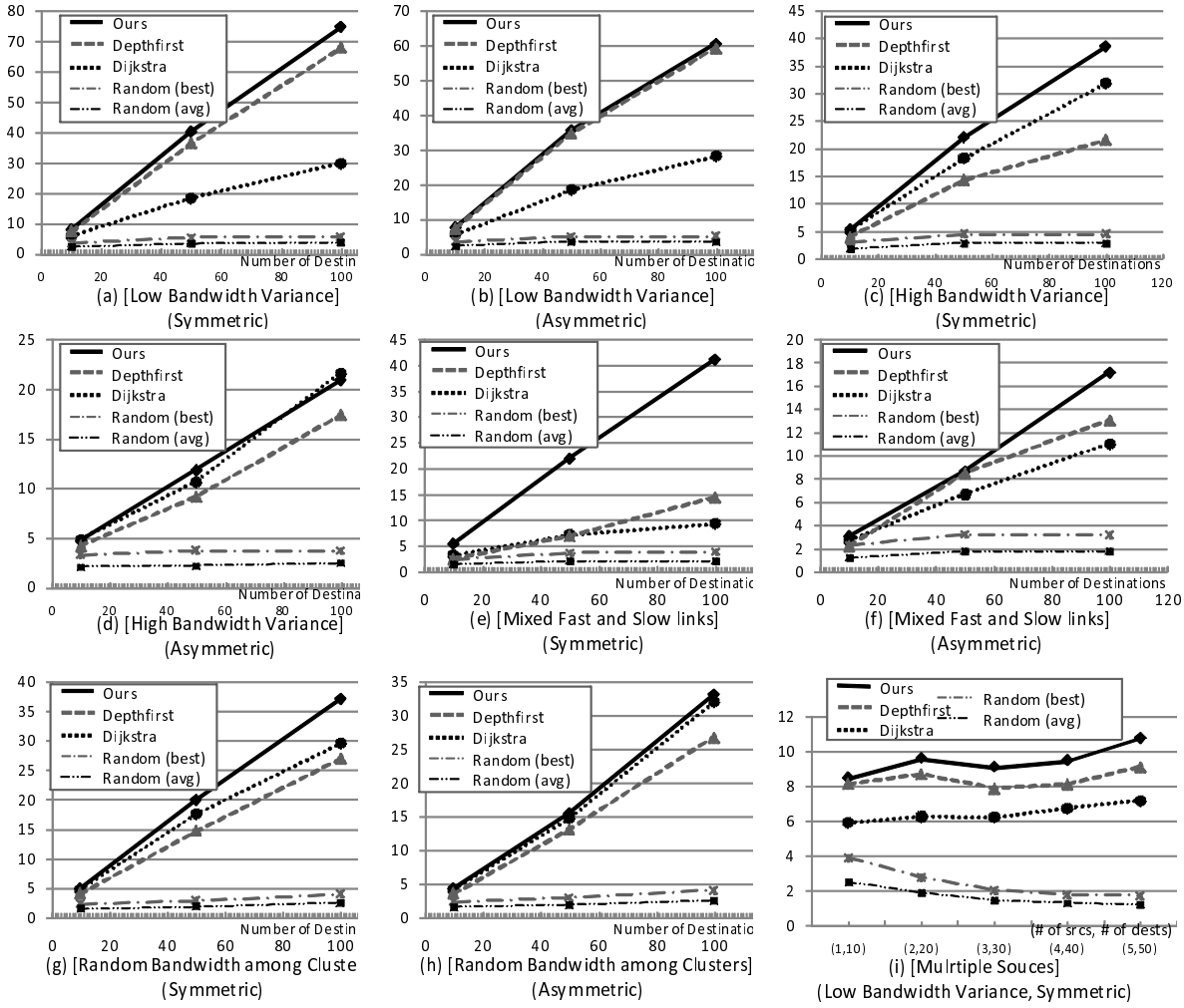


Figure 6.2: The Relative Bandwidth to the *FlatTree* Algorithm in Simulations (Vertical axis: relative throughput to FlatTree)

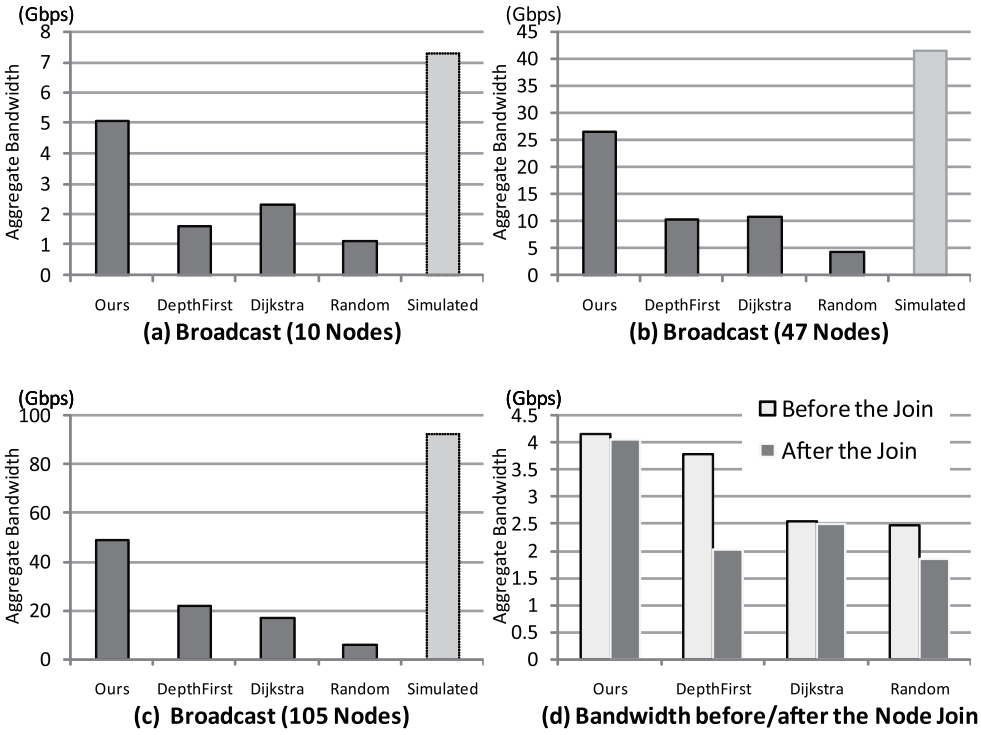


Figure 6.3: Real Machine Experimets

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we introduced the notion of *stability* in broadcasts, and proposed a simple and efficient broadcast algorithm for heterogeneous environments.

Our broadcast improves a previously proposed class of broadcast algorithms that include FPFR and Balanced Multicasting, focusing on the fact that each node should receive data as fast as possible and start computation without waiting for other nodes. Like FPFR and Balanced Multicasting, our broadcast achieves high bandwidth by forwarding data along multiple spanning trees. In addition, our broadcast avoids the effect of nodes with narrow bandwidth by forwarding data along multiple partial trees. While the slowest node only receives data from the spanning trees, faster nodes also receive data from the partial trees. For general graphs, our broadcast will always outperform FPFR and Balanced Multicasting, and for trees, we proved that it is *stable* as well as *optimal*.

In a real environment with 100 machines in 4 clusters, our scheme achieved 2.1 to 2.6 times aggregate bandwidth compared to the best result in the other algorithms. We also demonstrated the stability by adding a slow node to a broadcast. While the aggregate bandwidth dropped to 52% in *depth-first*, which works the same as *FPFR* for a tree, the deterioration in our algorithm was only 2.5%. Some simulations also showed that our algorithm also performs well in many bandwidth distributions. Even in an asymmetric environment, our algorithm outperformed all other methods that we have tested.

Our algorithm is implemented on a broadcast tool called *ucp*. Although direct connections are sometimes blocked by NATs and firewalls in a distributed environment, *ucp* traverses them by using backward connections and relaying technique. With its flexible host specification syntax, multiple nodes and nodes that only have private addresses are easily specified. As a result, users can interactively perform a broadcast task, ensuring high usability.

7.2 Future Work

In the future work, we are going to invent an algorithm that maximizes aggregate bandwidth in arbitrary graph networks. As shown in Section 4.4, our algorithm improves the aggregate bandwidth also in a case of common graph topologies. However, it is not proved to be optimal. Although no algorithm can achieve both stability and optimality in a graph topology, it is still important to

deliver as much amount of data as possible to each host in a broadcast, in order to effectively execute data-intensive applications.

It is also desirable to consider some other factors that limit data transfer throughput, such as processing throughput of nodes and switches. Since our model only treats link capacities, the experiment in our environment could not perform the theoretical aggregate bandwidth. By counting other factors into account, we can expect the behavior more precisely, and might obtain a better transfer plan in some cases.

Another future work is about a method that can adaptively improve the transfer schedule. In a WAN environment, bandwidth sometimes fluctuates because of other traffic. To cope with this fluctuation, we will utilize dynamically measured bandwidth during a transfer, and plan the transfer schedule again.

Our original purpose was efficient execution of data-intensive applications. To achieve this goal, we plan to integrate our broadcast algorithm and a task scheduler. With our algorithm, data transfer time as well as arriving bandwidth on each node is predicted. Based on this information, a better task schedule requiring small transfers will be obtained.

References

[1] NLANR/DAST : Iperf - the TCP/UDP bandwidth measurement tool http://dast.nlanr.net/Projects/Iperf/, Accessed 2007.

[2] Torque resource manager, Accessed Jan. 2008.

[3] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Lasislau L. Bölöni, Muthucumara Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J. Parallel Distrib. Comput.*, 61(6):810–837, 2001.

[4] Charlie Catlett. The philosophy of teragrid: Building an open, extensible, distributed terascale facility. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 8, Washington, DC, USA, 2002. IEEE Computer Society.

[5] L. Cherkasova and J. Lee. FastReplica: Efficient Large File Distribution within Content Delivery Networks. In *4th USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[6] Mathijs den Burger and Thilo Kielmann. MOB: Zero-configuration, High-throughput Multicasting for Grid Applications. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 159–168, June 2007.

[7] Mathijs den Burger, Thilo Kielmann, and Henri E. Bal. Balanced Multicasting: High-throughput Communication for Grid Applications. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, November 2005.

[8] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[9] Fabrizio Gagliardi, Bob Jones, Mario Reale, and Stephen Burke. European datagrid project: Experiences of deploying a large scale testbed for e-science applications. In *Performance Evaluation of Complex Systems: Techniques and Tools, Performance 2002, Tutorial Lectures*, pages 480–500, London, UK, 2002. Springer-Verlag.

[10] N. HU and P. STEENKISTE. Evaluation and characterization of available bandwidth probing techniques, 2003.

[11] R. Izmailov, S. Ganguly, and N. Tu. Fast Parallel File Replication in Data Grid. In *Future of Grid Data Environments Workshop (GGF-10)*, March 2004.

[12] N. T. Karonis, B. R. de Supinski, W. Gropp I. Foster, E. Lusk, and J. Bresnahan. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. pages 377–384, May 2000.

[13] Daisuke Kawahara and Sadao Kurohashi. Case frame compilation from the web using high-performance computing. *IPSJ SIG Notes*, 2006(1):67–73, 20060112.

[14] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the grid, 2004.

[15] Yuya Machida, Shin'ichiro Takizawa, Hidemoto Nakada, and Satoshi Matsuoka. Multi-replication with intelligent staging in data-intensive grid applications. In *GRID*, pages 88–95, 2006.

[16] W. Gentzsch (Sun Microsystems). Sun grid engine: Towards creating a compute power grid. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 35, Washington, DC, USA, 2001. IEEE Computer Society.

[17] Dongyu Qiu and R. Srikant. Modeling and Performance Analysis of BitTorrent-Like Peer-to-Peer Networks. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 367–378, August 2004.

[18] Tatsuya Shirai, Hideo Saito, and Kenjiro Taura. A Fast Topology Inference — A building block for network-aware parallel computing. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 11–21, June 2007.

[19] Se-Chang Son. The kangaroo approach to data movement on the grid. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, page 325, Washington, DC, USA, 2001. IEEE Computer Society.

[20] Jacob Strauss, Dina Katabi, and Frans Kaashoek. A measurement study of available bandwidth estimation tools. In *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 39–44, New York, NY, USA, 2003. ACM.

[21] Kenjiro Taura. Gxp: An interactive shell for the grid environment. In *IWIA '04: Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'04)*, pages 59–67, Washington, DC, USA, 2004. IEEE Computer Society.

[22] Srikumar Venugopal, Rajkumar Buyya, and Kotagiri Ramamohanarao. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Comput. Surv.*, 38(1):3, 2006.

[23] Reen-Cheng Wang, Su-Ling Wu, and Ruay-Shiung Chang. A novel data grid coherence protocol using pipeline-based aggressive copy method. In *GPC*, pages 484–495, 2007.

[24] Baohua Wei, G. Fedak, and F. Cappello. Scheduling independent tasks sharing large data distributed with bittorrent. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 219–226, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Takahiro YAGI, Shigeo SHIODA, and Kenichi MASE. A new approach for measuring the bottleneck link bandwidth using probe packets. **情報処理学会研究報告**. *QAI, [***高品質インターネット***]*, 2003(12):93–98, 2003.

[26] 斎藤秀雄, 鴨志田良和, 澤井省吾, 弘中 健, 高橋 慧, 関谷岳史, 柴田剛志, 横山大作, and 田浦健次朗. Intrigger：柔軟な構成変化を考慮した多拠点に渡る分散計算機環境. **情報処理学会研究報告**. *HPC, [***ハイパフォーマンスコンピューティング***]*, 2007.

[27] 蓬来祐一郎, 西田 晃, and 小柳義夫. 木構造型ネットワークにおける最適 broadcast スケジューリング (hpc-4 : ネットワーク)(2003 年並列/分散/協調処理に関する『松江』サマー・ワークショップ (swopp 松江 2003)). **情報処理学会研究報告**. *[***ハイパフォーマンスコンピューティング***]*, 2003(83):59–63, 2003.

[28] 長沼 翔, 高橋 慧, 柴田剛史, 田浦健次朗, and 近山 隆. ネットワークトポロジを考慮したバンド幅推定の高速化手法. **情報処理学会研究報告**, 2008.

Publications

Refereed Papers

1. Kei Takahashi, Hideo Saito, Takeshi Shibata and Kenjiro Taura. A Stable Broadcast Algorithm. To appear in the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid2008), May 2008

Unrefereed Papers

1. 高橋 慧, 田浦健次朗, 近山 隆. トポロジを考慮しソース選択を行うデータ転送スケジュラー. 並列／分散／協調処理に関するサマーワークショップ（SWoPP2007），旭川，2007 年 8 月.
2. 高橋 慧, 田浦健次朗, 近山 隆. マイグレーションを支援する分散集合オブジェクト. 並列／分散／協調処理に関するサマーワークショップ（SWoPP2005），武雄，2005 年 8 月.
3. 柴田剛志，斎藤秀雄，横山大作，弘中健，澤井省吾，高橋慧，頓楠，鴨志田良和，田浦健次朗. 多拠点分散環境（InTrigger）における計算支援ソフトウェアの検証. 日本ソフトウェア科学会第 24 回大会，奈良，2007 年 9 月.
4. 斎藤秀雄，鴨志田良和，澤井省吾，弘中健，高橋慧，関谷岳史，頓楠，柴田剛志，横山大作，田浦健次朗. InTrigger：柔軟な構成変化を考慮した多拠点に渡る分散計算機環境. 情報処理学会研究報告 HPC-111 (SWoPP 2007)，pp.237-242，旭川，2007 年 8 月.

Poster Presentations

1. 高橋 慧, 田浦健次朗, 近山 隆. マイグレーションを支援する分散集合オブジェクト. 先進的計算基盤システムシンポジウム（SACSIS2005），筑波. 2005 年 5 月.