Towards Efficient Distribution of High-Volume Content

Mukund Seshadri

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-167
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-167.html

December 11, 2006

Towards Efficient Distribution of High-Volume Content

by

Mukund Seshadri

B.Tech. (Indian Institute of Technology - Madras) 2000
M.S. (University of California, Berkeley) 2002

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Randy H. Katz, Chair
Professor Ion Stoica
Professor Charles Stone

Fall 2006

The dissertation of Mukund Seshadri is approved.

Chair	Date
	Date
	Date

University of California, Berkeley

December 2006

Abstract

Towards Efficient Distribution of High-Volume Content

by

Mukund Seshadri

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Randy H. Katz, Chair

We consider the problem of a source distributing a large file, such as a software package or multimedia content, to a large number of clients in the least possible time. We first tackle it under the assumption that the order of data delivery is immaterial. We obtain a cooperative algorithm analytically, and prove that it is optimal for a simple homogeneous client model. This algorithm is at least twice as fast as the well-known multicast tree method. We also explore, by simulation, a simple randomized version, and find that this algorithm performs surprisingly close to optimal. We then consider a scenario with heterogeneous client bandwidths and design an algorithm to handle this case.

Next, we analyze and simulate a non-cooperative scenario in which the clients need incentives to upload data to each other. Specifically we consider different algorithms based on "barter" and find that certain relaxations of barter perform almost as well as the optimal cooperative algorithm. Our work is inspired by the popular BitTorrent [8] protocol, which loosely incorporates mechanisms for efficiency and incentives. We explore BitTorrent's performance via simulation, studying the impact of different parameters. We find that BitTorrent's completion time is around twice the time of an optimal algorithm. BitTorrent's scaling behavior can be improved closer to the optimal algorithm's, but only if carefully tuned.

Finally, we propose an architecture that allows applications to customize the content delivery algorithms we develop. We propose that the delivery mechanism be divided into two layers, one application-specific, and one application-independent. The former interfaces to the latter via a narrow block prioritization channel. We argue that this is simple and powerful enough to accommodate a wide variety of applications. We design two different application-specific schemes as proof of concept, and evaluate them via simulations. These results indicate that our priority-based algorithms allow high rates of delivery of streaming or ordered data.

Professor Randy H. Katz
Dissertation Committee Chair

Contents

List of Figures

Chapter 1

Introduction

The world of content distribution is experiencing rapid growth, with many new challenges. Situations where gigabytes of content have to be quickly distributed to thousands of clients are likely to grow in importance in the near future. The volume of content and the scale of the client base pose a significant impediment to rapid distribution. Numerous algorithms have been proposed thus far, each with different assumptions and goals. But the focus has largely been on uncoordinated distribution of files to individuals, rather than *coordinated distribution of large files to large communities.*

Examples of the former are music file-sharing networks and web-page caching systems. This usually involves file-sizes of a few megabytes. While the number of clients is large, there is a lot of variance in the content desired by each client. The set of clients interested in a particular file tends to be extremely dynamic, with frequent arrivals and departures. Therefore, traditional algorithms have sought to minimize the latency or time for each individual's file to be downloaded.

Contrast this with the following example. A server wishes to distribute very large software packages and updates. These files can be hundreds of megabytes to several

http://mlb.mlb.com - Major League Baseball : MLB.com Media Player
MLB.com
ESPN.com
GO TO: Jul 18 OR Team Games GO
PREFERENCES
MLB.TV ::
MLB.COM ALL ACCESS ::
MLB.COM GAMEDAY AUDIO ::
MLB.COM, WHERE BASEBALL IS ALWAYS ON.
MLB.TV WATCH LIVE GAMES ONLINE TODAY
CLICK FOR TERMS & CONDITIONS BLACKOUT RESTRICTIONS APPLY

Game schedule for 07.18.2005

COL	@	WSH	LIVE
HOU	@	PIT	LIVE
KC	@	CLE	LIVE
TB	@	BOS	LIVE
CHC	@	CIN	LIVE
DET	@	CWS	LIVE
NYY	@	TEX	LIVE
BAL	@	MIN	LIVE
MIL	@	STL	LIVE
FLA	@	ARI	9:40pm
OAK	@	LAA	10:05pm
ATL	@	SF	10:15pm

NEWS > Trade winds: Burnett hot topic

Figure 1.1. A screen-shot of MLB-TV: subscribers can watch baseball online!

gigabytes in size. Such sizes are already seen today in operating systems releases like Fedora Core [20]. Assume that the server wishes to distribute its content as fast as possible to its entire client base. The clients are a well-known, mostly static, set of subscribers. The number of clients can be very large, numbering in the thousands. Since software updates can be critical, the total time to distribute entire packages to all clients, or the "completion time", becomes an important metric.

As another example, consider a service that distributes large video files, like movies or TV shows, to a registered client base. There is a demand for such services today, indicated by the plethora of movie download sites in existence. These range from legitimate commercial services like MovieLink [43], to other quasi-legal services based on IRC [33] and BitTorrent [8]. Even ITunes [34], which is one of the leaders in online music downloads, has recently added TV shows like "Lost" and other video content to its online catalog. If such a video distribution service were offered on a legal commercial basis, the service provider would have to be concerned with metrics for overall system performance, including completion time.

Motivated by these two examples, our focus is on *coordinated distribution of large* *files to large well-understood static communities.* While numerous content distribution algorithms have been proposed, and some deployed, there is no established science to understand what is the best method. Therefore, our first objective is to **define the best possible performance**, for a given set of assumptions, along with an algorithm to achieve it. Our second objective is to understand how other known algorithms perform in comparison.

In the two examples above, if the transmission bandwidth of the server is limited, traditional methods like client-server unicast could result in very large completion times. This can be reduced by arranging for high transmission bandwidths at the server. This can be an expensive proposition. Alternatively, one could consider utilizing the transmission capacities of the clients to help reduce the burden on the server. Several methods based on this idea have been proposed in the past.

File-sharing networks like Napster [47], Kazaa [38], and Gnutella [28] emerged as extremely popular applications around the turn of the century. Using these applications, users could search for other users hosting a file of interest. While these clients allowed clients to download files from other clients, very little optimization was provided to the actual download process. Therefore, if a large number of clients were interested in a particular file residing at a single source, these file-sharing applications would perform much worse than several alternative algorithms.

Another class of distribution algorithms have emerged, based on multicast trees. Proposals like End System Multicast [32] formed overlay network graphs of client nodes and propagated data along distribution trees in these graphs. A more popular, and more recent protocol is called BitTorrent [10, 8]. This is based on organizing clients into random overlay graphs. The file to be distributed is divided into smaller pieces, and these pieces are transmitted between client nodes in the overlay graph.

We aim to understand how well these different methods perform. Given our focus on large files and large static client-sets, neither Multicast nor BitTorrent are optimal if we consider completion times. BitTorrent, for example, was developed for client sets with high churn rates. Further, it cannot assume that clients are willing to upload data to each other, and has to build in incentives for this purpose. The corresponding design features adversely affect completion time. Therefore, it is important to quantify the performance of BitTorrent and other algorithms. We achieve this goal, for our scenarios of interest.

We find that BitTorrent can be significantly worse than optimal, if the objective is to distribute data to a cooperative static client base. Our research reveals ways to improve BitTorrent's performance for this scenario. We also propose new algorithms with better performance. We use a few basic principles of the BitTorrent protocol as a starting point for designing or adapting the new algorithms. Our improvements are in terms of the completion time for bulk data. We also consider streamed data, and consequently, metrics other than completion times. BitTorrent was not designed for ordered data delivery and performs very poorly in this case. We propose high-performance algorithms for this purpose as well.

The design and performance of our content-distribution algorithms are strongly influenced by assumptions about client behavior, the network environments, and application requirements. One important factor is whether clients are cooperative, in offering their upload bandwidth to serve other clients. In a system where clients are cooperative, we can assume they are always willing to transmit any block at the maximum possible rate to any other client. Then there is a great deal of freedom in devising the content-distribution algorithm. However, if clients are non-cooperative, or selfish, the set of algorithms one can consider is restricted. In this case, we can only consider algorithms that include mechanisms providing incentive for clients to upload data.

Another factor is the nature of the application and the requirements it poses on block delivery. For example, bulk content would require the blocks to arrive in no particular order. Therefore our only performance metric is completion time. On the other hand, streaming media prefers blocks to be downloaded in order. Furthermore, applications could pose additional constraints on delivery parameters, like loss rate and delay.

A third factor influencing algorithm design is the nature of client bandwidths. A heterogeneous client environment necessitates more robust and potentially slower distribution algorithms than a homogeneous environment.

Thus, we are faced with a large design space, characterized by several dimensions, or factors. One could consider more dimensions like the rate of arrival of clients into the system, or different file sizes. However, for our purposes, we assume that all other dimensions or factors are fixed. We have already mentioned our assumption of very large file sizes and client set sizes. We made this choice because small files and client sets do not pose a significant challenge, given the resources and algorithms available today.

Further, we assume that the client set involved is mostly static. That is, the rate of arrival and departure of clients into the system is zero, or very low compared to the time of distribution of the file. As Chapter 2 describes in detail, this space of problems has not been adequately addressed by techniques like Gnutella and BitTorrent. This assumption is reasonable, since network connectivity for desktop computers is already permanent in corporate LANs and broadband home connections. And we anticipate that the availability of "always-on" connectivity will only grow in the future. We further assume that client or network failures occur at a rate that is insignificant relative to typical file distribution times.

Thus, we have a restricted design space that requires study, and is characterized by variability along three dimensions. We have two choices for each dimension, which are mutually exclusive and exhaustive. These are summarized as:

- The data distributed can be "bulk" data or can require ordered delivery.
- Clients can be either cooperative or non-cooperative.
- The client bandwidths can be either homogeneous or heterogeneous.

Our goal is to *understand and design the fastest possible distribution algorithms for each scenario in this design space*. We explore these scenarios by partitioning the design space as illustrated in our graphical roadmap, in Figure 1.2. We begin by designing algorithms for the simplest combination of assumptions. We then increase the complexity step by step, until we cover the entire space of possible scenarios.

We first explore algorithms that do not make any assurances about the order of data delivery. That is, we are only concerned about the total distribution time. We answer the question: how can a server transfer a large file to *all* its clients in the shortest possible *total time*? A more detailed introduction of our study into this subspace of scenarios, along with further partitioning of this subspace, is presented in Section 1.1.

We then consider applications that could benefit from having their data delivered in accordance with some specific requirements. We investigate if such features can be provided without an adverse impact on the total completion time of the distribution process. Our investigation into this subspace of scenarios is introduced in greater detail in Section 1.2.

1.1 Bulk Content Distribution

We first fix one dimension to a simple choice. That is, we assume that the data is "bulk" data, with no significance attached to the order of delivery of individual

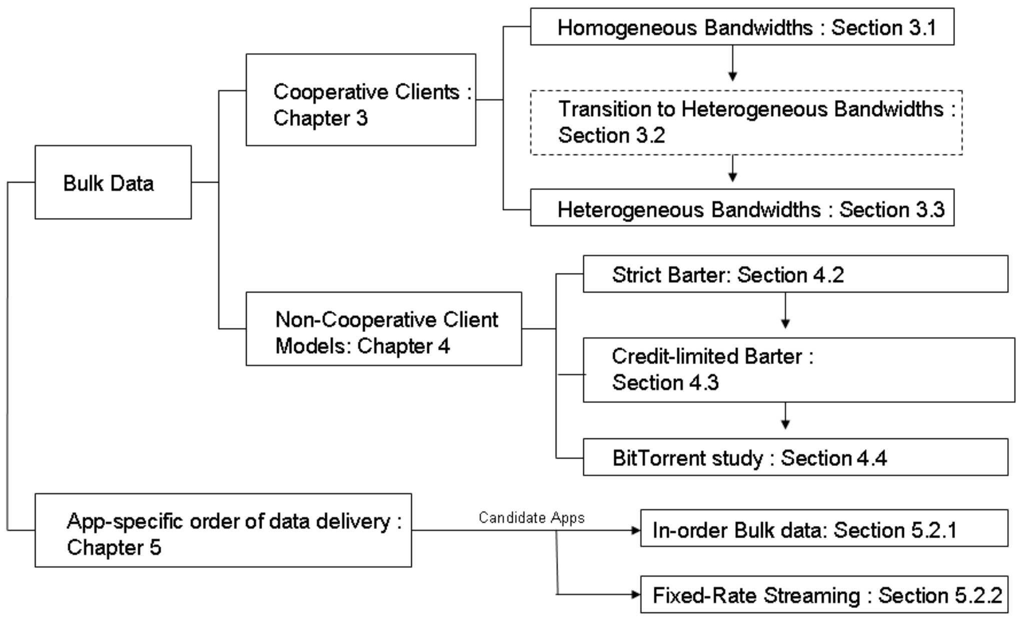

Bulk Data
Cooperative Clients : Chapter 3
Homogeneous Bandwidths : Section 3.1
Transition to Heterogeneous Bandwidths : Section 3.2
Heterogeneous Bandwidths : Section 3.3
Non-Cooperative Client Models: Chapter 4
Strict Barter: Section 4.2
Credit-limited Barter : Section 4.3
BitTorrent study : Section 4.4
App-specific order of data delivery : Chapter 5
Candidate Apps
In-order Bulk data: Section 5.2.1
Fixed-Rate Streaming : Section 5.2.2

Figure 1.2. Design Space and Roadmap of Study

fragments. The design of distribution algorithms is influenced by assumptions about how cooperative clients are in offering their upload bandwidth to help other clients. If the clients are cooperative, we can assume that any client will be willing to upload data to any other client at any time at its maximum possible rate. Thus, no constraints are imposed on algorithm design. However, if clients are likely to be non-cooperative, or selfish, the distribution algorithm needs to build in mechanisms to force clients to upload data. Typically, this is achieved by establishing a link between upload rates and download performance. We study different mechanisms for this purpose, based on the principle of *barter*. That is, a client does not upload data to another client unless it receives data in return.

We now introduce our work in the cooperative and non-cooperative scenarios is greater detail.

1.1.1 The Cooperative Case

Let us first consider a cooperative model, where clients are always willing to upload data at the maximum rate of which they are capable. In this case, one might imagine a variety of different algorithms for distributing content. For example, a simple well-known family of solutions is based on a multicast tree of clients rooted at the server S. A more sophisticated solution named SplitStream [11] involves organizing clients into an overlay graph based on multiple parallel trees. Possibly the most widely used solution, named BitTorrent [8], is based on unstructured graphs. BitTorrent client nodes communicate in a random overlay graph, exchanging data whenever possible.

This plethora of solutions leaves us with two questions. Given a large file to distribute to a large number of clients:

- What is the optimal solution that minimizes the time taken for *all* clients to receive the file?

- How well do known strategies for content distribution perform compared to the optimal algorithm?

Chapter 3 addresses these questions. We begin with a simple homogeneous bandwidth model that we define in Section 3.1. By analytical means, we derive the optimal solution and establish its completion time. This optimal algorithm is based on clients communicating over a hypercube-like overlay network. The file is divided into blocks. Clients receive blocks from the server and other clients. Our contribution here is a optimal schedule of block transmissions between clients. We call this the "hypercube algorithm". We also show, by analysis, that this optimal solution is faster and simpler than related structured overlay solutions like multicast and SplitStream.

We then consider adapting our optimal algorithm to a heterogeneous client-bandwidth model. The deterministic and synchronous nature of the hypercube algorithm renders it ill-suited towards this heterogeneous scenario. Therefore, we adapt the distribution algorithm accordingly, to utilize random graphs and random matching. We first investigate the performance of this "randomized algorithm" in the simple homogeneous model, by simulations. The purpose is to find the price we pay for moving from a deterministic optimal algorithm to a randomized algorithm. We find that the randomized algorithm's completion time is within 2% of the optimal algorithm. This is detailed in Section 3.2.

In Section 3.3, we tailor the basic randomized algorithm of Section 3.2 towards several different heterogeneous scenarios. We achieve this using a heuristic scheme for client matching, where each client tries to ensure that its neighbors have useful blocks to serve other clients. We evaluate this scheme by simulations and find that it significantly reduces completion time, in comparison with other known alternatives. This reduction is by as much as 50%, particularly in cases where the client bandwidths exhibit topological or geographical clustering.

Once we exhaust the field of cooperative clients, we proceed to the study of non-cooperative clients.

1.1.2 The Non-cooperative Case - *What Price Barter?*

In Chapter 4, we change the assumption of client cooperation. We now assume that clients are not willing to freely upload data to other clients. Therefore, they need to be incentivized to upload. We study different mechanisms for this purpose, loosely based on the principle of barter. That is, a client does not upload data to another client unless it receives data in return. It turns out that the exact definition of the barter model has a big impact on the efficiency of content distribution.

Our objective here is to explore the three-way trade-off between the mechanisms' enforceability, their ability to incentivize uploads, and the efficiency of content distribution. To this end, we consider two different models based on barter. We informally analyze their incentive structure and formally analyze performance bounds. We then develop actual algorithms for content distribution under the constraints of these models.

The first incentive model is the *strict barter* model, investigated in Section 4.2. One client transfers data to another only if it simultaneously receives an equal amount of data in return. We analyze this model, and derive an optimal solution. This solution is exponentially worse than the optimal cooperative solution from Section 3.1.

The second incentive model we investigate is the *credit-limited barter* model, in Section 4.3. Here, one client transfers data to another only if the difference in the data transmitted in each direction between the two clients is less than some threshold. Our analysis of this model indicates that theoretically feasible algorithms that are as efficient as the optimal cooperative algorithms can exist, in many cases. We adapt the randomized algorithm of Section 3.2 to operate under this model, and evaluate this "randomized barter" algorithm by simulations.

We discover that randomized algorithms operating under the barter model are extremely sensitive to parameters such as the degree of the overlay network they operate on. Our simulations shed light on the critical value of the overlay-network degree, as well as the importance of using the right block-selection policy. Specifically, we find that the "rarest-first" policy, explained in Section 3.2, is crucial to good performance under this barter model. Given sufficiently high graph degree and block selection policy, the randomized barter algorithm performs nearly as well as the optimal cooperative solution.

Prior research has often focused on game-theoretic analysis of different mechanisms to identify the optimal strategy for selfish nodes. This is *not* a subject of this dissertation. Rather, our contribution is to understand the performance of certain incentive-based distribution mechanisms, assuming that nodes obey the incentive mechanism.

BitTorrent also has an incentive mechanism, defined in a public-domain specification [8]. The relationship between neighboring clients, in this mechanism, is somewhat loosely defined. In contrast, our barter-based mechanisms enforce well-defined relationships between nodes, and can be made to perform well, as mentioned above. The performance of BitTorrent however, in terms of completion time, is not something that is well-understood. Therefore we conduct a performance study of BitTorrent, via simulation, investigating the impact of different parameters. We find that BitTorrent's completion time is typically about a factor of 2.2 higher than the optimal cooperative solution. It can be improved to a factor of 1.3, if carefully tuned. This tuning, however, involves a parameter that could adversely affect the incentive structure of BitTorrent. This parameter involves the frequency at which a BitTorrent node prioritizes the neighbors to which it uploads data to. These results are detailed in Section 4.4.

Once we finish exploring different non-cooperative solutions, we proceed to the final region of our design space, involving ordered data delivery. This is crucial for supporting multimedia applications.

1.2 Customizing the delivery schedule - *a Layered Architecture*

The chapters we have described so far assumed that the data could be delivered in any order. However, several applications, especially involving video content, would benefit from the ability to customize the order in which blocks are delivered. For example, watching a live video stream of a music concert, one has little desire to receive any data pertaining to past points in the stream. On the other hand, when downloading a software package to use later, the order of arrival is irrelevant and the download's total time is more important. Several other application-specific scenarios exist, each with a slightly different feature demanded of the data delivery method.

These additional features may come at a price: an increase in total download time. Different application scenarios might be suited to different points in this space of trade-offs. Our approach is to propose an architectural division of the distribution mechanism into two layers: one application-dependent, and the other, independent. This architecture is proposed and investigated in Chapter 5.

The lower layer is the application-independent delivery layer, which focuses on reducing the total distribution time using the methods for bulk content distribution from Chapter 3. This layer provides an interface to the upper layer which enables the upper-layer to signal its data delivery preferences in a simple but powerful way.

The upper layer is the application-aware layer which decides the preferences for the order of data delivery, if any. For example, suppose a user was downloading a video to be stored on her disk, but also wanted to start watching the video as soon as possible. This layer would be the place where such preferences are recognized and communicated to the lower layer, via the afore-mentioned interface.

The data being distributed is fragmented into an ordered list of blocks. Communicating across *the interface* between the two layers simply consists of indicating a block priority. This is in addition to basic parameters involved even in bulk data distribution, like the file name and source. We argue that this simple interface is able to handle a wide range of application semantics. While is not proved, we consider a wide range of example application behaviors in support of this claim.

To evaluate this proposal, we consider two different applications. For each application, we describe how they can be achieved with our architecture, and evaluate the performance of our solution by simulations. in both cases, the solutions involve a sliding window of high-priority blocks. The results in both cases indicate significant improvement over existing alternatives. The two applications we consider are:

- **IBULK:** In-order delivery of bulk data, where the user downloads bulk data, but can obtain additional benefit if the data arrives in order. While overall completion time is still a metric of interest, we will also define a rate-based metric that captures the extent to which the data is ordered. This application is studied in Section 5.2.1
- **ISTREAM:** Fixed-rate streaming media, where the data has to arrive in order at a rate equal to or greater than some minimum rate. We need to minimize the number of losses or gaps in the arriving stream. This application is studied in Section 5.2.2

This concludes the introduction of the different facets of our work.

1.3 Summary and Roadmap to the Dissertation

Our dissertation considers the problem of distributing a large file to a large static set of clients in the minimum overall time, or completion time. Using a combination of analysis and simulation, we find that currently used content distribution algorithms are sub-optimal for this problem. We design a new algorithm based on a hypercube overlay network, and prove that it is optimal for a simple homogeneous client-bandwidth model. We also propose a randomized algorithm that we optimize for three different scenarios: (a) heterogeneous bandwidths, (b) non-cooperative clients, and (c) application-specific ordering of data delivery. Simulations indicate that our randomized algorithms improve completion time by a significant factor when compared to prior work, including BitTorrent. This factor can be as high as two in several scenarios.

The remainder of this dissertation is organized as follows.

We provide a brief overview on existing data distribution techniques related to our work, in Chapter 2. This chapter also discusses performance studies and analyses covering areas related to this dissertation. In addition to providing background on content distribution research, this chapter also highlights differences between existing work and our dissertation, where relevant.

The next three chapters comprise our technical investigation of different client scenarios:

- Chapter 3 covers our research on bulk content distribution in the *cooperative case* - for both homogeneous and heterogeneous scenarios.
- Chapter 4 covers the *non-cooperative case*, including analysis, proposal and simulation of barter-based schemes; as well as our study of BitTorrent.
- Chapter 5 describes our two-layer architecture for incorporating *application-*

specific preferences about data delivery - and includes two specific application case studies.

Finally, Chapter 6 summarizes our findings, and concludes with a discussion of open problems and future work.

Chapter 2

Background and Related Work

The last three decades have seen many scenarios with the need to distribute content to a large number of clients. These scenarios have varied widely in their requirements and parameters, e.g., the size of the content, the number of clients, the metric of interest, etc. As a result, several different techniques have been developed which address some variant of the content distribution problem. Some background on the methods used are essential in understanding the context of our research. We provide such background and context in Section 2.1.

Several researchers have measured and analyzed existing algorithms for content distribution, as well as incentive schemes. In Section 2.2, we briefly discuss the important studies that relate to our work. Where relevant, we also highlight how our research differs from the prior work.

2.1 Background on Content Distribution Techniques

We now discuss the different methods that have historically been used for content distribution. We begin with the simplest method and proceed to more sophisticated proposals.

2.1.1 Simple Client-Server Method

The most natural and least sophisticated method for distributing content to many clients is simply to have the clients all get the content from the same server. This implies that the server's upload capacity would have to be increased to handle an increase in the number of clients. For example, Google Video [30] and YouTube [64] have large server farms that have high upload bandwidths, allowing them to serve a large number of users. Smaller companies and individual users might find such a system too expensive. Alternatively, the size of the content could be reduced by downgrading the quality, as suggested by the authors of [24] and [31]. While acceptable to users of small mobile devices, such quality degradation would be undesirable to stationary users with desktops or laptops who pay for content.

2.1.2 Caches and CDNs

A natural alternative to the client-server approach is to use one or more mirrors or caches of the original data, to share the load of the server. Additional techniques[36, 12] are employed to ensure that mirrors are selected efficiently, and network and server load is reduced to the greatest extent. While this has been useful in many instances[55], the number or capacity of the mirrors must grow linearly with the demand for clients. Further, if the number of mirrors is large, the task of distributing high-volume content to the mirrors is a significant challenge. This is, in fact, an application of the problem addressed by our dissertation.

In the last ten years, a few large commercial CDNs, like Akamai [2], have sprung into growth. They operate large networks of servers which act as mirrors or caches for commercial clients. The goal is primarily to enable their clients to serve a large geographically distributed client base with very low latency or response times. The data being served is mostly low volume, such as image files and web pages. This is in contrast with our research goals, which involve files on the order of several hundred megabytes. Consequently, latency is much less of an issue than the overall distribution bandwidth, or distribution time. While these cache networks can also be used to serve large files, the issue of efficiently distributing these files from the source to the caches still remains.

2.1.3 Tree-based Multicast

The concept of multicast has long been studied in the Internet routing community. The goal is to send a single packet of data efficiently to many destinations. This function can be implemented in many ways. IP Multicast, first proposed by Deering et al. [18], implements this function at routers, in the network layer. Alternative proposals, like End System Multicast [32], Scattercast [13], and Yoid [25], implement the multicast function at the application layer. Most techniques usually achieve this function by organizing the routers or end-hosts into a tree-shaped topology, with the data source at the root of the tree. Subsequently, packets are transmitted in order from each parent-node in the tree to all its child-nodes, replicated as necessary.

IP Multicast protocols [21, 1, 62] organize routers into such a tree. Data packets are replicated at routers, and traverse physical links no more than one time each.

Thus, the load on the source and clients, as well as network links, is greatly reduced, compared to the simple client-server method. However, such methods are difficult to deploy large-scale due to the difficulty in incremental deployment, as well as due to flaws in security and usage models.

Application-layer multicast proposals attempt to address some of the issues with IP multicast. They propose that end-hosts organize themselves into overlay trees or similar structures, along which data packets are replicated and forwarded. This is amenable to incremental deployment, and allows application-specific security and semantics. A significant portion of the clients actually use their upload capacities towards serving the data. Therefore this technique reduces the load on the server. Typically, the focus of such proposals is on streaming media applications, although there are some proposals that focus on bulk content, such as Overcast [35]. We shall consider tree-based distribution in our dissertation, and find that they suffer from fundamental performance limits, compared to other alternatives.

2.1.4 SplitStream

SplitStream [11] is a proposal targeting cooperative distribution of streaming data, but can also be used for bulk data. It uses a clever arrangement of parallel multicast trees to ensure that all nodes upload data at a high capacity. It is optimized for an environment with dynamic nodes and heterogeneous bandwidths. Consequently, while it is an improvement over simple tree-based multicast, it can be sub-optimal. Further, it is a rather complex algorithm, likely requiring high maintenance in a practical setting. Recall that our focus is on optimal algorithms for static systems, with zero or low churn rates. Here, simpler algorithms can perform as well or better than Splitstream, as we shall see.

2.1.5 Peer-to-peer File-sharing

Around 1999, Napster [47, 46] emerged as an extremely popular application, allowing home users to download vast quantities of multimedia content. Typically the individual files being transmitted were music files, a few megabytes in size. This was the first popular peer-to-peer file-sharing application. A centralized search facility was provided to enable clients to find other clients with a particular file. The clients would then download files from the the other clients, instead of from some single server. This was not optimized for the situation where a large number of clients were interested in a particular large file.

The Year 2000 saw the emergence of Gnutella [29, 28], a fully distributed alternative to Napster. Gnutella organized clients in the system into a random graph. Whenever any client A wished to search for a file, it propagated a search message along several random paths along the client graph, possibly encountering clients that held the required file. The message propagation was bounded by a pre-defined radius along the client graph, as well as a limit on the number of search results. Subsequently, A could directly download the required file from one or more of the clients in the search results. Like Napster, this was not targeted for a situation where a large number of clients were interested in downloading a single large file, at the same time.

The popularity of Napster and Gnutella spawned a second generation of peer-to-peer file-sharing protocols like LimeWire [42, 41], FastTrack [19, 37], and the next generation of Gnutella [28]. These protocols categorized clients into two classes, based on their bandwidths and processing power. Those with higher bandwidths and processing power were chosen to act as hubs for routing search messages or indexing search results. Such networks were decentralized, when compared to Napster. At the same time, they were more robust to network bottlenecks than Gnutella. Further, a client could download different fragments of a file from several different clients simultaneously. However, the actual download process was still not optimized when the number of clients interested in a particular file was in the thousands. The focus of this entire family of protocols was still on improving the search mechanism for files in the client networks.

2.1.6 BitTorrent and similar protocols

BitTorrent [10, 8] is a widely used P2P content-distribution system that is based on an unstructured overlay network of end-hosts. While earlier peer-to-peer file-sharing protocols focused on scalable search mechanisms, BitTorrent aimed at scaling the actual file transfer process. Therefore it was the first popular peer-to-peer protocol that helped deliver huge files to large communities of clients, from one or a small number of sources. We now provide a quick overview of BitTorrent's operation.

Each file is divided into "pieces", in the terminology of the BitTorrent specification [8]. The specification recommends piece sizes of 128 KB, 256 KB, or 512 KB. Most implementations use 256 KB pieces. The files transferred using this system vary in size from a few tens of megabytes to over a gigabyte. The basic principle of BitTorrent is to organize the set of end-hosts interested in a particular file into a random overlay graph. The source of the file is also a node in this overlay graph. The end-hosts then obtain different pieces of a file from their neighbors on this overlay graph.

Figure 2.1 illustrates the different operations performed by a client in a BitTorrent network. Consider a network of clients, or a "torrent", downloading a particular file F. Each new client joining this network, first contacts a special node called "the tracker". This is depicted in Part (a) of Figure 2.1. The IP address of the tracker is known before-hand, usually from well-known search engines. For example, BitTorrent's official website [10] provides a search engine to find trackers corresponding to file-names.

The tracker gives the new client a list of several randomly chosen clients already downloading F. The new client then establishes a "peering" or "neighboring" relationship with the clients in that list. The number of neighbors is subject to a predefined limit, which the BitTorrent specification suggests should be 50. This results in a random overlay graph of clients. This is depicted in Part (b) of Figure 2.1. Each node informs its neighbor of the blocks it possesses, when the peering relationship is set up.

The server transmits different blocks to its neighboring clients. Every client in this graph sends a message to each of its neighbors whenever it finishes receiving a new piece. This is shown in Part (c) of Figure 2.1. If a client X is informed that its neighbor Y has a piece P, and X does not have P, then X sends a request for that piece to Y. Clients use a prioritization mechanism to decide which requests to satisfy. This mechanism ranks neighbors in the order of bandwidth *received from* them. We discuss the implementation of this mechanism in more detail in Section 4.4.

Clients then transmit pieces to a small number of requesting neighbors. This is shown in Part (d) of Figure 2.1. A node usually transmits to multiple neighbors simultaneously, up to a maximum of four. Each client continues requesting pieces from its neighbors until it has received the entire file. At this point, the client can either leave the overlay network, or continue serving other clients.

BitTorrent is widely used to distribute large files like TV episodes and movies. A large portion of the files being distributed today are probably in violation of copyright laws. However, we believe that algorithms developed in this area will be used by legitimate content providers, in the near future. As an indication of this trend, the creators of BitTorrent recently formed a commercial agreement with a major

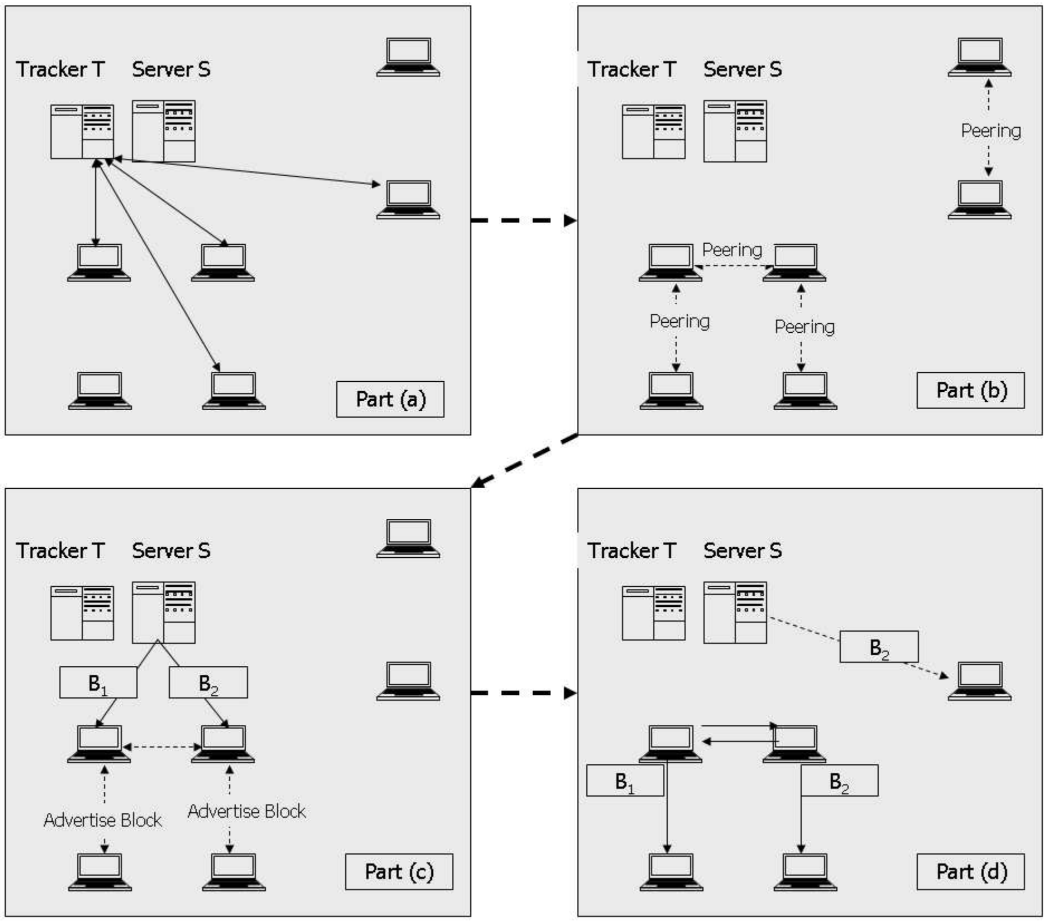


Figure 2.1. BitTorrent overview

movie studio to distribute their content [9]. Our technical focus is restricted to the investigation and improvement of the algorithms used for content distribution. We do *not* discuss legal business models or methods for ensuring legality of content. We consider such issues to be orthogonal to the distribution problem.

The goal of this dissertation is to find the best possible method for large-scale distribution. For this problem, BitTorrent appears to be the most popular solution in use. However, the performance of BitTorrent for different system sizes has not been quantified in published literature. It is not currently known if it is the best-performing algorithm that could be designed. Further, BitTorrent is optimized to deal with a very dynamic and potentially selfish client base. In other words, BitTorrent is targeted towards a high rate of arrival and departure of clients into the network or "torrent" for a particular file. In addition, the clients might not transmit files to other clients at a high rate unless given some incentive. Therefore, the design of BitTorrent makes tradeoffs that could adversely affect the distribution time in our scenarios of interest. In contrast, our focus is on static client bases, in both cooperative and non-cooperative scenarios. Therefore, a significant part of our dissertation involves quantifying the performance of BitTorrent in such scenarios, as well as comparing it to alternative distribution methods. The algorithms we propose begin with the same principles as BitTorrent, namely file fragmentation and overlay graphs. However, our algorithms are optimized towards several specific situations, and perform better than BitTorrent in those scenarios.

A protocol called CoolStreaming [66, 15] adapted the principles of BitTorrent for large-scale streaming media distribution. Clients are organized in a random overlay graph, requesting "segments" of the media stream from their neighbors. The key contribution here is the process used by a client to decide the segments to request, and the target neighbors for these requests. The segments are chosen from a small window near the currently viewed point in the stream. Preference is given to segments held by fewer neighbors. In addition, neighbors with higher bandwidth are preferentially chosen as a target for segment request. The exact algorithm is published in [66].

Several programs for peer-to-peer streaming have been developed recently [61, 60, 56]. They are mainly used to view television channels that are not easily available in the user's geographic location. They typically use techniques based on application-layer multicast or CoolStreaming, but the details of these algorithms are proprietary and unpublished. One of our goals is to optimize content distribution for applications which depend on the order of data delivery. In solving this goal, we will draw comparisons to CoolStreaming's distribution techniques, in Chapter 5. Our goals do not, however, encompass the design and analysis of protocols used for multimedia encoding and rendering.

Several other content distribution algorithms [7, 27] have been proposed that are based on the principles of BitTorrent. These proposals are tailored towards network locality, robustness or ability to handle rapid peer arrivals/departures. The published performance for these algorithms are worse than the optimal algorithms we propose, when the overall completion time is the metric of interest.

2.2 Related Studies of Distribution Algorithms and Incentive Schemes

One of our goals was to optimize large-scale content distribution in scenarios where clients would require incentives for cooperation. Therefore, we survey prior work on incentive schemes and non-cooperative content distribution in Section 2.2.1.

Several researchers have analyzed or measured existing methods of content distribution. We describe related studies and compare them to our work, in Section 2.2.2.

2.2.1 Background on Incentive Schemes for Non-cooperative Scenarios

Several practical content distribution applications involve clients that cannot expect cooperation from each other. Each client is concerned primarily with its own download performance, and not with that of the other clients. Therefore, to ensure reasonable overall performance, clients need some incentive to upload data to others. We will now provide a brief overview of prior work on incentive schemes for such scenarios.

A qualitative discussion of the taxonomy of general incentive patterns can be found in [50]. In the context of peer-to-peer networks, there have been studies on the effectiveness of reputation systems and incentive mechanisms [39]. Their focus is on encouraging clients to share more files. In contrast, we consider the download of a single large file. Therefore, we need to develop schemes encouraging clients to transmit blocks at high rates to their neighbors.

This problem can be solved by leveraging the idea of "barter". Here, a node gives data only in exchange for receiving some data. Barter models have been discussed in the context of other resources like computation [14] and storage [16]. However, these resources are not amenable to instant exchange. Our focus is on bartering actual content, which poses a different set of problems. The authors of [49] sketch an incentive mechanism which uses the notion of "credit thresholds" to enforce fair sharing of bandwidth. We will use a similar idea. But when completion time is the metric of interest, the performance of such algorithms has not been adequately studied in prior work. Quantifying this performance is one of the contributions of our dissertation.

A mechanism for "cyclic" or transitive barter has been evaluated via simulations in [3]. Indivisible objects are bartered in a complete graph overlay. The authors' metric of interest is the mean download time per object. In contrast, our problem involves all nodes requiring *all* blocks, and seeks to optimize the total time required to achieve this. We briefly consider one our proposed algorithms in the context of cyclic barter. However, a detailed treatment of cyclic barter is outside the scope of this dissertation.

An alternative to the barter system is the use of an electronic currency system. Such mechanisms [59, 52] typically involve a greater degree of complexity and centralization than pairwise barter systems. Hence we restrict our research to barter-based mechanisms. The rationale behind this choice is further discussed in Chapter 4.

2.2.2 Related Studies of Content Distribution Algorithms and Incentive Schemes

There have been numerous studies related to ours, involving analysis and performance estimation of content distribution algorithms. We describe them next.

Prior publications [63, 54] have described a theoretical lower bound on content distribution for a simple homogeneous client-bandwidth model. These studies also outline algorithms for this scenario that are optimal only for certain client-set sizes. Therefore the optimal algorithm for arbitrary client-set sizes is left an open challenge by these studies.

We recently discovered two other solutions from the literature on discrete mathematics, which can be adapted to content distribution in the homogeneous scenario. The first is a *near*-optimal hypercube-based approach in [5], and the second is a different optimal algorithm in [6]. These two algorithms were proposed for message broadcast among an arbitrary number of processors. In addition, a recent technical report [45] provides an optimal solution for cooperative content distribution with homogeneous client bandwidths. These solutions are for arbitrary values of n. However, the optimal algorithms proposed are rather complex. More importantly, no bounds are demonstrated on the degree of the resulting overlay connectivity graph between clients. This is a weakness, since high graph degrees lead to high messaging overhead. The impact of graph degree is further explained in Chapter 3.

We consider the same homogeneous distribution problem, and derive an optimal algorithm based on a hypercube embedding. This is simpler than the solutions reported in [45] and [6]. In addition, our approach has a provably low degree bound. The authors of [45] also simulated a randomized content distribution algorithm. They found that the finish times depend linearly on the file size and on the logarithm of the number of client nodes. In our work, we propose and simulate a very similar algorithm, and our results are in agreement with [45]. However, our simulations are over a much wider range of file sizes. In addition, we propose several extensions to this randomized algorithm for heterogeneous client bandwidths and ordered delivery, which distinguish our work.

In [17, 63, 54], models of BitTorrent are developed to analyze the evolution of the system upload bandwidth as nodes join and leave. Completion time is also considered. These analyses use simplifying assumptions about the content at different nodes. For example, they assume a pre-defined efficiency of content distribution, or assume that it is optimal [54]. This approach is orthogonal to ours, where we focus on a static set of nodes. Further, we specifically seek to understand how to optimize the efficiency of content distribution, rather than assume a certain level of efficiency.

The authors of [54] also analyze the incentive structure of BitTorrent as a "bandwidth game". They show that all nodes upload at peak capacity in a homogeneous system. Their model assumes that (a) nodes can only make a one-time choice determining the upload capacity to use, and (b) every pair of nodes always has useful data to exchange. Our goals and thesis are different, in two ways. Firstly, we do not use the listed assumptions in our analysis of any algorithm, since we wish to understand the complexities arising from individual nodes' content. Secondly, we do not focus on game-theoretic analysis of incentive schemes. Rather, we aim to quantify the best possible performance for a given incentive scheme, assuming the clients all follow that scheme.

2.3 Summary of Prior Work

In our survey of prior work, we have described content distribution methods for a wide range of application domains. We find that the problem of distributing large files to a large static client set in the minimum overall time has not been adequately studied. While this was not a common scenario in the past, the growth in content consumption foreshadows the importance of this problem domain in the near future. Therefore, this is the focus of our dissertation. In solving this problem, we encounter several challenges that have not been addressed in prior work. We summarize these challenges and our contributions next.

While numerous distribution algorithms exist, there is no clear understanding of their performance relative to each other, and relative to system sizes. Therefore, we first define a performance model parameterized by file size and number of clients, with completion time as the metric. Our model, while simple, enables useful comparison of different algorithms.

Algorithm performance is heavily influenced by assumptions made about client cooperation, bandwidth models, and the order of data delivery. In contrast to prior work, we optimize our distribution algorithms for each resulting scenario. These scenarios, and their associated challenges, are enumerated below.

Cooperative Clients and Bulk Data:

- We design a provably *optimal* algorithm for a simple homogeneous client-bandwidth scenario. To our knowledge, the only prior solutions for this problem are described in [6] and [45]. Our algorithm is distinguished by a simpler formulation, as well as bounded overlay graph degrees.
- While BitTorrent is widely used, its completion time behavior remains poorly understood. We quantify the performance of BitTorrent, in the context of our performance model, via simulation. Its completion time is worse than the optimal completion time by a factor of 2.2, approximately, in a simple homogeneous scenario. Our research also reveals ways to improve BitTorrent's completion time to near-optimal values.
- For a heterogeneous client-bandwidth model, we propose a randomized distribution algorithm, based on unstructured overlay graphs. These basic principles are already in use, by BitTorrent. However, BitTorrent's design trades performance for the ability to handle non-cooperative clients, thus leaving the study of cooperative clients incomplete. Therefore, we optimize our randomized algorithm for the cooperative scenario, augmented by a novel heuristic for heterogeneous bandwidth environments. We find, via simulation, that our heuristic significantly improves completion time compared to BitTorrent, by a factor between 1.25 and 2.

Non-Cooperative Clients and Bulk Data:

- The concept of barter has been studied in many contexts, e.g., to encourage exchange of *different files* in a peer-to-peer file-sharing system. However, bartering fragments of a single large file under distribution, and the resulting completion time, has not yet been investigated. This is our focus.

- BitTorrent is the most widely used solution for the non-cooperative scenario. However, is not optimized for a static client base. Therefore, we augment our randomized algorithm with a "credit-limited" barter model. We find, via simulation, that we can obtain near-optimal performance from this algorithm for a homogeneous bandwidth model.

Ordered Data Delivery:

- Consider applications involving high-quality multimedia, where the delivery order of data fragments is important. Large-scale delivery methods for this problem are currently optimized for specific application details and data rates. They are not designed to obtain the maximum possible rates, or to handle small changes in application semantics. To address this issue, we propose an architecture that separates application-specific details from the application-independent functions of content distribution. The interface between the two layers is based on block priorities.
- To demonstrate the flexibility of our architecture, we demonstrate how it can be adapted for two different candidate applications. In addition to flexibility, our solutions also provide an improvement in performance, which we estimate via simulation. When compared to alternative known methods, our proposals can support data rates that are 25% higher.

We have presented background on content distribution. We have also described challenges that are yet to be addressed by prior work. We now commence our investigation of these challenges, starting with the simplest part of the design space.

Chapter 3

Cooperative Bulk Content Distribution

Our goals are to investigate the performance of algorithms for content distribution in several different scenarios. Each of these represents a different set of assumptions regarding client cooperation, the network environment, and the data being distributed. We have categorized these scenarios by a design space involving the following three choices:

- Clients can be cooperative or non-cooperative.
- The order of delivery of data fragments to the application matters or is immaterial.
- The clients' bandwidth distribution can be either homogeneous or heterogeneous.

These choices, along each dimension, are mutually exclusive and exhaustive. Figure 3.1 depicts this design space, and the decision tree that governs our exploration of this space. In this chapter, we explore a subset of the scenarios described by our taxonomy, involving cooperative clients and bulk data. The other chapters explore the remaining scenarios.

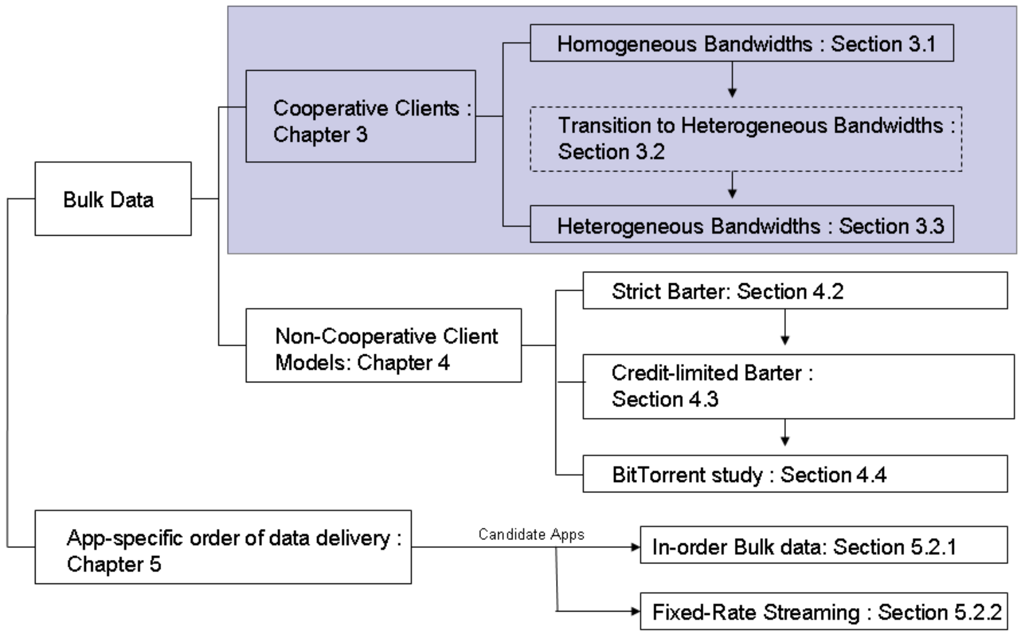

Bulk Data
Cooperative Clients : Chapter 3
Homogeneous Bandwidths : Section 3.1
Transition to Heterogeneous Bandwidths : Section 3.2
Heterogeneous Bandwidths : Section 3.3
Non-Cooperative Client Models: Chapter 4
Strict Barter: Section 4.2
Credit-limited Barter : Section 4.3
BitTorrent study : Section 4.4
App-specific order of data delivery : Chapter 5
Candidate Apps
In-order Bulk data: Section 5.2.1
Fixed-Rate Streaming : Section 5.2.2

Figure 3.1. Design Space and Roadmap of Study

We consider the subset where clients are cooperative and the data is bulk data. The third dimension, regarding client bandwidths, has not yet been fixed. In general, the client bandwidth distribution is not known. We begin with a special case. This assumes the simplest possible distribution, one in which the clients identical bandwidth. We treat this case separately, since this allows the most flexibility in our algorithm design and analysis. The solutions available for this case are likely to be better than those under more general bandwidth assumptions. Therefore, a real-world scenario with homogeneous bandwidths would benefit from our special-case analysis. In addition, the theoretical results from this special case serve as a useful baseline for comparison of every other algorithm discussed in our dissertation.

We then proceed to the more general case, of heterogeneous client bandwidths. The shaded area of Figure 3.1 is the part of the design space covered in this chapter.

Cooperative Homogeneous Bulk content distribution- *An Introduction*

We begin with the simplest possible set of assumptions: the clients are cooperative and homogeneous, and the application does not depend on the order of delivery of data fragments. The assumption of cooperation implies that no incentives are required for clients to send data to each other at the highest possible rate. The assumption of homogeneity means that the transmission bandwidths of the clients and the server to other clients are all the same. We name this the Cooperative Homogeneous Bulk content distribution scenario, and investigate it in Section 3.1.

Our study begins by defining a simple system model. This serves as a useful framework to define and analyze our problem, as well as several existing solutions to our problem. Our metric of interest is "completion time", or the total time taken for all clients to receive the entire file being distributed. Our analysis yields a new algorithm for content distribution, based on a hypercube overlay network of client nodes. We prove analytically that this "hypercube" algorithm provides the *optimal* completion time. We find that our hypercube algorithm is significantly faster than existing known algorithms like multicast trees and BitTorrent [1]. This difference is typically around a factor of two.

In Section 3.2, we *begin the transition* to a heterogeneous scenario. The deterministic synchronous nature of the hypercube algorithm is not suitable for heterogeneous scenarios, demanding a less rigidly structured non-deterministic algorithm. We present the basic template for such a "randomized" algorithm in Section 3.2. While *still assuming a homogeneous client set*, we explore its performance by simulations. This is primarily to quantify the price paid, in performance, for the lack of structure and deterministic schedule, in comparison with the provably optimal algorithm of Section 3.1. We find this price to be insignificant, since the randomized algorithm's completion time is within a factor of 2% of the optimal the algorithm.

Cooperative Heterogeneous Bulk content distribution - *An Introduction*

In Section 3.3, we remove the assumption of client homogeneity and examine different models of heterogeneous client bandwidth conditions. We motivate the need to make our distribution algorithm bandwidth-aware. We develop an extension of the randomized algorithm from Section 3.2 tailored to these heterogeneous conditions. It is based on a heuristic to ensure a high degree of utilization of clients' upload bandwidth capacity. That is, each client node tries to ensure that its neighbors have useful blocks to serve other nodes. We call this algorithm "$HRAND$", and estimate its completion time via simulations. We compared this to other existing solutions, including an approximation of BitTorrent. $HRAND$ is found to reduce the completion time by a factor between 1.25 and 2, depending on the specific system parameters simulated. The detailed results are presented in Section 3.3.

[1] We compared our hypercube algorithm's performance to an estimate of BitTorrent's performance, that we obtained by performing simulations - these BitTorrent simulations are described in detail in a different chapter (Chapter 4), and only the relevant results are referenced here in Section 3.1.

Finally, Section 3.4 concludes and summarizes our work on cooperative bulk content distribution. We now proceed to the details of our research into the Cooperative Homogeneous Bulk distribution scenario.

3.1 Cooperative Homogeneous Bulk Distribution

In this section, we study existing algorithms and develop new ones for content distribution in a simple scenario. Our core assumption is that clients are homogeneous in terms of their data transmission rates, and are always willing to upload data at their maximum upload bandwidth. Our main goal is to answer the following:

- What is the best possible performance for a content distribution algorithm? Can we design an algorithm to achieve this performance?
- How well do different currently known content distribution algorithms perform under this model? How do they compare with the best possible algorithm?

To better define our goals and assumptions, we first describe our general system model in Section 3.1.1. We also present a simplified version representing the cooperative homogeneous scenario. We restrict our study to the simplified version of the system model in Section 3.1. Our model involves the characterization of any content distribution algorithm's performance in terms of a few simple parameters. These parameters represent the total distribution time, the number of clients in the system, and the size of the file. This characterization, while simple, has not been adequately studied in previous research.

We analyze several commonly known algorithms, like tree-based multicast, using this model, in Section 3.1.2.

In Section 3.1.3, we analyze our system model to obtain a provably optimal algorithm. We derive an algorithm that we prove minimizes the completion time. This algorithm is based on a hypercube overlay network of client nodes. Its performance serves as a useful basis of comparison for other algorithms discussed subsequently in this dissertation, including BitTorrent. Such comparison requires estimates of how the other algorithms perform. Section 3.1.2 provides analytical estimates for some known algorithms, not including BitTorrent. We performed simulations of BitTorrent to quantify its completion time behavior. The relevant results from these simulations are referenced here, but presented in detail in Chapter 4. This is because we categorize BitTorrent as non-cooperative content distribution, and analyze that class of algorithms in the next chapter.

3.1.1 Model

When designing content distribution methods, several metrics are possible. Examples from related work are the latency of access to small files and the average download time of larger files. First, we specify our focus. In particular, we specify the different components involved in the overall system we consider, the interactions that are possible between them, and any constraints on these interactions. Since these interactions will involve data transfers, we define the level of accuracy and detail of interest when considering such data transfers. Furthermore, our system model specifies a single metric of interest, and provides a problem statement based solely on these system components and this metric.

System models can have varying degrees of complexity and realism. For example, a model can be sufficiently detailed for analysis of individual packet delays and physical network-level hops. However, if we assume that the file sizes involved are very large, and any node can connect to any other node if required, a model that captures file-block level transfers is sufficiently fine-grained for our purpose. We can use this to analyze the completion time of distribution algorithms.

We now define the model of client bandwidth and data transfer used throughout this dissertation. This model, while simple, helps us compare different content distribution techniques and their scaling behavior.

We have a server S and a set of clients C. For notational convenience, we assume that there are $n-1$ clients $C_1, C_2, \ldots C_{n-1}$, for a total of n nodes including the server. C_0 is the same as the server S. Any of these nodes can establish connectivity to any other node using some universally available underlying network mechanisms. For example, they could use standard protocols based on TCP/IP.

Bandwidth Model We will begin with the *"simple homogeneous model"*, which assumes that all nodes, including the server, have the same upload bandwidth U and the same download bandwidth D, with $D \geq U$. Therefore, any node can transmit a block to any other node at bandwidth U. Most real-life scenarios satisfy our assumption that download bandwidth is not exceeded by upload bandwidths.

This model also assumes that transmission bottlenecks are always at the tail links. That is, the effective transfer bandwidth from node X to node Y is the minimum of X's available upload bandwidth and Y's available download bandwidth. This equals U. While this bandwidth model is simple, it is still useful for reasoning about the behavior of different algorithms.

In Section 3.3, we will investigate content distribution under a more generalized "heterogeneous" model. Each node C_i, including the server, has the capacity to transmit to another node C_j at a transmission rate of $U_i j$. In other words, the transmission bandwidth may differ based on both source and destination of the transmission.

Data-Transfer Model A data transfer from a node X to a node Y must involve a minimum quantum of B bytes, which we call a *block*. The block size B must be large enough to ensure that (a) the entire available bandwidth is saturated by the transmission, and (b) the propagation delay and the transmission start-up time are much smaller than the transmission time. A node cannot begin transmitting a block until it has received that block in its entirety.

We assume that any client or server can communicate with and transfer data to any other client or server. That is, nodes can establish overlay connectivity to any other node, using standard underlying network-layer protocols. We cannot assume that the latency of communication between any two nodes is the same. We do assume that the size of typically transmitted data is large enough to render such latency differences insignificant. Bandwidth is the only transmission characteristic that matters.

Time Unit For notational convenience, we choose one time unit, or one *tick* to be the time taken for some particular node to transmit a block at full capacity to some other given node. If we break down any distribution algorithm into a series of "rounds" or "stages" taking the same time, then a tick is the time taken for each of these rounds. This choice of time unit allows us to capture the relationship between distribution time, file sizes and transmission bandwidths, without specifying a separate metric for each.

In the simple homogeneous model, B/U will equal one tick, so that each node can transmit at the rate of one block per tick. Later, in the heterogeneous model, a tick is defined as the time taken for some chosen node to transmit a block at full capacity to some other chosen node. Any two nodes can be chosen for this purpose without affecting correctness. For notational convenience, we choose the pair of nodes that have the fastest block transfer time.

File Blocks Finally, we let the file F, which is to be transmitted from the server, consist of exactly k blocks $B_1, B_2, \ldots B_k$, of size B. We ignore round-off issues that may make the last block smaller.

Problem What is the best way to organize data transfers to ensure that all $n-1$ clients receive file F at the earliest possible time? In other words, if client C_i receives the complete file at time t_i, we want to minimize $\max_i(t_i)$, also termed as the *"completion time"*. The focus on completion time distinguishes our work from prior studies and algorithms that considered other metrics such as average download time or latency.

Throughout this dissertation, we assume that there are no failures in the network or at the clients or server. The design of features that provide robustness to such failures is outside our scope. Throughout the rest of this section, we will assume the simple homogeneous model.

3.1.2 Completion Times of Some Well-known Algorithms

To illustrate the use of our model, we analyze some well-known algorithms. These were introduced in Chapter 2. We assume the simple homogeneous model and the associated problem parameters k and n.

3.1.2.1 1-to-Many Unicast.

The first algorithm we consider, and the least sophisticated, is for the server to transmit the file of interest to each client, either in serial or in parallel. Trivially, we observe that the completion time for this strategy is $k * n$. This implies that either the server bandwidth or the completion time grows proportionally with the size of the client set. This is not a scalable solution.

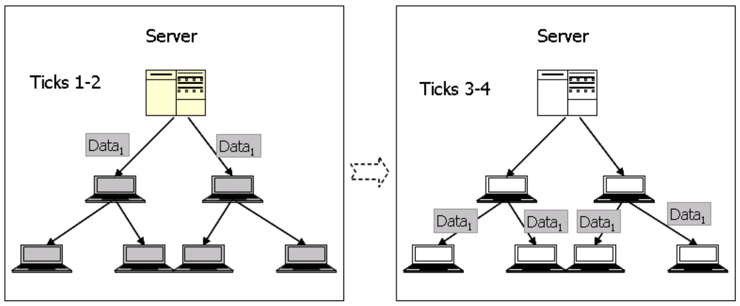


Figure 3.2. Binary Multicast Tree distribution for 7 nodes and 1 block

3.1.2.2 The Pipeline.

Another simple solution is to pipeline the download. That is, S sends the file, block by block, to C_1, which pipelines it to C_2 and so on. Assuming that each stage of this pipeline involves the complete transmission of one block, the completion time for this strategy is $k + n - 1$ ticks. Observe that it takes k ticks to get all k blocks out of the server, and a further $n - 1$ ticks for the last block to propagate to the last client. While better than 1-to-Many Unicast, this algorithm still exhibits completion time growth proportional to the number of clients, if n is large compared to k.

3.1.2.3 A Multicast Tree.

In Chapter 2, we described several proposals for overlay multicast that relied on arranging clients into a tree-based structure, and transmitted the data blocks down the tree. We now analyze the algorithmic complexity of this method. Consider all n nodes to be in a d-ary multicast tree ($d > 1$) with server S at the root. Figure 3.2 shows an example of such a tree, where $d = 2$.

In this algorithm, each node simultaneously transmits the same block to both its neighbors, before transmitting the next block, and so on. Since each node in a multicast tree can transmit data at the total rate of one block per tick, it takes d ticks for a block to be transmitted from a node to all its children. Figure 3.2 shows two phases: On the left, the root node takes two ticks to transmit block B_1 to both its child nodes. Subsequently, on the right, each child node transmits that block to each of their children, again in parallel. This takes another two ticks.

Thus, the amount of time required for S to transmit all k blocks is kd. In addition, the last block transmitted by S needs to be propagated down the tree, which requires time equal to d times the depth of the tree. Therefore, the total completion time using a $d-$ary multicast tree is equal to $d(k + \lceil \log_d(n(d-1)+1) \rceil - 2) \simeq d(k + \lceil \log_d n \rceil)$.

Since the completion time exhibits logarithmic growth with the number of nodes, the multicast tree is a more scalable method than 1-to-Many Unicast and the Pipeline. Is this the lowest possible completion time for this model? We answer this question in the following sub-sections.

3.1.3 Analysis of the Simple Homogeneous Model

We now analyze the simple homogeneous model, to derive the optimal completion time, as well as an algorithm with that completion time. We begin with a simple algorithm for the fastest way to get a single block from the server to all clients. After solving this simple problem, we increase its complexity step-by-step. We finally arrive at a solution for distributing k blocks among n nodes, for *arbitrary* values of k and n, in the simple homogeneous model.

Several of these results have been mentioned in prior related work, discussed in detail in Chapter 2. But it is necessary to present a full treatment here to facilitate a better understanding of our solution to prove its optimality.

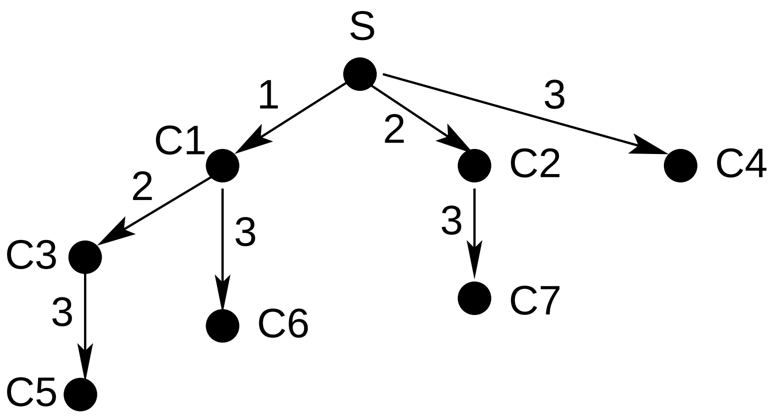


Figure 3.3. A binomial tree with $n = 8$. Edges labeled with the tick where they are used.

3.1.3.1 The Binomial Tree.

Consider the case when the file F consists of exactly one block, i.e., $k = 1$. We can use the following strategy, depicted in Figure 3.3. During the first tick, S sends the block to C_1. In the second tick, S and C_1 transmit to C_2 and C_3 respectively. In the next tick, all four of these nodes transmit to four new nodes, and so on, thus *doubling the number of completed nodes* at each tick. The resultant pattern of data transmission forms a binomial tree, as seen in the figure. It is not possible to grow the number of nodes that possess the block by more than a factor of two during each tick, due the constraints of the homogeneous model. Therefore, this is the optimal strategy for distributing one block. The completion time for this strategy is $\lceil \log n \rceil$, and this is optimal for the case $k = 1$.

When $k > 1$, a simple way to extend the binomial tree strategy is to send the file one block at a time, waiting until a block finishes before initiating transfer of the next block. This strategy has a completion time of $k \lceil \log n \rceil$[2]. As we shall see, this can be significantly improved.

[2]All logarithms are to base 2 unless otherwise noted.

3.1.3.2 A Lower Bound.

We now present the following theorem establishing a lower bound on the time required for cooperative content distribution.

Theorem 1. *Transmitting a file with k blocks to $n-1$ clients requires at least $k+\lceil\log n\rceil-1$ ticks.*

Proof. After the first $k-1$ ticks, there is still at least one block, say B_x, left in the server that is not possessed by any other node. The amount of time needed for block B_x to propagate to all nodes, starting from the server is at least $\lceil\log n\rceil$ ticks, since the number of nodes with B_x can at most double in each tick. Therefore, the overall content distribution requires at least $k+\lceil\log n\rceil-1$ ticks. □

The above lower bound is *tight*, as we will demonstrate. Neither this lower bound, nor the completion times of the simple algorithms seen so far, depend on the *download bandwidth* of nodes. So long as the download bandwidth is larger than the upload bandwidth, the system is bottlenecked by the latter.

3.1.3.3 The Binomial Pipeline

We now present the Binomial Pipeline, which achieves the optimal completion time for content distribution, given the following assumptions. After stating the simplifying assumptions, we will present the Binomial Pipeline as a three-phase algorithm, with each phase characterized by a set of rules dictating the pattern of block-transfers.

Two Simplifying Assumptions. We begin with two assumptions to explain the algorithm's intuition: (a) $n=2^l$ for some integer $l>0$, and (b) any pair of nodes can communicate with each other. Although a slightly sub-optimal solution for this special case has been introduced in prior work [63], we describe it in detail to set the stage for the remainder of this sub-section. We partition the algorithm's operation into the following three stages.

The Opening The opening phase of the algorithm ensures that each node receives a block as quickly as possible, so that the entire upload capacity in the system can begin to be utilized. This phase lasts for l ticks (where $n = 2^l$), and is characterized by two simple rules:

- During tick i, the server S transmits block B_i to a client that possesses no data.
- Each client, if it has a block before tick i, transmits that block to some client that possesses no data.

Observe that since there are 2^l nodes in total, all clients have exactly one block at the end of l ticks. The communication pattern is, in fact, the same as in the binomial tree of Figure 3.3, with all nodes in the C_1-subtree having block B_1, all nodes in the C_2-subtree having B_2 and C_4 having block B_3. In general, we can partition the $n-1$ clients into l groups, $G_1, G_2, \ldots G_l$, of sizes $2^{l-1}, 2^{l-2}, \ldots, 1$ respectively, with all nodes in G_i having B_i.

We illustrate the binomial pipeline with an example, shown in Figure 3.4, with $k = 3$, $n = 8$ and $l = 3$. This figure depicts the block transfers in each tick of the algorithm's progression. Observe the division of clients into 3 groups G_1, G_2 and G_3 of sizes 4, 2 and 1, respectively. The Opening phase is shown in the first three ticks of the figure: a client in G_1 gets the block B_1 after Tick 1; G_2 gets a block after Tick 2; and G_3 gets a block after Tick 3. As soon as a block enters G_1, that block is replicated within G_1 via the Binomial Tree from Section 3.1.3.1. Group G_2 gets a block, B_2, only after Tick 2. It finishes replicating the block within G_2 at the same time as G_1, since G_2 contains half as many clients as G_1. Thus all clients in all groups possess exactly one block at the end of three ticks.

The Middlegame After the opening, all nodes have a block, and the algorithm enters the middlegame. During this phase, the objective is to ensure that every node transmits data during every tick, so that the entire system upload capacity is utilized. One may view the middlegame algorithm as a strategy that determines, at every tick, which node transmits which block to which client.

The Binomial Pipeline ensures that $n - 1$ nodes transmit during every tick t for all $t \geq l$, while maintaining the following invariants:

- Clients are partitioned into l groups $G_{t-l+1}, G_{t-l+2}, \ldots, G_t$ with sizes $2^{l-1}, 2^{l-2}, \ldots, 1$ respectively.
- For $t - l < i \leq t$, block B_i is possessed exactly by the clients in group G_i.
- All clients have blocks $B_1, B_2, \ldots, B_{t-l}$. No client has blocks $B_{t+1}, B_{t+2}, \ldots, B_k$.

Observe that all three invariants hold after l ticks, since the opening partitions nodes into l groups with each group possessing a unique block in $B_1, \ldots, B_l$.

Again, we refer to the example of Figure 3.4, to illustrate the middlegame. After three ticks, all the clients in G_1 possess the block B_1; all clients in G_2 possess block B_2; and all clients in G_3 possess block B_3. Now, the number of clients who have only B_1 and want some other block equals the number of clients who want B_1 and can supply some other block. Therefore we can match each node inside G_1 with some node outside G_1, and perform an exchange of blocks. This occurs in the fourth tick of Figure 3.4. The arrows in the figure show the transmissions that take place: S hands off block B_4 to C_1, while nodes C_3, C_5 and C_7 are paired up with C_2, C_4 and C_6 respectively, exchanging the only blocks that they have. The consequence is the formation of a new set of three groups, shown between the fourth and fifth ticks of

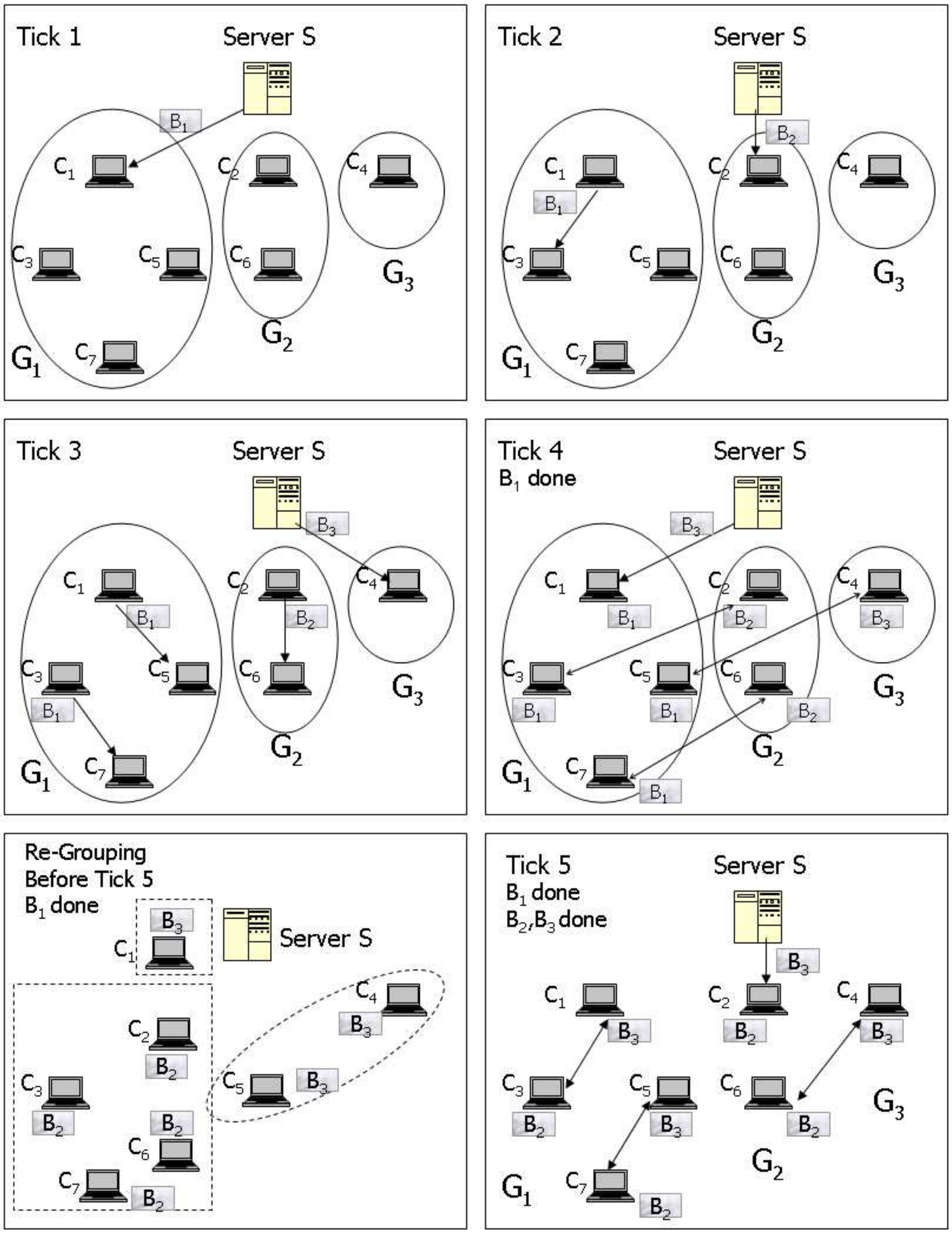


Figure 3.4. Binomial pipeline stages for 8 nodes and 3 blocks

Figure 3.4, that restores the invariants. Block B_1 is now held by all the nodes, four nodes have B_2, two have B_3, and since there are no more blocks, there is one final node C_1 with B_3. Observe that this situation is almost identical to that at the end of the third tick, with only the group names and members being different.

In general, the following transfers occur at any tick t:

- Server S selects any node in G_{t-l+1}, say C_x, and hands it B_{t+1}. C_x becomes the sole member of G_{t+1}.
- Each node in $G_{t-l+1} - \{C_x\}$ is paired with a unique node in the remaining groups, and transmits B_{t-l+1} to it, thus ensuring every node holds B_{t-l+1}.
- Similarly each node in group G_i, $t - l + 1 < i \leq t$, sends block B_i to a unique node, say C_r, in $G_1 - \{C_x\}$. Node C_r then migrates to group G_i.

The Endgame The middlegame proceeds until k ticks elapse. That is, S transmits block B_k to create a new group G_k. After this tick, the middlegame algorithm runs out of blocks for S to transmit. To work around this, we define $B_{k+\alpha} = B_k$ for all $\alpha > 0$, and let the middlegame algorithm to continue running, i.e., the server keeps transmitting the last block when it runs out of new blocks to send.

After tick $k+l-1$, all nodes have all blocks from B_1 to B_{k-1}, from the invariants. The only groups left are $G_k, G_{k+1}, \ldots, G_{k+l-1}$. But since $B_{k+\alpha} = B_k$ for all $\alpha > 0$, this implies that all nodes have B_k. This indicates that the transfers are complete[3]. Since $l = \lceil \log n \rceil$, the completion time of the Binomial pipeline, when n is a power of two, is $k + \lceil \log n \rceil - 1$. *This is optimal*, since it equals the lower bound shown in Section 3.1.3.2.

Again, we refer to the example of Figure 3.4, to illustrate the endgame. After four ticks, there are three re-organized groups, shown between the fourth and fifth ticks of

[3]Note that if $k < l$, we may once again set $B_\alpha = B_l$ for all $\alpha > l$ and proceed as usual.

Figure 3.4. Block B_1 is now held by all the nodes. Four nodes have B_2. Two nodes have B_3. Since there are no more blocks, there is one final node C_1, which has B_3. As in the middlegame, we now match all the nodes with B_2 to all the other nodes. Because there are only three blocks, we notice the Endgame rule being invoked, as the server simply transmits the final block B_3 to one of the nodes. These transfers are illustrated in Tick 5 of Figure 3.4.

3.1.3.4 A Hypercube Embedding

Our description of the Binomial Pipeline leaves open the question of exactly which nodes communicate with which others at what times. In fact, the algorithm is non-deterministic, and there are many options available at every tick. In practice, if a node has to maintain and manage connections with nearly all other nodes, the system is unlikely to scalable, with respect to n. Thus, we would ideally like our algorithm to require each node to interact only with its "neighbors" on a low-degree overlay network.

Interestingly, the Binomial Pipeline can be executed on an overlay network with degree exactly equal to l. This is the hypercube network. To explain, let us assign a unique l-bit ID to each of the n nodes, with the server node assigned the ID with all bits zero. Links in the overlay network are determined by the hypercube rules applied to node IDs. That is, two nodes form a link if and only if their IDs differ in exactly one bit. Define the *dimension-i link* of a node to be its link to the node whose ID differs in the $(i+1)^{\text{st}}$ most significant bit. The Binomial Pipeline is then summarized as below.

During the t^{th} tick, for $1 \leq t \leq k+l-1$, each node X uses the following rules to determine its actions:

- X transmits data on its dimension-($t \bmod l$) link.

- If $X = S$, it transmits block B_t. ($B_t = B_k$ if $t > k$.)
- Otherwise, X transmits the highest-index block that it has, i.e., the block B_i with the largest value of i. If X has no blocks, it transmits nothing.

The first rule states that each node uses its l links in a round-robin order. At every tick, all data transfers occur across one dimension of the hypercube. The latter two rules dictate what data is transferred, and compactly characterizes the actions of the Binomial Pipeline. We refer to the algorithm defined by these rules, as well as the generalization in Section 3.1.3.5, as the "*Hypercube Algorithm*". Since the Hypercube algorithm is a version of the Binomial pipeline, **this algorithm is also optimal**.

We illustrate how the Hypercube algorithm is a version of the Binomial Pipeline using an example, depicted in Figures 3.5 and 3.6. It uses the same nodes and indices as the earlier Binomial Pipeline example from Section 3.1.3.3. In this case though, we depict the binary representation of the client nodes' indices. Since there are eight nodes in total, they are organized into a hypercube of dimension three: a cube.

The first tick in Figure 3.5 shows the grouping of nodes by encircling each group by an oval. One plane of the cube forms G_1 with four nodes, while the rest can be divided into $G2$ with two nodes and G_3 with one node, plus the server. Therefore, the first three ticks, each shown in Figure 3.5, are exactly the same as the Opening Game of the Binomial Pipeline of Figure 3.4.

Now, Figure 3.6 shows Ticks 4 and 5, which corresponds exactly to the middlegame and the endgame phases of the Binomial Pipeline. Observe the invariants at the beginning of Tick 4: exactly four client nodes, corresponding to an entire plane, possess B_1; two clients possess B_2, and one client possesses B_3. Therefore, when we match nodes in G_1 with all other nodes, it corresponds to a transfer of blocks across one of the dimensions of the cube. This is seen in Tick 4 of Figure 3.6. Now a new set of groups is formed, as in the Binomial Pipeline. This re-grouping and subsequent

Endgame transfers are shown in Tick 5 of Figure 3.6. Again, ovals are used to encircle nodes in a group. This hypercube algorithm ends up sending all blocks to all nodes at the end of Tick 5. This is exactly the same completion time as the Binomial Pipeline of Figure 3.4.

3.1.3.5 Generalizing to Arbitrary Numbers of Nodes.

What happens when the number of nodes n is not an exact power of two? We can carry over the intuition of the Binomial Pipeline to this general case too. Let $l = \lfloor \log n \rfloor$ and assign each client a non-zero l-bit ID. The server receives the ID with all zeroes. We ensure IDs are assigned such that (a) each ID is assigned to at least one client, and (b) no ID is assigned to more than two clients. Observe that this is always feasible since the number of clients is at most twice the number of available IDs.

Consider the hypercube resulting from the ID assignment, in which each vertex corresponds to either a single node or to a pair of nodes. We may now run the same Hypercube algorithm as in Section 3.1.3.4, treating each vertex as a *single logical node*. All that remains is to describe how this "logical node" actually transmits and receives data.

If the logical node is a single physical node, it simply acts as in the Hypercube algorithm. Consider a logical node (X, Y) consisting of two real clients X and Y. During every tick, the Hypercube algorithm requires (X, Y) to transmit some block, say B_i, and receive some block, say B_j. We use the following rules to determine the actions of X and Y during this tick:

- If X has B_i, it transmits B_i. Otherwise, Y is guaranteed to have B_i and Y transmits it.
- Whichever node is not transmitting B_i will receive B_j.

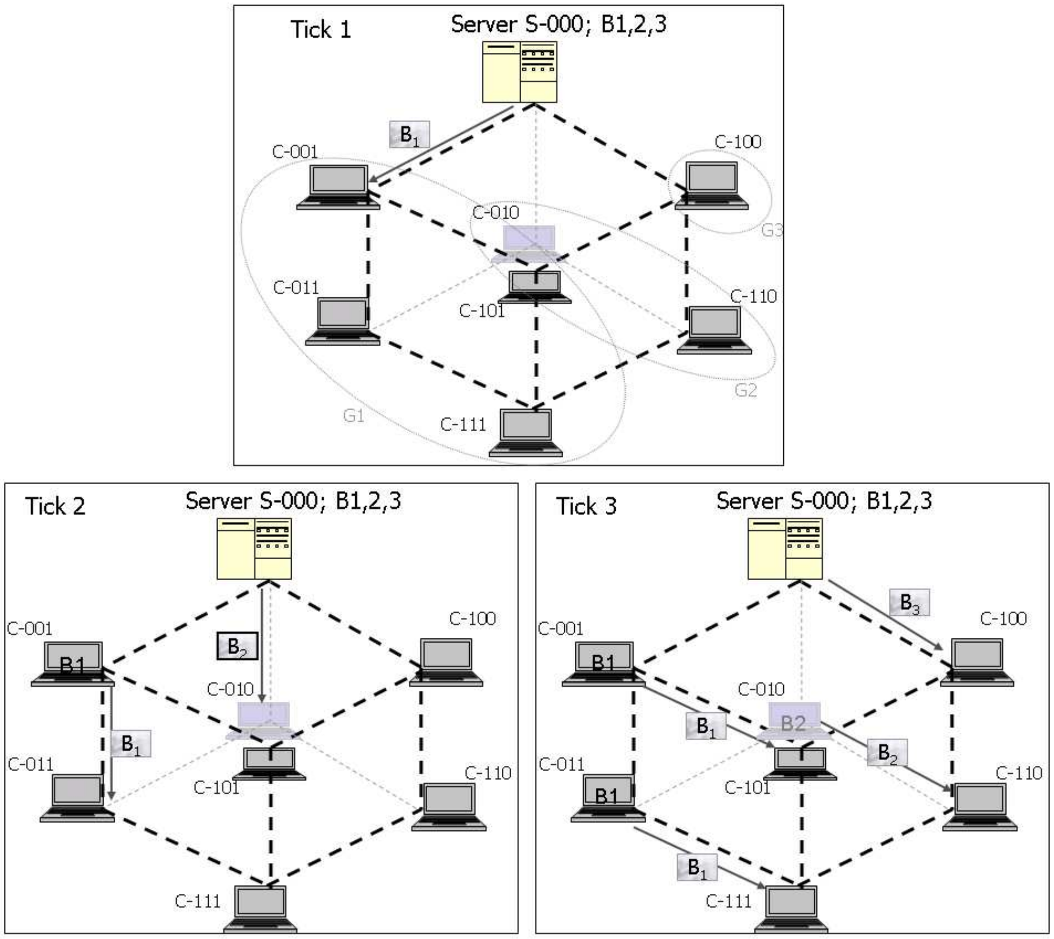


Figure 3.5. Ticks 1-3 of the hypercube algorithm with 8 nodes and 3 blocks

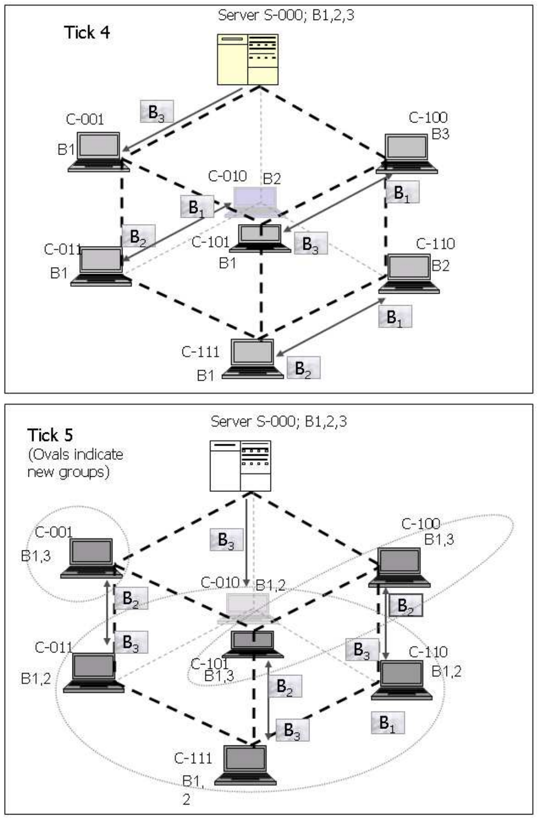


Figure 3.6. Ticks 4-5 of the hypercube algorithm with 8 nodes and 3 blocks

- Say X transmits B_i and Y receives B_j. If Y has a block that X does not, Y transmits this block to X. A similar exchange occurs if the roles of X and Y are reversed.

At all times, X can have at most one block that Y does not, and vice versa. Therefore, once the Hypercube algorithm terminates after $k+l-1$ ticks, both X and Y might be missing at most one block. They may use an extra tick to exchange these missing blocks, ensuring that the overall algorithm terminates in $k+l = k+\lceil \log n \rceil - 1$ steps. Thus, this generalization of the Hypercube algorithm *is optimal for all n.*

This algorithm uses an overlay network with the out-degree of each node being $\lceil \log n \rceil$, although the in-degree of some nodes may be as high as $2\lceil \log n \rceil$.

3.1.3.6 Other Observations

The hypercube algorithm described above is optimal in terms of completion time $T(k, n)$, for any value of k and n. This holds true since we showed that its completion time is the same as the lower bound derived in Section 3.1.3.2. This algorithm possesses many additional interesting properties, generalizations, and requirements. We discuss these properties below.

Individual Completion Times: While we have shown that the overall completion time of the algorithm is optimal, when does each node finish receiving the file? It can be trivially seen that all nodes finish receiving the file at exactly the same tick, so long as $k > 1$.

Higher Server Bandwidths: So far, we have assumed the server bandwidth equals client bandwidths. What happens if the server S has a higher upload bandwidth? Let the server have an upload bandwidth of λU, where U is the upload bandwidth of the clients. Then the natural strategy of breaking up the clients into λ equal groups, and breaking up the server into λ virtual servers, with one for each client group, is optimal.

Optimizing for Physical Network: Consider a situation where the available bandwidth between alternative pairs of nodes may be different, depending on their location in the physical network. We could perhaps "optimize" the hypercube structure using embedding techniques, such as those discussed in [26]. Such techniques help find the "best" hypercube that may be constructed with the given set of nodes, thus optimizing for the location of nodes in the physical network. We do not discuss this any further, as such techniques as part of future research.

Rendezvous and Coordination: A distributed implementation of the hypercube algorithm requires a method for all nodes joining the system to find each other and organize themselves. For this purpose, we suggest the use of a centralized well-known node, analogous to the Tracker used in BitTorrent. Our contribution is in the data-plane of the client network, and not in the control plane. The tracker is a control plane mechanism. So we will assume the use of a tracker henceforth, without further discussion of its operation.

Dealing with asynchrony: So far, we have assumed that the entire system operates in lock-step . In reality, different nodes may have slightly differing bandwidths. We may consider operating the hypercube algorithm even in these settings, with each node simply using its links in round-robin order at its own pace. This approach is closely related to the randomized algorithms that we discuss in the following sections.

3.2 The Randomized Approach - a Bridge to Heterogeneous Scenarios

In the preceding section, we described our optimal algorithm for the simple homogeneous model. Our next challenge is the scenario where the source and clients might have differing bandwidths. The transmission bandwidth from a client node could further depend on which client is the destination. The optimal algorithm described in Section 3.1.3.4 uses a deterministic matching algorithm, which needs to be executed in a synchronized, or lock-step manner. If the block transmission rates are significantly different across clients, it is not practical to sustain lock-step execution. Each "tick" or round of execution would have to last as long as the slowest block transmission during that round. Therefore, we consider an alternative. This requires a non-deterministic method of matching nodes to transfer blocks among each other. In this section, we develop a method for this purpose, which performs node matching in a random manner. We refer to this algorithm and its variants as the "randomized approach".

In this section, we describe and investigate a simple randomized algorithm for this purpose. Before actually considering specific scenarios involving heterogeneous clients, we consider whether our simple randomized algorithm can replace the optimal hypercube algorithm without a significant performance penalty. Therefore, we compare the results of this randomized algorithm with the optimal hypercube algorithm, in the simple homogeneous scenarios. We perform detailed simulations and report our results in Section 3.2.2.

We find that the randomized algorithm performs close to optimal in under these simple assumptions. Therefore it can serve as the basic template for algorithms tailored to more complex environments. We consider these complex environments and the associated algorithm design problems in the a later section.

3.2.1 The Basic Randomized Approach

The optimal solution of Section 3.1.3.4 required nodes to be interconnected in a rigid hypercube-like structure, and tightly controlled the inter-node communication pattern. In heterogeneous client scenarios, such lock-step execution may not be particularly robust, leading us to investigate the performance of simpler, randomized algorithms for content distribution.

Recall that, in every tick of the Hypercube algorithm, the optimal algorithm carefully finds a "maximal mapping" of uploading nodes to downloading clients to ensure that nearly all nodes upload data[4].

The deterministic mapping used in the Hypercube will not be suitable for situations where the block transfer for each pair of clients lasts a different length of time. Therefore, an obvious alternative is to attempt a distributed randomized mapping, rather than a deterministic maximal one. The advantage of this non-deterministic method is that synchronization is no longer needed. We now describe our basic randomized approach in detail.

3.2.1.1 Overlay structure

Nodes are interconnected in some overlay network G, and each node is aware of the exact set of blocks present at all its neighbors. This is achieved by every node informing all its neighbors every time it finishes receiving a block. This is similar

[4]Note that such a maximal mapping is necessary but not sufficient for optimality. There are many maximal mappings for a particular tick which, if used, would rule out maximal mappings in subsequent ticks. In the special case of $n = 2^l$ however, all maximal mappings lead to optimality.

to how BitTorrent operates. No assumptions about the structure of G are necessary for the correct operation of our distribution algorithm. However, for a practical implementation, a random graph is the simplest structure. Therefore, we henceforth assume that G is a random graph, unless otherwise specified. We also assume the use of a tracker, as in BitTorrent, to enable nodes to find each other, to form the overlay network.

3.2.1.2 Distribution Algorithm.

Whenever a node X is not uploading data to any other node, X uses the following two-step process to find a client to which to upload (we explain italicized segments subsequently):

1. Let $\mathcal{N}$ be the set of neighbors that require a block held by X. Select a node Y from $\mathcal{N}$ *with sufficient download capacity*, using the *neighbor-selection* policy.

2. Let R be the set of blocks available with X and desired by Y. Select one of the blocks in R, selecting the block according to the *block-selection policy.*

Having chosen a destination node and a block, X then transmits that block to that destination node. When the transmission is completed, this process is repeated.

Neighbor Selection Policies Step 1 requires X to select a neighbor Y from among the set of "interested" neighbors, i.e., the set of neighbors that require blocks that X already possesses. Such a neighbor can be selected via simple random selection. Alternatively, neighbor selection can depend on other factors such as bandwidths of these neighbors or how much data they have transmitted to X. In this section, we only consider random neighbor selection. It is the simplest possible policy, requiring no information at the implementing node other than its list of neighbors . However, in Section 3.3, we address the need for more sophisticated neighbor selection policies in the presence of heterogeneous clients, and design one such policy.

If many nodes simultaneously pick the same Y to which to upload, there may be a downloading bottleneck at Y. To address this problem, we exploit the fact that a real system is asynchronous. Therefore, a handshake protocol between X and Y is used to verify that Y has sufficient download capacity - otherwise X will not select Y. If no node Y matches the requirements of Step 1, X does not transmit any data during that tick.

Block Selection Policies There are many possible block-selection policies that can be used in Step 2. These policies could utilize information such as the frequency of blocks in the system or the order of blocks needed by the application, or simply use no such information at all. The simplest policy is *Random*, in which a random block in R is uploaded to Y. A handshake protocol may be used to prevent Y from getting the same block from more than one sender.

We can also use *Rarest-First* block selection, in which the least frequent block is uploaded. There are different ways to estimate block frequency. We briefly mention one such way here. Rarity of a block may be estimated in different ways. One way to do so is for each node to maintain a "hop counter" associated with each block. Every time the node transmits a block, it increments the hop counter. A block with a low hop counter is likely to be rarer than a block with a high hop counter.

In this chapter, we only consider Random and Rarest-First block selection. Alternatives include policies based on block ordering or priorities. These are not relevant here because the order of data delivery is immaterial. When we remove this assumption, in Chapter 5, such block selection policies become important, and will be further explored.

A Note on Costs Observe that an implementation of this algorithm requires (a) a protocol to inform nodes of their neighbors' content and (b) a handshake protocol to decide on data transmission. The cost of implementing these grows with the degree of the overlay network G. The cost of (a) grows linearly with degree. It is therefore important to keep the degree of G small. The degree of the random graphs required to obtain low completion times is an important parameter in our simulation results.

3.2.1.3 Analysis and (Counter-)Intuition.

A theoretical analysis of the general algorithm presented above remains an open problem. Instead, we try to gain some intuition about its performance. For simplicity, assume there are $n = 2^l$ nodes with infinite download capacity, interconnected in a complete graph. Observe that the algorithm's completion time is inversely proportional to the average fraction of nodes that upload data in each step.

Assume, optimistically, that the algorithm proceeds just like the optimal algorithm in the opening phase. This is reasonable to assume, since the opening phase involves all nodes finding neighbors that are not already receiving data from other nodes, which will take only slightly longer than the optimal opening phase. Nodes are then in l groups, $G_1, G_2, \dots G_l$ with group G_i owning block B_i. Consider what happens at the $(l+1)^{st}$ tick. Our algorithm maps each node X in group G_i to a random node Y that does not have B_i. That is, X is mapped to a random $Y \notin G_i$, while ensuring that multiple nodes in G_i do not map to the same Y.

Consider the nodes in G_1. Since $|G_1|$ is $n/2$, each of the G_1 nodes will map to a distinct node outside G_1 and all nodes end up owning B_1 at the end of this tick. For any node in group G_i, $i \neq 1$, all nodes outside G_i are interested in its content. Since $|G_i| \leq n/4$, at least $3n/4$ nodes are interested, out of which $n/2$, or two-thirds, belong to G_1. Therefore, at least one-third of the G_1-nodes, i.e., $n/6$ nodes, are not expected to receive any block in this tick. If a transfer occurs between a pair of nodes, neither of which is in G_1, then some node in G_1 will not be able to receive any block in that tick.

Observe that these $n/6$ nodes will not upload any data at all in the next tick $(l + 2)$, since the only block they will have is B_1, which is already owned by all the nodes. Therefore, at most five-sixth of the nodes will transmit data at tick $(l + 2)$. Given the simple and weak inequalities used in our argument, it seems reasonable to conjecture that at most five-sixth of the nodes will transmit data at *every* tick. From this conjecture, we may guess that the randomized algorithm is at least 20% worse than the optimal algorithm. The experimental results, however, prove otherwise, as we shall see next.

3.2.2 Results

In the case of the hypercube algorithm, we were able to prove analytically that it was optimal in terms of completion time. In the case of the randomized algorithm, such analysis remains unsolved. Our approach to estimating the performance of this algorithm is via simulations. We describe the simulator and the results below. This simulator was also extended in several ways to handle more general models of client and network behavior. These extensions are detailed in later sections.

3.2.2.1 Simulator

We built a simulator to investigate the performance of the randomized approach described in Section 3.2.1. This simulator assumes a homogeneous synchronous model in which the simulation progresses as a series of ticks. Each tick begins and ends at the same time for all nodes. A block transmission between two nodes always takes exactly one tick. The time required for any other messages between nodes is assumed to be insignificant compared to the block transmission time. In this section we only discuss scenarios where a node can upload data to at most one other node at a time, and the download bandwidth capacity of a node is never a bottleneck. Later, in Chapter 4, we will relax this assumption, and consider its effects. Without loss of generality, we can then assume the tick to be the unit of time. We now discuss our simulation results for the cooperative randomized matching approach of Section 3.2.1.

We simulated the randomized algorithm, identifying the completion time T for different values of k and n, using a variety of different overlay networks G, and with different download bandwidths D and block-selection policies. Our key results are below. Block size B is assumed to be a constant across all simulations and completion time T is expressed in ticks. Since a tick is defined as B/U, T can be meaningfully compared across simulations.

3.2.2.2 Completion Time T vs. k and n

First, we estimate T as a function of k and n to understand the quality of the algorithm. We set G to be the complete graph, $D = U$ and use *Random* block selection, although the results are almost identical with other settings, as we discuss later.

Figure 3.7 plots the mean value of T as a function of n with the x-axis on a log scale. The error bars on each point represent the 95% confidence intervals on the mean, obtained through multiple algorithm runs. We observe that T increases roughly linearly with $\log n$. Replicating the experiment with different values of k produces the same near-linear behavior. Figure 3.8 plots T for different values of k on a log-log scale, keeping n fixed at 1024. We see that T increases linearly with k. The same behavior was observed with other values of n too.

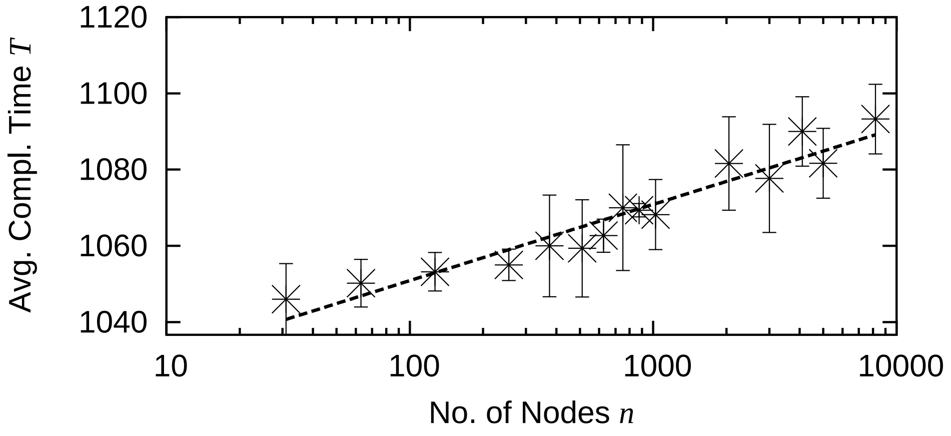


Figure 3.7. Completion Time T vs. n

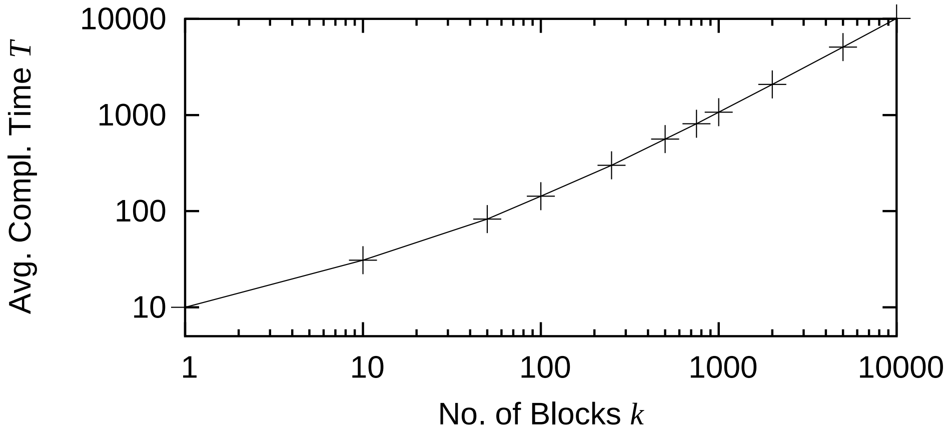


Figure 3.8. Completion Time T vs. k

Given the above evidence, we hypothesize that, to a first order of approximation, T is a linear function of k and $\log n$. [5]

Using least-square estimates over a matrix of 106 data points, we estimate that the expected completion time $T = 1.01k + 4.4 \log n + 3.2$, suggesting that the algorithm is less than 2% worse than the optimal for large values of k. This is a surprising result unexplained by our earlier intuition. A closer analysis of the algorithm's runs suggests that there is some "amortization" going on, by which a "bad" tick where few data transfers take place is compensated for by many succeeding "good" ticks where the transfer efficiency is 100%. This belies the intuition developed earlier.

3.2.2.3 Effects of the Overlay Network

The experiments described above use a complete graph as the overlay network. We now consider random regular graphs, in which each edge is equally likely to be chosen. We investigate the effect of graph degree on completion times. Figure 3.9 shows this effect with $n = 1000$, and two different settings of k: 1000 and 2000. We observe that the completion time drops steeply as the degree increases, and converges quickly to the final value when the degree is around 25, irrespective of the value of k. This suggests that the phenomenon may be related to the connectivity properties of G, with near-optimal performance kicking in when the graph degree is $O(\log n)$.

We executed the same randomized algorithm using a hypercube-like overlay network with average degree 10, for $n = 1000$. We found that the performance matches that of the randomized algorithm on the complete graph. Thus, it is useful to replace a random graph by a hypercube-like structure if the objective is to reduce the graph degree to further extent, without increasing the completion time.

[5] We conjecture that the true equation is of the form $T = k + O(\textit{polylog } n) + o(k)$.

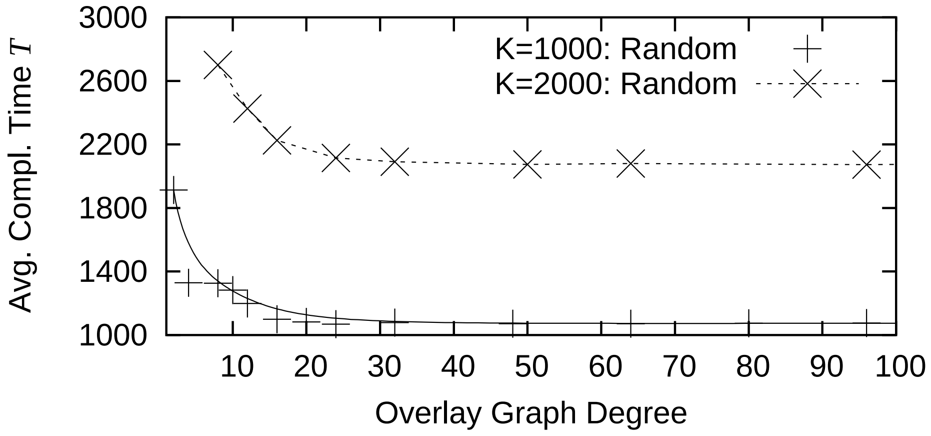


Figure 3.9. T vs. $Degree$ with $n = 1000$

3.2.2.4 Block-Selection Policy and Incoming Bandwidth Constraints

Finally, we also attempt using the Rarest-First block-selection policy instead of Random, and also vary the incoming bandwidth of nodes from U to infinity. We find that there are no significant differences in the overall results or trends as a consequence of these variations. However, as we shall see, when we consider more challenging environments in Chapter 4, there are other scenarios in which the Rarest-First policy performs better than Random block selection.

3.3 The Cooperative Heterogeneous Bulk Scenario - Optimizing the Randomized Algorithm

Section 3.1 began with the assumption of homogeneous clients. Section 3.2 introduced a randomized algorithm with the intent of facilitating algorithm development for a more complex heterogeneous environment. However, our discussion and investigation of this randomized algorithm has so far avoided heterogeneous environments. The results presented in Section 3.2 were for a homogeneous environment.

In this section, we aim to develop distribution algorithms that perform well in heterogeneous environments. In particular, we tailor the basic randomized algorithm proposed in Section 3.2 with policies that enable low completion times in heterogeneous client-bandwidth environments. We then evaluate these algorithms in several specific heterogeneous client conditions.

In our description of the Randomized algorithm in Section 3.2, we mentioned the existence of an important customizable component: the Neighbor Selection policy. This is how a node decides which of its neighbors to choose as a target for block transmission. Since Section 3.2 introduced the most basic version of the randomized algorithm, we have discussed only one candidate policy for neighbor selection: Random Neighbor Selection. Because we have only reported the performance evaluation of our algorithms in homogeneous environments in Section 3.2.2, we have found this Random Neighbor Selection to be close to optimal. However, when we consider a heterogeneous network, the performance of this simple algorithm, as well as other algorithms like BitTorrent, might suffer. Given a choice between two neighbors, one with a low transmission capacity and the other with a high transmission capacity, choosing randomly between them might not be the best course. In this section we seek the best way to make such decisions. We aim to design the neighbor selection policy that is most likely to minimize the completion time for several heterogeneous client-bandwidth environments.

To better understand these scenarios, we need some background on the nature of clients bandwidths and how they differ from client to client. In addition, given our focus on neighbor selection, the reader also needs some intuition regarding the pitfalls of the currently used neighbor selection methods. We first present this background and intuition in Section 3.3.1, motivating the need for and design of a new neighbor selection method. We then proceed to the proposal of our Neighbor Selection policy in Section 3.3.2. This proposal is named "$HRAND$". It is based on a heuristic aimed at maximizing the utilization of the clients' upload capacity, assuming heterogeneous client bandwidths. Finally, in Section 3.3.3, we evaluate $HRAND$ via simulation. We compare this to other candidate Neighbor Selection policies, including BitTorrent's. The results indicate that $HRAND$ performs better, in general, than the other known or obvious choices for neighbor selection. $HRAND$ reduces completion time by a factor between 1.5 and 2 in several cases.

3.3.1 Background on Heterogeneous Client Conditions

In prior sections, we assumed that the rate at which a client or the server can transmit a block to another client is the same. This was determined solely by the tail link or "last-mile" capacity at each client, irrespective of which client we considered. This is indeed possible in certain restricted scenarios. One example is a number of clients from a single controlled enterprise network. We name this "Client Example A", for future reference. Another example is a number of clients behind a low-bandwidth DSL or dial-up network. Here the first outgoing link from each client has the same capacity, and is always the bottleneck. We name this "Client Example B".

However, in practice, we are likely to encounter scenarios where the client set consists of machines with a wide variation in their transmission bandwidths. For example, a client set could be a mixture of two types of machines: (a) at-home PCs using cable modems with a maximum transmission rate of 128 kbps, and (b) PCs in a university LAN connected to a Tier-1 ISP via a high-bandwidth path, resulting in transmission bandwidths of more than 1 Mbps to other clients. We name this "Client Example C".

We had also assumed, in the simple homogeneous model, that the transmission rates were bottlenecked by the capacity of the "last-mile" link. This is true in Client Example B. However, there are several other scenarios where this assumption is not valid. We now provide some insight into the possible nature of client bandwidth distribution in such scenarios.

If the bottleneck on different network paths between nodes in our client set is actually in the network rather than the "last-mile", then we could observe the outgoing transmission bandwidth from a node vary depending on the destination node. This is because the network path and associated bottlenecks would depend on the choice of destination node. For our purposes, and from the existing literature, the best model for the distribution of client bandwidths is unclear. Many distributions are feasible depending on the application scenario and choice of client set. Therefore, in this dissertation, we consider many possible distributions. One natural choice is a uniform random distribution of bandwidths for each source-destination client pair.

Another distribution we consider is based on the idea that we might encounter "clusters" of clients. The clustering could be either on a topological or geographical basis. Prior research has suggested that, in the near future, one likely spot for the bandwidth bottlenecks in end-to-end Internet paths will be the border between large networks or ISPs [51]. Furthermore, it is likely that clients communicating across geographically distant continents will experience lower data rates than clients in the same continent and geographic vicinity. It is reasonable to consider a scenario where all clients within an ISP (or continent) exhibit high transmission rates to each other. But these same clients all exhibit low transmission rates to clients within other ISPs (or continents). Consider another example, named "Client Example D", which illustrated this "clustered" distribution. All clients belong to a single company's global network, which is composed of several geographically distributed sub-networks. The connectivity between these sub-networks is likely to be more expensive to provision than within a single sub-network's local area. Therefore the transmission rates across the sub-networks would be much lower than those within each sub-network.

The environments described above determine the efficiency of our distribution algorithms. In particular, the neighbor selection algorithms must account for the nature and distribution of inter-node bandwidth. We now consider the well-known or obvious candidates for neighbor selection policy, and illustrate their possible pitfalls. We mentioned earlier that a random neighbor selection scheme, agnostic to client bandwidths, might fail to provide optimal performance. We illustrate this with an example.

Consider Figure 3.10, in which node S has a block B_1 that the remaining nodes do not possess. The goal is to distribute this block to the remaining nodes X, Y, Z, A and B. All these nodes have established peering relationships among each other that are represented by the non-directional solid lines in the figure. At this point we do not focus on how we obtained this peering graph. Given this situation we aim to distribute the block B_1 in the shortest possible time. Nodes only learn of blocks possessed by their peers. Thus, blocks can only be transmitted along the edges of this peering graph. In other words, the edges in this graph represent virtual links (also known as overlay links) between the peering nodes. The underlying physical network topology limits the rate at which data can be transmitted between S and X to a quarter of the rate between other neighbors in this peering graph. This graph is a miniature-scale representation of 'Client Example D". We can consider the six nodes as belonging to two separate clusters with high bandwidth within each cluster, and much lower bandwidth between clusters.

Now, S has a block B_1 which is desired by both its neighbors in the peering graph, and has to decide which of these neighbors should receive the block. If the neighbor selection policy used in this system were Random selection, then in many cases, the distribution would progress as shown in Figure 3.10. Here S chooses to transmit blocks to neighbors in its clusters during Tick 1, before transmitting it to its neighbor X in the other cluster during Ticks 2-5. The block then needs to be distributed by X

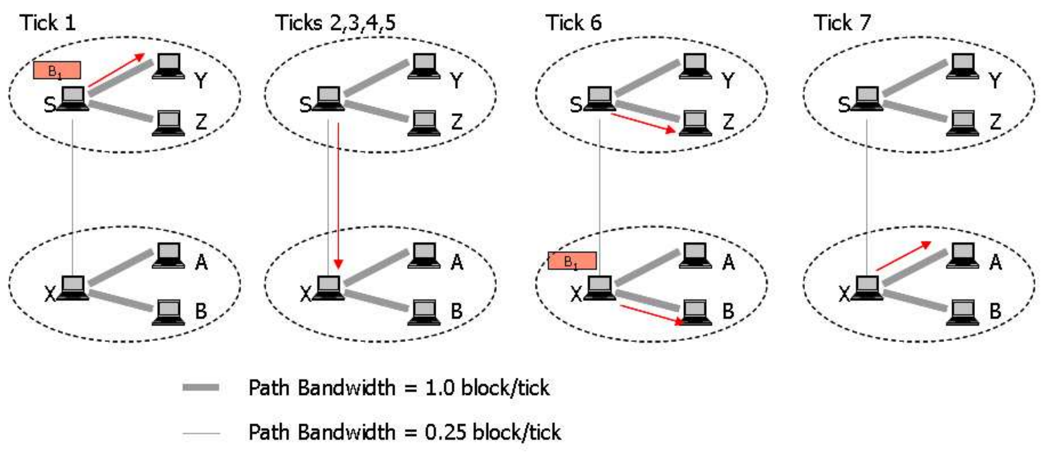

Tick 1
Ticks 2,3,4,5
Tick 6
Tick 7
B1
S
Y
Z
X
A
B
Path Bandwidth = 1.0 block/tick
Path Bandwidth = 0.25 block/tick

Figure 3.10. Example of Random Neighbor Selection Policy in operation

to other nodes A and B in its cluster, during Ticks 6 and 7. The total completion time in this particular case is seven ticks. Since the Neighbor Selection policy is Random, other progressions could occur, with different completion times. But the point is that this particular completion time can occur with non-trivial frequency.

Now, consider an alternative Neighbor Selection Policy, which, through some (as yet unknown) logic, decides that S should serve X first during Ticks 1-4, before its intra-cluster neighbors Y and Z. This leads to the progression depicted in Figure 3.11. During Ticks 5 and 6, the block can be distributed by both S and X to each of their neighbors, leading to a total completion time of six ticks. This is faster than the earlier observed finish of Random Selection, which was seven ticks. While the difference in this example might seem small, the point is that it is desirable to introduce blocks to all clusters earlier, so that subsequent high-bandwidth intra-cluster distribution can proceed in parallel, leading to lower completion times.

This raises the question: is there a Neighbor Selection policy that can perform better than Random Neighbor Selection? It is natural to consider policies based on selecting neighbors with which there is higher-bandwidth connectivity than to other neighbors. We will call this family of policies "Greedy" bandwidth-based neighbor

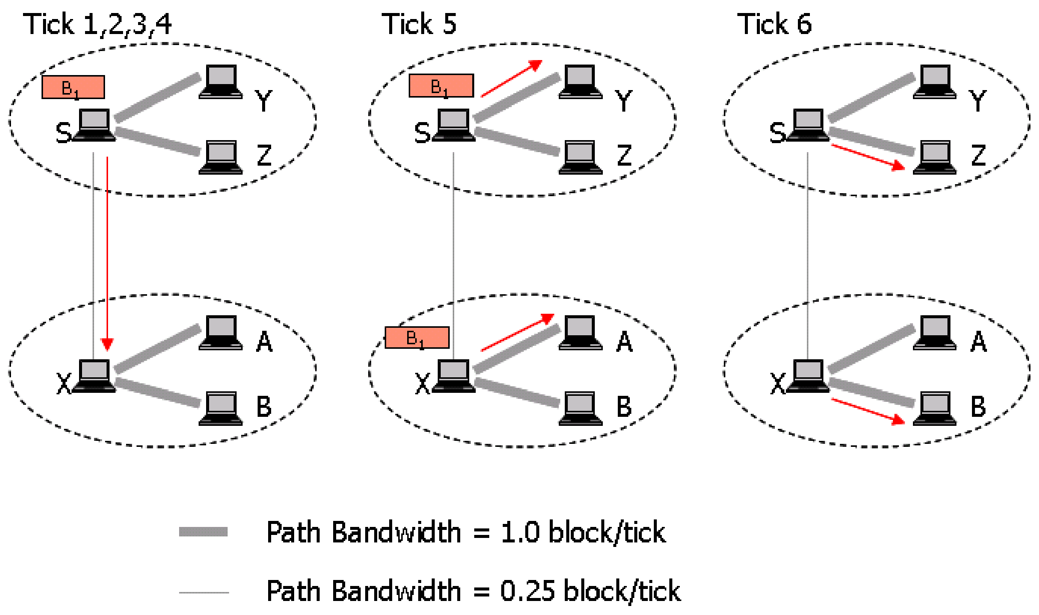
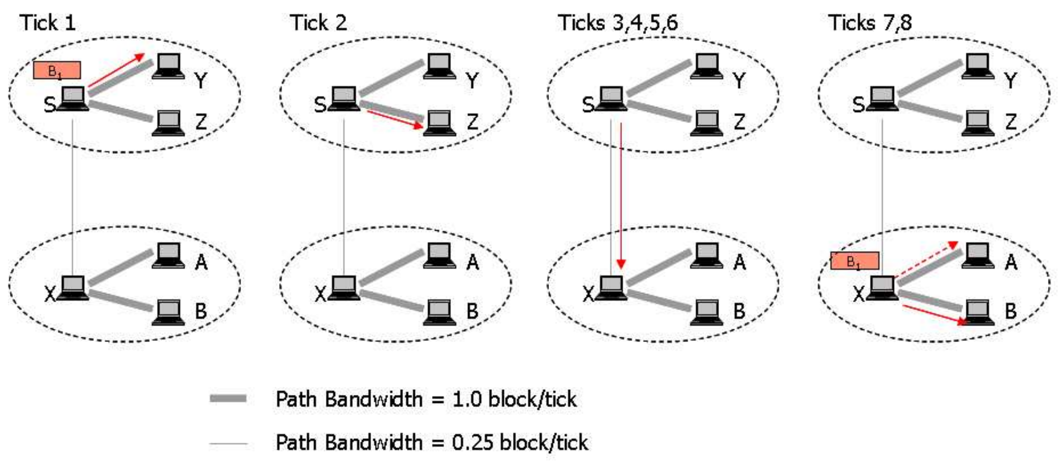

Tick 1,2,3,4
Tick 5
Tick 6
B1
S
Y
Z
X
A
B
Path Bandwidth = 1.0 block/tick
Path Bandwidth = 0.25 block/tick

Figure 3.11. Example of a fast Neighbor Selection Policy in operation

selection. BitTorrent's neighbor selection policy, described in Chapter 2, is a version of Greedy bandwidth-based selection. In particular, neighbors are selected on the basis of received data transmission rate, i.e., in a tit-for-tat manner. This method is effective in an environment where clients are selfish and need incentives to cooperate and upload data to each other. However, if we assume that the clients will upload data cooperatively, without any such incentive mechanism, would BitTorrent's neighbor selection be the best-performing method? We illustrate the possible shortcoming of this method using the following example, depicted in Figure 3.12.

We use the same topology and scenario from our prior discussion of Figures 3.10 and 3.11. We now assume that the neighbor selection policy in the system is Greedy bandwidth-based selection. Since bandwidths between nodes are symmetric, this policy behaves exactly as would BitTorrent's policy. As before, S has a block B_1 desired by other nodes, and has to choose between Y, Z and X to first send the block to. Since the bandwidths from S to/from Y and Z are higher than the bandwidth from S to X, the Greedy bandwidth-based policy dictates that Y and Z should be



Tick 1
Tick 2
Ticks 3,4,5,6
Ticks 7,8
B1
S
Y
Z
X
A
B
Path Bandwidth = 1.0 block/tick
Path Bandwidth = 0.25 block/tick

Figure 3.12. Example of a "Greedy" bandwidth-based Neighbor Selection Policy in operation

served first, during Ticks 1 and 2. Only then is the block transmitted to X, during ticks 3-6. Later, during Ticks 7 and 8, A and B finally receive the block, from X. This progression, depicted in Figure 3.12, leads to an overall completion time of eight ticks. We already know that this can be improved if we design a policy that leads to the progression of Figure 3.11. In fact, Greedy selection appears to be potentially counter-productive in this example, performing worse than the simple Random selection of Figure 3.10.

We have now provided some background and context on the nature of client bandwidth environments and their relationship with neighbor selection policy. The shortcomings in existing/known neighbor selection methods were only illustrated for certain specific examples. However, this does motivate the need to carefully examine whether these known methods are the best possible and to consider the possibility of designing better methods. We now present our design and proposal of a new Neighbor Selection policy, in Section 3.3.2. Then, in Section 3.3.3, we will evaluate this and compare with the current known methods, including Random Neighbor Selection and BitTorrent's neighbor selection policy.

3.3.2 Our Proposal - the *HRAND* algorithm for heterogeneous environments

Thus far, we have discussed the potential impact of the neighbor selection policy on the completion time under heterogeneous conditions. We motivated the design of a new policy for these conditions, to achieve the lowest possible completion time. In this section, we propose a variant of our earlier randomized algorithm of Section 3.2.1 that is tailored to client heterogeneity. In particular we propose a Neighbor Selection policy that is based on a heuristic that aims to maximize the utilization of service capacity available from the total set of clients, under various models of client-to-client bandwidth distributions.

From our experiences in Chapter 3, we found that the key to reducing completion time is to utilize the service capacity, or transmission capacity, of the clients to the maximum possible extent. The examples from Figures 3.10,3.11, and 3.12, discussed in Section 3.3.1, illustrate this principle. In other words, our primary goal is to keep nodes "busy" serving useful blocks to other nodes, and to ensure that this service is at a high data rate. Imagine a queue of blocks held by each node that can be supplied to other nodes. We aim to ensure that this supply queue never runs too low at any node, as far as possible.

Therefore, we consider a neighbor selection policy that gives preference to nodes that can serve their neighbors at high data rates, but are running out of useful blocks to serve. To account for nodes whose neighbors already possess most blocks, we use an idea complementary to the aforementioned "supply" concept. We identify preferred nodes by adding up the number of blocks demanded from these nodes from their neighbors, weighted by the transmission rates to them.

Here is the actual heuristic calculation we propose, termed "*HRAND*". It is performed by each node X that has to choose which of its neighbors $Y_1 \ldots Y_d$ to send some block. Consider each neighbor Y_i, one by one. Now, Y_i has its own neighbors $Z_1 \ldots Z_e$.

- $ndemand(Y_i, Z_j)$ is the number of blocks that Z_j does not have that Y_i also does not have.
- $wdemand(Y_i, Z_j) = weight(Y_i, Z_j) \cdot ndemand(Y_i, Z_j)$.
- $demand(Y_i) = wdemand(Y_i, Z_1) + \ldots + wdemand(Y_i, Z_e)$.
- Output is the node Y_i with the highest value of $demand(Y_i)$, for $i = 1 \ldots d$.

$weight(Y_i, Z_j)$ is a measure of how fast Y_i can serve Z_j. We are trying to assess how useful it is to choose the node Y_i to send a block. So the weight function should be higher if the transmission capacity from Y_i to its neighbors $Z_1 \ldots Z_d$ is high. We can use one of two simple methods for this purpose. In one, $weight(Y_i, Z_j)$ is proportional to the bandwidth from Y_i to Z_j. In the other, $weight(Y_i, Z_j)$ is 1 if the bandwidth from Y_i to Z_j is above some pre-set threshold $BWTHRESH$, and 0 otherwise. We will assume the latter threshold-based method for all further discussion.

Finally, $demand(Y_i)$ is the final measure of high-bandwidth service capacity associated with each neighbor. Neighbors with higher values of this measure will get selected in preference to those with lower values. We abbreviate $demand(Y_i)$ to $demand()$ when the operand is obvious from the context, for the sake of convenience.

We now reprise the scenario from Section 3.3.1, and depicted in Figure 3.11. Consider what happens when nodes utilize the $HRAND$ policy. The bandwidths achievable along the peering edges are all above $BWTHRESH$, except for the edge between S and X. S has to choose between X,Y and Z to transmit its block. $HRAND$ dictates than S should consider the $demand()$ values of X,Y and Z. X has the highest $demand()$ value, since $demand(X) = 2$, whereas $demand(Y) = demand(Z) = 0$. Therefore X is chosen first by S to send the block. This leads to the same progression that we mentioned earlier in Figure 3.11, leading to a completion time of six. This is better than both Random and Greedy Bandwidth-based neighbor selection.

3.3.2.1 Implementation Details

We have described the $HRAND$ algorithm in concept. We now address its implementation details, and point out specific issues that bear further discussion.

For a node X to perform the $HRAND$ calculation, it needs to know the $demand()$ values of all its neighbors. This can be achieved by a message from each neighbor Y containing $demand(Y)$. We will henceforth refer to such messages as DEMAND messages.

Each node Y can calculate $demand(Y)$ locally with no additional messaging, since the basic randomized framework already keeps Y informed about the blocks its neighbors possess. Y has to recalculate $demand(Y)$ potentially every time Y or one of its neighbors receives a block. This raises the issue of how often Y should send a DEMAND message to X and its other neighbors. One option is to send it every time $demand(Y)$ changes. While this keeps up-to-date information at all nodes, it increases the messaging traffic considerably, by a factor proportional to the degree of the peering graph. Therefore it would only be useful if the degree of the peering graph were considerably lower than the total number of nodes. We assume this through the remainder of this dissertation, unless stated otherwise.

In scenarios where the messaging overhead needs to be reduced, we can assume that DEMAND messages are propagated less frequently compared to changes in the $demand()$ values. For example, DEMAND messages could be sent by a node to all its neighbors periodically. Alternatively, they could be sent in response to a notification from a neighbor that it just received a new block. The frequency of DEMAND messages governs a trade-off of the performance of the neighbor selection heuristic against the messaging overhead. The further exploration of this tradeoff in several real-world scenarios remains an area of future work.

In implying that each node Y can locally calculate $demand(Y)$, we have assumed that Y knows the bandwidth at which it can transmit data blocks to each of its neighbors. In practice, this can be achieved by initially using pre-set values of these bandwidths, which can subsequently be modified with actual observed rates of block transmission whenever Y sends a block to one of these neighbors. With no initial information the pre-set values would all be equal and the neighbor selection would therefore resemble Random selection in the initial stage of the distribution. However, there are several ways in which we could obtain a useful pre-set bandwidth value. One is to utilize the tracker introduced in Chapter 2, to collect and disseminate bandwidths from nodes that are bottlenecked at the last-mile. These nodes would therefore exhibit the same transmission bandwidth to all destinations. Another way is simply to maintain a record of bandwidths observed in past distributions. This is feasible in a scenario where a static client set is organized into a distribution system that is repeatedly used to disseminate new data. An example of such a scenario is a set of clients subscribed to a video distribution service, that sends a different large video file every day to all the clients.

3.3.3 Evaluation

We have described our proposal for a randomized distribution algorithm that uses the $HRAND$ neighbor selection method. We now present its evaluation and comparison with other alternatives. As in Section 3.2.2, we performed simulations to evaluate these algorithms. We first describe our simulation methodology, and then the results.

3.3.3.1 Simulation Methodology

We extended the simulator described in Section 3.2.2 to handle heterogeneous client bandwidth models. As before the simulator captures the existence of the client nodes, and the peering graph formed by them. In addition, the simulator also allows for an arbitrary data transmission bandwidth between each pair of client nodes. This method has some drawbacks in comparison with a real-world implementation. First, we assume that the effect of two simultaneous block transmissions between two different pairs of nodes that share a common bottleneck link in the network is not significant. Second, we assume that messaging overheads are not significant in comparison with the actual block transmission times. However, it allows us to experiment with various models of heterogeneity, as well as variants of distribution algorithms and their parameters.

The data unit used in these simulations is the data block, and the time unit used is the tick. However, the definition of tick used in Section 3.1 requires clarification in the heterogeneous case. A tick is now any arbitrary unit of time, usually chosen to be the lowest block-transmission time in the system. This allows us to compare completion times for several different distribution algorithms, for a given client set BPD, including the parameters of the BPD. But we cannot, in general, compare completion times across client sets with different BPDs. The metric of interest is the completion time, in ticks, of each distribution algorithm.

As in Section 3.2.2, the nodes are assumed to form peering relationships at random, with a pre-configured low value of peering-graph degree. In fact, this particular assumption is held for the rest of the dissertation unless otherwise specified.

Bandwidth Distribution Models We assign the bandwidths between nodes, for each pair of nodes, using some model of how the client bandwidths of distributed.

In practice, we expect that several different distributions of client sets might be encountered. For example, if the client set consisted of a set of machines behind low-bandwidth dial-up nodes, they would probably all have the same bandwidth to each other, governed by the bandwidth of the dial-up link. On the other hand a client set consisting of a mix of nodes behind high-bandwidth LANs, DSL modems, and dial-up nodes, would necessitate a model of randomly distributed client-to-client bandwidths. In fact, each of the Example Scenarios A, B, C, and D, from Section 3.3.1 would be represented by a different bandwidth probability distribution. Our approach is to choose several different bandwidth distribution models representing different reasonable real-world scenarios, and evaluate our algorithms for each of them. Here are the bandwidth probability distributions we consider, which we refer to as "BPDs":

- **Homogeneous BPD:** This represents the simple homogeneous model presented in Section 3.1, and is only listed here for completeness.
- **Two-Level BPD:** Nodes in this BPD are assigned a transmission bandwidth chosen from one of two values: a high value and a much lower value. Furthermore, it is assumed that the transmission bandwidth remains at this chosen value irrespective of the destination node for the transmission. This BPD would represent client sets drawn from both dial-up and DSL nodes, for example, and is related to Client Example B from Section 3.3.1.
- **Clustered BPD:** Nodes in this BPD are assumed to be divided into logical or topological clusters. Nodes have a pre-configured high value of bandwidth to other nodes that are in the same cluster, and a pre-configured low value of bandwidth to nodes that are not in the same cluster. The number of clusters, nodes within each cluster, and the high and low values of bandwidths are all input parameters of this BPD. This is meant to represent scenarios like Client Example D from Section 3.3.1, where the clustering of nodes occurs due to topological

reasons like nodes lying in different sub-networks of an enterprise network, or different ISPs, or continents.

We have chosen several BPDs, and consider them to be a sufficient exploration of the possible range of scenarios for the scope of this dissertation. However, this list is not exhaustive, and more BPDs can be considered. Such considerations are in the realm of possible future work.

Distribution Algorithms Simulated We have described what the simulator captures and how we assign client bandwidths using the various listed BPDs. We now specify the distribution algorithms used in the simulations. In Section 3.2.2, when we simulated our basic randomized algorithm for a simple homogeneous client bandwidth model, we were able to recognize whether an observed completion time result was good or not, by comparison with the *optimal* completion time. This was because we derived the optimal completion time for that model analytically. When considering heterogeneous clients, the derivation of completion time by analytical means remains open. We therefore compare the simulation results of different possible distribution algorithms to each other, to determine the best algorithm. Essentially, our goal is to compare the $HRAND$-based randomized algorithm to BitTorrent and Random Neighbor Selection. BitTorrent was chosen because it is the closest related work, and Random selection was chosen because is the most intuitive or obvious alternative.

We specifically want to compare Neighbor selection policies. We seek to eliminate the effects of incentive mechanisms and other features of these algorithms unrelated to handling heterogeneous clients. The cost and effect of the incentive mechanism is described later, in Chapter 4. Here are the different neighbor selection policies, abbreviated to "NSPs", that we consider:

- $RAND$**:** This refers to Random Neighbor Selection, introduced in Section 3.2.1.

Whenever a node X has to choose among a list of neighbors that seek some block from X, a node is chosen uniformly at random from that list.

- *HRAND*: This refers to the *HRAND* policy of Section 3.3.2. The simulator does not explicitly capture messaging delay and bandwidth effects of control messages such as DEMAND messages. We assume that these effects are negligible in comparison to actual block-transmission effects, as long as peering-graph degrees are low.
- *GRAND*: This is a Greedy Bandwidth-based approach which captures BitTorrent's neighbor selection policy. When a node X has to choose among a list of neighbors that desire to receive some block from X, the neighbor with the highest bandwidth to X is chosen. In cases where client-pair bandwidths are symmetric, the neighbor's bandwidth to and from X is the same. While we seek to compare *HRAND* to BitTorrent, we want to eliminate the effects of the incentive and "choking" mechanisms of BitTorrent, detailed in Chapter 2. Therefore we isolate its neighbor selection policy by using this *GRAND* approximation.

3.3.3.2 Results for Heterogeneous BPDs

We have already explored the results of the Homogeneous BPD in detail in Section 3.2.2. We now consider our first truly heterogeneous BPD: the Two-Level BPD. The clients' bandwidths can be one of only two values BW_l and BW_h, one of which is ten times larger than the other. This ratio is chosen to represent, approximately, the ratios observed when comparing some kinds of cable modems and dial-up modems, or when comparing DSL/cable modems to high-connectivity enterprise LANS. Half the clients are chosen from the higher-bandwidth pool, and other half from the lower-bandwidth pool. We refer to this BPD with these parameters as BPD-A.

As always, n is the total number of nodes including the source, and k is the

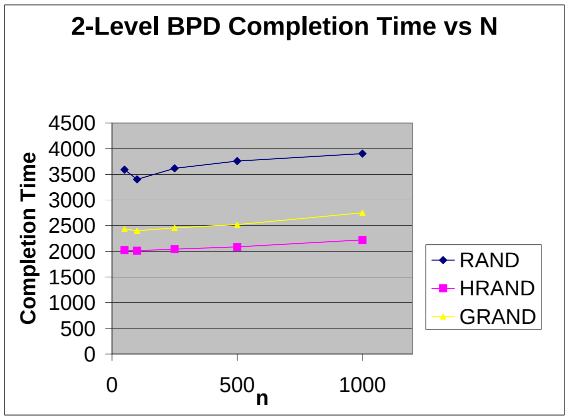


Figure 3.13. Completion Time T vs. n for BPD-A (Two-Level BPD)

total number of blocks in the file being distributed. The block-selection policy used by default is Rarest-First. When a node X has to choose among several blocks to transmit to a particular neighbor Y, X chooses the block that is possessed by the fewest neighbors of X. The value of $BWTHRESH$ for $HRAND$ is pre-configured to lie between BW_l and BW_h.

Figure 3.13 shows the completion times for BPD-A, for all three different NSPs we mentioned: $HRAND$, $GRAND$, and $RAND$. The X-axis varies n through different values, and k is fixed. The Y-axis represents the completion time in ticks. We notice two things in this graph. Firstly, $HRAND$ improves upon $RAND$, by a factor of 1.75-2. This is expected, and due to the fact that $HRAND$ will preferentially disseminate blocks to higher-bandwidth nodes early in the simulation. This increases the overall service capacity utilized much more rapidly than the bandwidth-agnostic $RAND$ method.

Secondly, Figure 3.13 indicates that $HRAND$ improves upon $GRAND$ too, but to a much smaller extent than over $RAND$. Their performance tends closer to each other because both $HRAND$ and $GRAND$ will preferentially select the higher-bandwidth nodes in the same manner. This is because the high-bandwidth nodes preferred by $GRAND$ are the very same nodes that will have higher *demand*() values in the

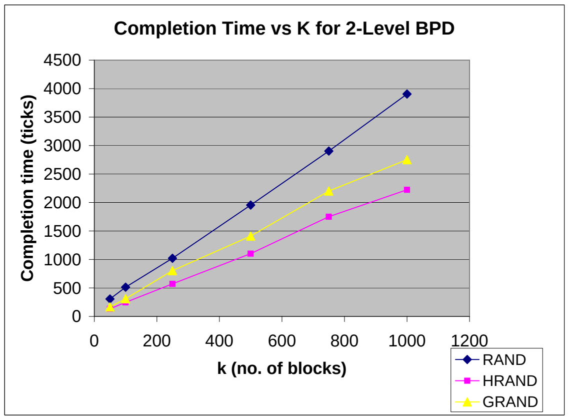


Figure 3.14. Completion Time T vs. K for BPD-A (Two-Level BPD)

$HRAND$ calculation. The low bandwidth nodes will all have $demand()$ values of zero, since the associated weights are all zero. On the other hand, there is a small improvement by $HRAND$ over $GRAND$, since $HRAND$ ensures better service capacity utilization within the high-bandwidth set.

Similarly, we also depict the completion times for BPD-A as k varies, and n is fixed, in Figure 3.14, and observe the performance improvement shown by $HRAND$ in this case also. Again, we notice that $HRAND$ reduces completion time by a factor of around two in many cases compared to $RAND$, and to a much smaller extent compared to $GRAND$.

We now move on to a different BPD: the Clustered BPD. In this particular parameter setting, the bandwidth for a data transmission depends both on the source and the destination of that transmission. It can be either a low value BW_l or a high value BW_h, where the latter is ten times larger than the former. This ratio was chosen to represent, approximately, typical variations observed in bandwidth from the same source to different destinations. We also experiment with other values of this parameter, which we detail shortly. These clients were divided uniformly at random into ten clusters. The peering links between clients were established uniformly at random, given a pre-configured peering-graph degree of 20. The peering links did not

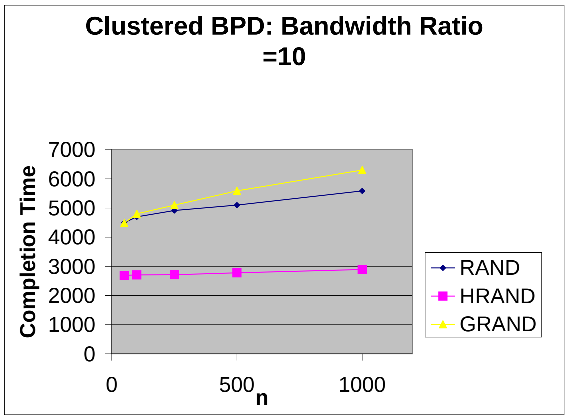


Figure 3.15. Completion Time T vs. n for BPD-B (Clustered BPD)

depend on the clusters. We refer to this Clustered BPD with these parameters as BPD-B.

Figure 3.15 shows the completion times for BPD-B, for all three NSPs of interest: $HRAND$, $GRAND$, and $RAND$. The X-axis varies the value of n, with k being fixed. As with BPD-A, we notice that $HRAND$ improves significantly upon $RAND$, by a factor of 1.6-2.1 However, in this case, $HRAND$ also improves upon $GRAND$ by at least the same extent. In fact, $GRAND$ is slightly worse than $RAND$ at many points.

The reason for this observation is that $GRAND$ will cause a node X to ignore a node in a different "cluster" since that node has low bandwidth to X, whereas $HRAND$ will give it credit for it's high bandwidth to other nodes. In the latter case, blocks enter each cluster relatively early compared to $GRAND$, and then are replicated within several clusters in parallel. Whereas $GRAND$ could actually lead to some clusters waiting for blocks to be fully replicated in the originating clusters, before ever receiving those blocks. This kind of partial serialization, illustrated in Figure 3.12, would also explain why $GRAND$ is sometimes worse than $RAND$.

We need to cover a wide range of feasible deployment scenarios. For example, if we considered the clients to belong to a large enterprise network, and the clusters to

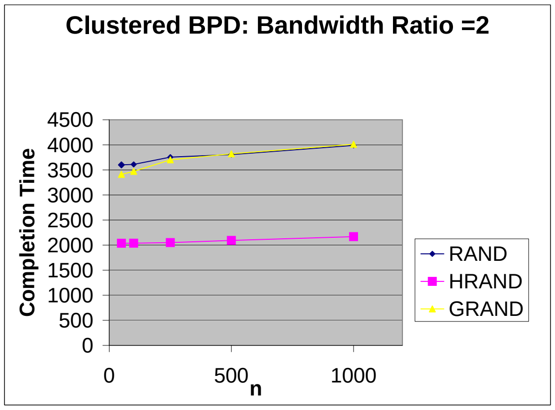


Figure 3.16. Completion Time T vs. n for BPD-C (Clustered BPD)

be different geographically separated sub-networks, the bandwidth parameters would depend on the provisioning and purchase of external connectivity. Therefore, we changed the parameters of the clustered BPD so that the ratio of intra-cluster to inter-cluster bandwidth was reduced to two, and we name this case BPD-C.

As in the case of BPD-B, we found that $HRAND$ improved significantly over $RAND$ and $GRAND$. We noticed, though, that the factor by which $HRAND$ improved the completion time was slightly reduced, compared to BPD-B. This can be seen from Figure 3.16, which depicts completion time of all three NSPs as three different lines. The X-axis again varies the value of n.

3.4 Chapter Summary

In this chapter, we have investigated the performance of content distribution algorithms under the assumption of a cooperative bulk-data distribution scenario, first assuming a simple homogeneous environment, and then generalizing our solutions towards a more complex heterogeneous environment. We first defined our system model, which distinguished us from other related work. Under the initial assumption of a simple homogeneous bandwidth model, we analytically derived an optimal algorithm, based on a hypercube overlay network, that we proved to possess the lowest completion time theoretically achievable. This optimal algorithm is found to be about twice as fast as currently known algorithms, in terms of completion time.

We then began the transition to a more general set of assumptions, by motivating the need for a more flexible asynchronous algorithm. We design such an algorithm based on a randomized matching of nodes. To compare this randomized algorithm to the optimal algorithm, we first estimated its performance in the homogeneous environment, via simulations. We found that the randomized algorithm exhibited nearly optimal scaling behavior, in terms of completion time.

We then completed the transition to a general heterogeneous client-bandwidth environment. We developed a heuristic-based variant of our randomized algorithm designed to maximize the growth of the total service capacity in the system in this environment. We evaluated this algorithm in comparison to other related algorithms, including an approximation of BitTorrent, via simulations. We found that our heuristic-based algorithm significantly reduced completion time, by a factor of 1.25-2, depending on the cases simulated.

Recall that our dissertation began with the goal of investigating distribution algorithms under different scenarios of client behavior. Until this point, we have only considered scenarios that assume that clients are cooperative, and will freely upload data at their maximum possible rate to any client that needs the data. In the next chapter, we will remove that assumption. We will consider a scenario, where clients are not cooperative and might need some incentive to upload data to each other.

Chapter 4

Content Distribution in a Non-Cooperative Client Economy

Our general goals are to investigate algorithms for content distribution in several different scenarios. Each of these represents a different set of assumptions, which we reiterate:

- Clients can be cooperative or non-cooperative.
- The data being distributed can be such that the order of delivery matters or is immaterial to the application. The former is referred to as "bulk data" and the latter is referred to as "in-order" data.
- The clients, and specifically, their bandwidth behavior, can be either homogeneous or heterogeneous.

In the previous chapter, we fixed the first and second assumption, and investigated both possible choices of the third assumption. In other words, we assumed clients were cooperative and the data delivery order was irrelevant. In this chapter, we no longer assume cooperation from the clients. In a scenario where clients need incentives

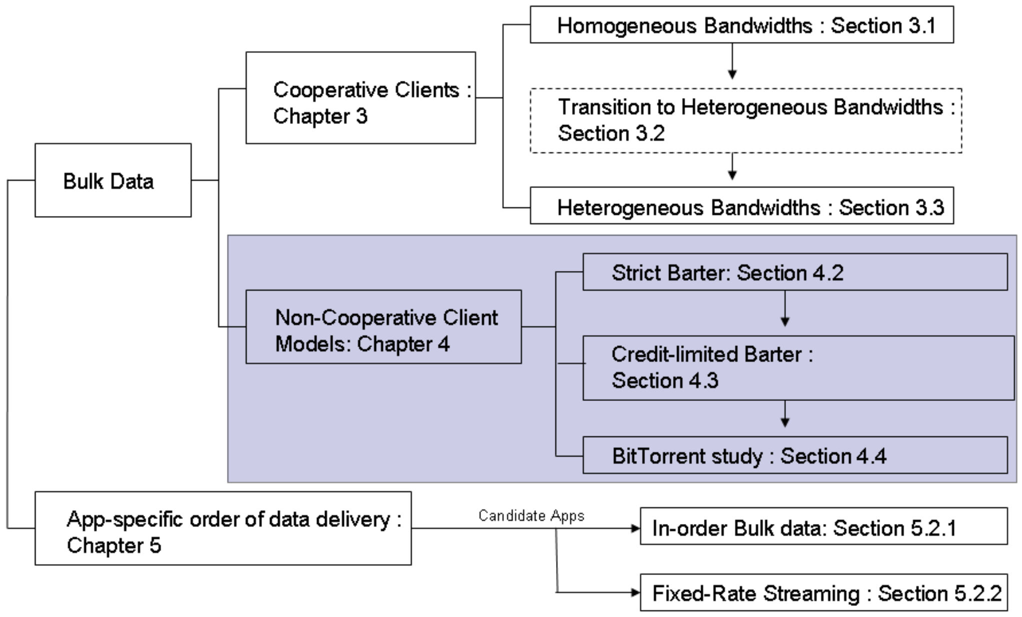

Bulk Data
Cooperative Clients : Chapter 3
Homogeneous Bandwidths : Section 3.1
Transition to Heterogeneous Bandwidths : Section 3.2
Heterogeneous Bandwidths : Section 3.3
Non-Cooperative Client Models: Chapter 4
Strict Barter: Section 4.2
Credit-limited Barter : Section 4.3
BitTorrent study : Section 4.4
App-specific order of data delivery : Chapter 5
Candidate Apps
In-order Bulk data: Section 5.2.1
Fixed-Rate Streaming : Section 5.2.2

Figure 4.1. Design Space and Roadmap of Study

to upload data to each other, and the order of data delivery is still unconstrained, we reprise the issues for investigation:

- Find the best possible completion time for distributing k blocks to n clients, and design the corresponding algorithm, under these assumptions.
- Establish and compare the performance of relevant known algorithms.

We recall our graphical roadmap in Figure 4.1. This chapter covers the newly shaded area of this roadmap. Since we now assume that clients are *not* willing to freely upload data to other clients, we must consider algorithms in which clients are incentivized to upload. This is what we do in Section 4.1. We consider several different mechanisms that we could use for this purpose, and motivate the use and study of mechanisms based on the principle of barter. This principle implies that a client does not upload data to another client unless it receives data in return.

Our objective here is to explore the three-way trade-off between the mechanisms' enforceability, their ability to incentivize uploads, and their efficiency of content distribution. To this end, we consider two different mechanisms based on barter, informally analyze their incentive structure, derive lower bounds analytically and develop actual algorithms for content distribution under the mechanism. We then investigate the completion time performance of these algorithms, either by analysis or simulation, and compare this to the equivalent algorithms in the cooperative case. In essence, we are investigating the *price of barter* - the performance penalty imposed by our incentive mechanisms. It turns out that the exact definition of the barter model significantly impacts performance.

The first specific barter model we investigate is the *strict barter* model, in Section 4.2. In this model, a client transfers data to another only if it simultaneously receives an equal amount of data in return. We prove analytically that even the optimal solution under this model performs poorly when compared with the optimal cooperative solution from Section 3.1.3.4.

In Section 4.3, we perform further analysis and discover that there are indeed mechanisms that could theoretically exhibit low completion times, while still providing robust incentives for uploads. They complete file distribution as quickly as the optimal cooperative algorithms. These mechanisms involve a relaxation of the strict barter model to allow clients to give each other some "credit". At any point in time, a client X will upload data to another client Y, only if the amount of data sent by X to Y until that point does not exceed the amount of data sent by Y to X by some threshold value. We call this model the credit-limited barter model.

We then consider the issue of using these algorithms in a practical environment. We already motivated the need for non-deterministic algorithms based on random matching of nodes. Therefore we propose the integration of the credit-limited barter principle within the basic randomized algorithm of Section 3.2. We call this the randomized barter algorithm and explore its performance via simulations, in Section 4.3.3.

We can indeed obtain near-optimal performance using the randomized barter algorithm. However, these results, reported in Section 4.3.3, are extremely sensitive to parameters such as the degree of the overlay network the randomized algorithm operates on. For example, in many cases, if the degree is low compared to n, the completion time is an order of magnitude worse than the optimal time. Our simulations shed light on the critical value of this parameter, as well as the performance impact of different policies and parameters governing the choice of exchanged data blocks.

BitTorrent also has an incentive mechanism, described in [8]. It does not attempt to enforce any well-defined invariant between nodes. In contrast, our barter-based mechanisms enforce well-defined relationships between nodes, and can be made to perform well. The performance of BitTorrent, in terms of completion time, is not well-understood. Therefore we also explore the performance of BitTorrent, via simulation, and report on the impact of different parameters. This is described in Section 4.4. We find that BitTorrent's performance differs from the optimal cooperative result of Section 3.1.3 by a factor of about 2.2 to 1.3, depending on the choice of certain parameters. While the latter number indicates that we can indeed obtain scaling behavior that is not much worse than the optimal behavior, we can achieve this behavior only with careful tuning. This tuning, however, involves certain parameters that could weaken the incentive structure of BitTorrent. In other words, we find that BitTorrent pays a significant price for its incentive mechanisms, compared to the barter-based mechanisms we propose.

Our focus is *not* on game-theoretic analysis of different mechanisms to identify the optimal strategy for selfish nodes. Rather, our contribution is to devise efficient distribution mechanisms and understand their *performance* in terms of completion time. This performance is subject to the natural and intuitive fairness constraints imposed by a barter-based incentive mechanism, assuming that any algorithm obeying the mechanism will be acceptable to nodes.

We now begin our study with motivation and discussion of the general principle of barter.

4.1 Motivation and Focus of Our Research into the Barter principle

So far, we have assumed that clients freely offer their upload bandwidth to help other clients. While there are scenarios where this might be the case, we also need to consider the cases where clients will not behave in this unselfish manner. In many practical scenarios, clients are unlikely to upload data unless they have an incentive to do so. Consider, for example, an ad-hoc network of peer-to-peer clients, like the Gnutella or BitTorrent based networks. In such networks, since there is no commercial or social relationship between the owners of different clients, why would a client transmit data to another client without being assured of getting something in return? Therefore, we need mechanisms that ensure that it is in the clients' interest to upload data. If there is no commercial relationship between these clients, then the only incentive available in such networks is to associate the clients' upload behavior to good download performance.

Now consider how we can enforce a relationship between a client's upload and download activity. It would be natural to pursue the following proposal based on a centralized monitoring point. We could assume that this centralized monitor, perhaps co-located with the server or tracker, tracks the upload activity of all clients at all times. The monitor would then "cut off" non-uploading clients, by informing other nodes that they should stop providing these clients with data. This would incentivize clients to upload, since they would realize that they could be cut off if they did not upload. Chapter 3 mentioned the use of a tracker in our algorithms, acting as a centralized rendezvous and coordination point. The tracker is a logical location to implement the functionality of the monitor.

However, this mechanism is impractical. First, it is not possible without complicated cryptographic mechanisms for the monitor to verify whether one node uploads to another, since either of the two nodes might "lie". Second, there may be far too many clients for the monitor to track all the time. The monitor would need to handle the load of communicating with every pair of clients in each block transaction. This could require considerable processing power at the monitor, as well as provisioning network connectivity at the monitor so that its latency to the clients was low. This would likely be expensive to provision, and we therefore seek decentralized alternatives. In fact, for the same reason, we have generally sought to minimize the role of the tracker or any point of centralization in all our algorithms in this dissertation.

A better solution to these problems is to make the mechanism decentralized, letting each node decide for itself to whom to upload and when. The upload decision is loosely guided by the principle of barter: X will not upload data to Y unless Y uploads to X in return. This does not require intervention from any central entity like a monitor or a tracker. Therefore, we will focus henceforth only on mechanisms of this type, which will refer to as "decentralized barter" mechanisms.

Another class of solutions exist for this problem which use some form of electronic currency. Such mechanisms, e.g. [59], [52], typically involve a greater degree of complexity and centralization than pairwise barter systems. Hence we restrict our research to decentralized barter-based mechanism. Here, we consider different mechanisms guided by this barter principle, discussing three issues for each:

- How well are nodes incentivized to upload data?
- What is the fundamental efficiency limit imposed by the mechanism on content distribution?
- Are there simple content-distribution algorithms that are optimal, or near-optimal, for that mechanism?

Our answers to the first question are based on intuitive arguments; we do *not* formally analyze the mechanism to identify the dominant strategy for selfish nodes. For the latter two questions, we are concerned with the fundamental obstacles imposed by the mechanism itself on content-distribution, and not with the inefficiency induced by selfish clients' strategic behavior. Thus, we consider any algorithm obeying the mechanism as an "acceptable" solution rather than requiring that the algorithm operate at a Nash equilibrium [48] under the mechanism.

For the second question, in particular, we analyze the different barter models, to see if there are any theoretical limits to the performance of content distribution algorithms under those models. If these limits are not prohibitively poor, we proceed to answer the third question.

For the third question, we design simple distribution algorithms to operate under the models of barter that we consider. In some special cases, we can use analysis to obtain the performance of these algorithms. But, in general, we perform simulations to estimate their performance.

We will now separately delve into two different candidate barter-based models, in Sections 4.2 and 4.3, that can be used for distributing high-volume bulk content. While these models are natural and intuitively easy to construct, their use in this context has not yet been studied adequately, to our knowledge.

In addition, BitTorrent uses an incentive mechanism that could arguably be considered a member of the barter family. It is also very closely related to our entire body of work in this dissertation. Accordingly, a significant contribution of our work is the investigation of BitTorrent's performance in the presence of the tradeoffs imposed by its incentive mechanisms. This is detailed in Section 4.4.

4.2 Strict Barter

We begin our discussion of different barter models with the simplest version. We consider a model where all data transfers between clients occur strictly by barter. Client X will transfer a block to Y only if Y *simultaneously* transfers a block to X. Of course, the blocks received by X and Y must not be possessed by them already. The one exception to barter-based transfers is for the server itself, which uploads data without receiving anything in return. In this sub-section, we discuss the incentives provided by this model and analyze its performance.

4.2.1 Incentive Analysis.

Observe that this mechanism creates a strong incentive for clients to upload data, since this is the only way to receive data from other clients. In particular, a client attempting to limit the *rate* at which it uploads data will experience a corresponding decay in its download rate, thus forcing the client to upload at the maximum rate to obtain best download performance. However, if nodes are capable of subverting the protocol itself, the mechanism can be "broken" by nodes uploading garbage data for the purpose of increasing their increase their download rate. Thus, this mechanism may be inappropriate in such scenarios.

4.2.2 Performance Analysis - A Lower Bound.

We now investigate the limits on the performance of distribution algorithms under the constraints of the strict barter model. In other words, we answer the question: how does the barter constraint affect the completion time T for *any* possible content distribution algorithm? We answer this question by analysis of the *simple homogeneous model* presented in Chapter 3, with the additional constraints of the strict barter model. This analysis yields the following lower bound on completion time T.

Theorem 2. *Any content-distribution algorithm based on strict barter imposes a completion time of at least* $\min(k + n/2, \frac{n-1}{n}(k + n/2 + 1/2))$ *ticks. If the download capacity of nodes* D *is equal to the upload capacity* U*, any content distribution algorithm requires at least* $k + n - 2$ *ticks.*

Proof. Let us first consider the case $D = U = 1$ block/tick. Observe that the first block received by a client must be obtained from the server S, since the client cannot barter for a block without already owning one block. Since there are $n - 1$ clients, there is at least one client, say X, which possesses at most one block after the first

$n-1$ ticks. After this time, X needs to receive at least $k-1$ more blocks. Since its download capacity is only one block per tick, the total time taken for X to receive all its blocks is at least $n-1+k-1 = k+n-2$ ticks.

Now consider the case $D > U$. We use a different argument here. Again, since nodes cannot initiate barter before receiving their first block, and at most one client can receive a block from the server at each step, the number of clients capable of barter after t ticks, for $t \leq n-1$, is at most t. Since barter requires pairs of nodes to exchange data, the maximum number of uploads that can take place at time t is t if t is odd, and $t-1$ otherwise. The maximum number of uploads when $t \geq n$ is n.

We seek to find a lower bound t_c on the completion time. If we perform a summation of the maximum number of block transfers that can occur at each tick, up to the end of tick t_c, we get an upper limit on the total number of block uploads possible by that time. This summation equals $A + B$, where A is the total number of uploads possible until time $t = n-1$, and B is the total number of uploads possible starting from time $t = n-1$ to $t = t_c$. The time taken for these two stages is $t_A = n-1$ and t_B.

Notice that the total number of block-transfers required for all nodes to get all blocks is $k(n-1)$. Since we are trying to get a lower bound on completion time, for a given t_c, $A + B \geq k(n-1)$. If we assume that $t_c \geq n-1$, we can calculate a lower bound on t_c as follows:

- $A = 1+1+3+3+5+5+7+7+\ldots+n-1 = \frac{n(n-1)}{2} - \frac{n-z}{2}$; where $z = 1$ if n is odd, and $z = 0$ if n is even.
- $B \geq k(n-1) - A$
- $t_B \geq \frac{k(n-1)-A}{n}$
- $t_c = t_A + t_B$, therefore $t_c \geq \frac{n-1}{n}(k + \frac{n+1}{2})$ if n is odd, and $t_c \geq k + \frac{n}{2}$ if n is even.

Thus, we have proved the lower bound described in the theorem statement, for all cases but the following. There is only one remaining case, where $t_c \leq (n-2)$: this is not possible, since each node has to get its first block only from the server, due to the restriction of strict barter. Since a server can upload at most one block per tick, it will take at least $n-1$ ticks to get a block to all the $n-1$ clients.

Thus, we have proved the lower bound described in the theorem.

□

The analysis reveals that barter is handicapped by a high start-up cost – it takes $n-1$ ticks before all nodes receive a block and are able to barter. In this period, only $n/2$ blocks are uploaded on average per tick, leading to a high overall cost linear in both k and n.

4.2.3 Algorithm.

We have thus far only discovered a lower bound on the completion time for strict barter. Is this lower bound achievable? As we shall see, the lower bound is moderately tight, under certain assumptions. We devise an algorithm called the Riffle Pipeline, which has a completion time close to the lower bound, as described by the following theorem.

Theorem 3. *If the client download capacity D is at least twice the upload capacity U, it is possible to complete content distribution under barter within $k+n-1$ ticks.*

We first describe the algorithm for the special case $k = n-1$. That is, the number of clients is exactly equal to the number of blocks.

The Riffle Pipeline is described by the data-transfer schedule below.

During each tick t, $1 \leq t < 2n-2$,

- If $t < n$, server S sends block B_t to client C_t.

- For each $1 \leq i \leq n/2 - 1$, if $i \leq (t-1)/2$ and $t - i \leq n - 1$, client C_i barters with client C_{t-i}, giving up block B_i and obtaining block B_{t-i} in return.

Observe that S talks to clients in the sequence $C_1, C_2, \ldots C_n$. Client C_1 also talks to clients in the same sequence, excluding itself, but trailing S by a tick. C_2 also uses the same sequence, excluding C_1 and itself, but trailing C_1 by a tick. The schedule is extended similarly to C_3, C_4, and so on. The algorithm completes in $2n-3 = k+n-2$ ticks.

To illustrate the above schedule, consider client C_1. In the first tick, it receives block B_1 from S. It does nothing in the second tick, while C_2 receives block B_2 from S. Client C_1 then obtains B_2 from C_2 in Tick 3 by bartering block B_1. It gets B_3 from C_3 in Tick 4, again bartering B_1, and so on. Thus, after n ticks, client C_1 obtains all the blocks; all the other nodes also obtain block B_1 by this time. Client C_2 barters with nodes in the same sequence as C_1, but one tick behind. Consequently, C_2 obtains all its blocks after $n + 1$ ticks, while all the nodes also obtain block B_2 by this time. The overall process completes in $2n - 3 =$ ticks. At tick $2n - 3 = k + n - 2$ both clients C_{n-2} and C_{n-1} complete, thus making the completion time $2n - 3$ instead of $2n - 2$.

What happens when the number of blocks is not exactly equal to $n - 1$? If $k = \lambda(n-1)$ for some integer λ, we may simply break up the blocks into λ groups, and distribute them one group at a time. After the first n ticks, client C_1 finishes receiving all blocks in the first group, and may start receiving a block from the tick group from server S. With every additional time tick, one more node finishes receiving blocks from the first group and may join the riffle pipeline for the second group. Similarly, after $2n$ ticks, client C_1 can start off the pipeline for the third group of blocks, and so on. The overall completion time for this algorithm is $(\lambda-1)n+2n-3 = k+n-3+k/(n-1)$.

If $D \geq 2U$, we can shave $\lambda-1$ off the completion time by starting the riffle pipeline every $n-1$ ticks, instead of n ticks. Thus, at time n, node C_1 would be performing barter with client C_{n-1} while simultaneously downloading the first block of the next group from the server. This reduces the completion time to $k+n-2$.

Finally, we arrive at the case where k is an arbitrary number, not necessarily a multiple of n. Let $k = \lambda n + c$. We may then run the same algorithm as earlier for the first λ cycles. After these cycles, there are only c blocks left to be delivered and $n > c$ nodes left. We may break up these n nodes into $\lceil n/c \rceil$ groups, with c nodes in each group, except possibly the last. The original algorithm is then applied to the first of these groups. That is, the server sends the c blocks to the c distinct nodes in the first group, and then lets the nodes exchange data among themselves. Once the server is done with the first group, it moves on to the second group, and so on, until it reaches the last group. If there are exactly c nodes in the last group, the server can again apply the original algorithm and we are done. However, if the number of nodes is less than c, we apply the entire new algorithm *recursively* to solve the problem of distributing c blocks to a number of nodes less than c.

If the download bandwidth $D \geq 2U$, it follows that the new algorithm has a completion time of at most $k+n-1$ to distribute the k blocks over n nodes.

Thus far, we have investigated the strict barter model, and found that its completion time scales linearly with the number of blocks and also the number of clients. In particular, the performance is severely limited by start-up overhead. We now consider alternative barter models, in search of better performance.

4.3 Credit-Limited Barter

In the previous section, we found that strict barter suffered from two issues. First, nodes could cheat the mechanism by uploading junk and receiving legitimate data in return. Second, a high start-up cost is involved. This causes the completion time to grow linearly with n. In comparison, the optimal cooperative completion time of Section 3.1.3.4 was proportional to $\log n$.

Both these problems can be solved by modifying the barter mechanism to allow some *slack*. We use this idea to extend the strict barter model, and define a new model: the credit-limited barter model. In this model, any node X is willing to upload a block to a node Y so long as the net data transferred from X to Y so far is at most s. Thus, a node can get s blocks "for free" from another, but has to upload blocks in return if it wants any more. We refer to this threshold value s as the *credit limit*.

The credit limit allows us to eliminate the start-up problem that we observed in the case of strict barter. Since nodes can get their first block for free, we can design algorithms such that all nodes can receive a block within a logarithmic amount of time. Once nodes get their first block or first few blocks, the rest of the distribution algorithm can proceed quickly, by harnessing the upload capacity of all clients.

In this sub-section, we investigate different aspects of this credit-limited barter model. First, we discuss the incentives provided by this scheme so that clients upload data to each other. We then attempt to analyze the performance limits of this scheme, and obtain results for some special cases. Then we proceed to design randomized algorithms for the general scenario, and investigate their performance by simulations.

4.3.1 Incentive Analysis.

After a few free blocks to start off with, the credit-limited barter model ensured that a node can get data only by uploading data. Therefore, as long as the number of free blocks is not large, this model creates a strong incentive for clients to upload data, similar to the strict barter model. However, we have to answer two questions about the robustness of this model: (a) can we ensure that nodes do not upload junk data to other nodes? (b) how do we ensure that the number of free blocks is small?

For the first question, the answer is yes. To ensure robustness of the incentive scheme, we propose that each block has a cryptographic signature, of which nodes are aware. For example, the server or tracker could provide this data to each node. These signatures allow nodes to independently verify the integrity of data blocks they receive from other nodes. Such signatures are feasible today. In fact similar methods of verifying data-block integrity are used in peer-to-peer networks, detailed in Chapter 2. In the credit limited barter model, nodes receive credit for uploading data only after the uploaded data has been verified by the receiver. Thus there is no longer an incentive to upload junk data. In this aspect, the credit limited barter model is more robust than the strict barter model. In the strict barter model, blocks were sent and received simultaneously. Therefore, received blocks could be verified only after block uploads were finished, which would be too late to be of consequence.

We now address the second question: since a node has a credit limit of s with every other node, it could obtain s blocks from each of them without ever uploading data. If k is less than $s(n-1)$, the node may be able to get away without uploading anything at all! Therefore we have to make sure that the total number of "free" blocks that a node can receive is small.

This problem appears fundamental to any *distributed* incentive scheme. One solution is to impose a *total* credit limit on borrowings of each node but that may be hard to enforce. This could be achieved by using a centralized monitor tracking the total number of blocks received and sent by each node. Furthermore, such block transmissions would have to be verified at both ends of the transmission by the monitor, since either end could lie about it. As we mentioned in Section 4.1, this is likely to impose a great load on the central monitor, which could be expensive.

An alternative solution is to let the server or tracker "dictate" the structure of the overlay network. In our algorithms discussed so far, the tracker is responsible for informing nodes about the IP addresses of other nodes involved in the distribution of a particular file. Given this role, the tracker is ideally situated to ensure that the overlay network obeys certain constraints. In particular, we require that each node has a designated small number of neighbors d, where the node could receive credit. Therefore, a client can receive s free blocks from each of these neighbors. If we designate both s and the graph degree d to be small, then we can ensure that the total number of free blocks ($s * d$) is also small in comparison to the total number of blocks (k).

Simple cryptographic schemes can be used to implement this restriction on overlay graph structure. All this requires is a one-time generation and transfer of a cryptographic token from the tracker to each node, with one token per node. In addition, the tracker would restrict the number of neighbors for a given node A by giving A a limited number of tokens corresponding to its future neighbors. If a node A wished to set up a neighbor relationship with node B, node A would require to show B that A possesses the token for B. This allows the tracker to govern the structure of the overlay graph. The overheads incurred by the token transfers of this scheme occur only when the overlay graph is constructed, or changed in structure. Compare this with the far more expensive tracker-to-node transmissions required for every block transfer, incurred by the other centralized schemes that we have discussed and abandoned.

We now investigate the performance achievable by algorithms operating under the credit-limited model. We first seek to understand any limits on the performance, by analysis. Later we proceed to simulations for a better estimate.

4.3.2 Performance Analysis - Lower and Upper Bounds.

Thus far, we have proposed a model for credit limited barter, and discussed the strengths of its incentive scheme, and other design issues. We now proceed to understand its performance. In particular, we ask the question: is there some fundamental limit on the completion time of *any* algorithm operating under the constraints of the credit-limited barter model?

To answer this question, we analyzed the simple homogeneous model presented in Chapter 3, with the additional constraints of the credit limited barter model. The best lower bound we can show is simply the same as the lower bound for the cooperative case, $k + \lceil \log n \rceil - 1$. In other words, our analysis could not provide any new information in comparison with the cooperative case analysis of Chapter 3. We have not yet made any claims about the tightness of this bound, i.e., whether this lower bound can actually be achieved by any algorithm.

When $s = 2$ and n is power of two ($n = 2^l$), this bound is, in fact, tight. To see this, consider Chapter 3's Hypercube algorithm with $n = 2^l$; each client gets one free block during the first l ticks, which is within the credit limit of $s = 2$. Subsequently, all inter-client transfers are symmetric, ensuring that the maximum credit limit of any client at the end of any tick is one. Since credit for uploads is only granted at the end of the upload, we require $s = 2$ to ensure that content distribution obeys the mechanism. However, the Hypercube algorithm for arbitrary n does not satisfy the credit-limited barter constraints unless s is very large.

The Riffle Pipeline also satisfies the credit-limited barter constraint with $s = 1$, since all transfers between nodes are governed by the strict barter constraint of simultaneous data transfer. In other words, for any value of s, we can run the Riffle Pipeline if we do not have a better-performing algorithm, since $s = 1$ is the strictest possible constraint. Thus, we observe that the Riffle Pipeline's completion time of

$k + n - 2$ is an upper bound on the optimal completion time for credit limited barter, for any value of s.

4.3.3 A Randomized Algorithm for Credit-Limited Barter

We have seen that it may be feasible to have efficient algorithms under credit-limited barter, at least for special values of n. This raises the question of whether we can devise practical algorithms that operate well under this mechanism. Section 3.2 already pointed out the advantages of a non-deterministic asynchronous algorithm based on randomized neighbor selection. Therefore, we modify the randomized cooperative algorithm from Section 3.2 to obey the credit-limited barter constraint.

Once again, we consider nodes connected in an overlay network G. As in Section 3.2, a well-known tracker is required to enable new nodes to join the overlay network. The tracker allows nodes to find other nodes and form neighbor relationships, thus creating an overlay network. The overlay graph is random, i.e., the tracker gives each new node a randomly selected list of neighbors. However, the tracker ensures that the number of neighbors for each node is limited to d. This is achieved using the cryptographic token-based mechanism described earlier in Section 4.3.1.

Whenever a node X is free to upload data to other nodes, X attempts to find a neighbor to upload to, just as in our cooperative algorithm. The node picks a neighbor Y that is interested in X's content, has sufficient download capacity *and is below the credit limit s*. If there are many eligible neighbor choices that satisfy these criteria, then a Neighbor Selection Policy is used to pick one of these choices. Node Y is then given a block chosen according to the block-selection policy. As we saw in Chapter 3, there can be several possible neighbor selection policies. For example, Random Neighbor selection is simple and was found to be effective in many cases, in Chapter 3. We have also discussed different block selection policies earlier: Random and Rarest-First are two examples that we investigate. In fact, we find that the choice of block selection policy has a much greater impact here in the barter-based scenario compared to the cooperative scenario of Chapter 3.

We will henceforth refer to the algorithm described above as the *randomized barter* algorithm. We now proceed to investigate its performance.

4.3.3.1 Results.

We now investigate the performance of the randomized barter algorithm, in terms of completion time. We performed simulations for this purpose. We extended the synchronous simulator described in Section 3.2.2, so that the Neighbor selection mechanism was subject to the credit limited barter constraint. This introduces a new parameter s, or the credit-limit, to our simulations, in addition to the main parameters k, n, and d that we mentioned in Section 3.2.2. We used these simulations to investigate the performance of the randomized barter algorithm for a wide range of values of k, n and s, as well as with different overlay networks G, and block-selection policies. We summarize the main observations here.

As in Section 3.2.2, B is constant and T is expressed in ticks. Unless mentioned otherwise, the clients are homogeneous. We compare the completion timed observed to the best theoretical lower bound that we were able to show, as mentioned in Section 4.3.2. Since this also equals the optimal value of completion time in the *cooperative* case for homogeneous clients, this serves as a useful baseline to compare the performance of the randomized barter algorithm against. In essence we are investigating the *price of non-cooperation.* As we shall shortly see, this price can be made very low!

Impact of Graph Degree We consider our overlay networks to be random regular graphs G, of a given degree d. Earlier in Section 3.2.2, we began by varying n and k and observing completion time, since the results were not too sensitive to parameters like graph degree and block selection policy. However, when simulating the model, we found that the results were particularly sensitive to the graph degree, for each given value of n. Therefore, we demonstrate the impact of the graph degree d, for fixed values of n and k. The first parameter we consider is the graph degree.

Figure 4.2 plots the mean completion time T, with 95% confidence intervals, against d for experiments using $k = n = 1000$, and using Random block selection. Consider the solid line marked *s=1* representing the case where the credit limit threshold s is one block. This line shows that the graph degree d plays a dramatic role in determining completion times. For $d < 80$, the algorithm performs very poorly, with its completion time being off the charts for $d < 50$. However, there is a sharp transition at $d = 80$, after which its performance is nearly optimal, *and identical to the performance in the cooperative case.* An optimal algorithm for this scenario would have a completion time of around 1010, i.e., the base of the Y-axis of this figure.

In Figure 4.2, the performance improves with increasing graph degree, until the performance can no longer improve. This is because it is already at the optimal value. One might imagine that, since each node is entitled to $d * s$ free blocks in total, the improved performance with increasing d is due to the increased total credit, or total free blocks, per node. However, this is not the whole story, as evidenced by the dotted line marked $s * d = 100$, in the same Figure 4.2. In this case, we consider various values of d, but fix the total number of free blocks across all these cases. We achieve this by setting the credit limit s such that $s * d = 100$. In other words, each node is always entitled only to a total of 100 free blocks. We choose the value 100 because, in the case where $s = 1$, we observed near-optimal performance with $d = 100$, or 100

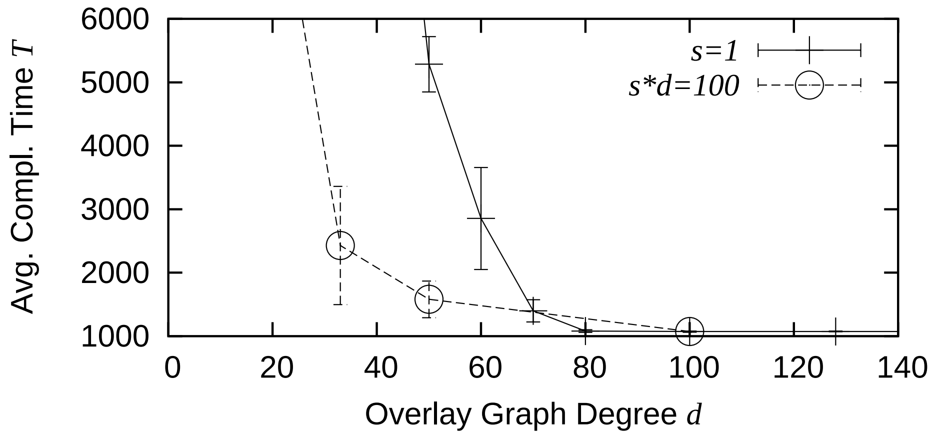


Figure 4.2. T vs. d with $s * d$ kept constant, for Random block selection

free blocks in total credit. So, if the total number of free blocks were the only factor, then 100 should be an adequate number to obtain good performance.

However, as we observe from the dotted line Figure 4.2, there is still a dramatic difference in the observed performance with different values of d. The lower graph degrees continue to exhibit very high completion times even though the total number of free blocks is high, equaling 100. Thus, *the graph degree plays a fundamental role in determining the performance of the randomized algorithm; increasing the credit limit with lower graph degrees is nowhere near as powerful as increasing the graph degree itself.*

In practice, a random graph with degree 80 is likely to be hard to build. Worse still, using even a slightly lower degree may have a serious impact on completion times. We did, however, notice that the effect on the *average* time for nodes to finish is less dramatic than on the completion time.

The Impact of the credit limit s We have observed that with a very low credit limit of 1, the completion time can vary drastically depending on the graph degree. At high degrees, it is clear that increasing the credit limit can have no impact, since

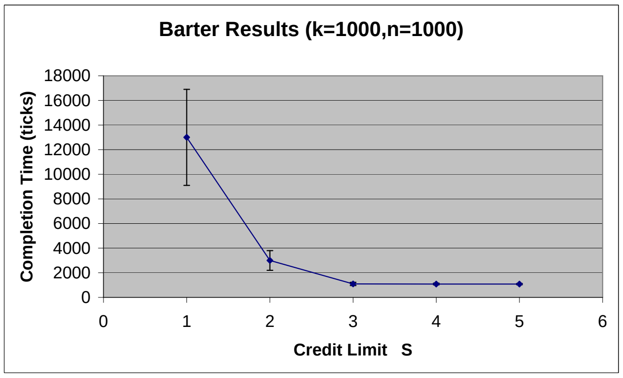


Figure 4.3. T vs. s with $d = 40$ and $k = n = 1000$, for Random block selection

the completion time is already close to its theoretical lower bound. However, at lower degrees, when $s = 1$ was observed to yield poor performance, can we improve the performance by increasing the credit limit?

Figure 4.3 shows that to be true. We vary the value of s along the X-axis, while keeping the degree d fixed at 40, and $k = n = 1000$. The graph plots the completion time T on the Y-axis. As we can see, increasing the completion time does elicit great improvement in completion time, but quickly reaches diminishing returns as s rises beyond three.

We found that the plots of T versus d retained the general shape of Figure 4.2 even when we increase the value of s to 2,3 and so on. However, the degree at which the transition occurs varies with s. In particular, increasing s from one to two has a significant impact on the transition point, with diminishing returns upon increasing s thereafter. For example, in the above scenario, increasing s from one to two, with a graph degree of 40, drops the completion time from 13000 to less than 3000.

All this indicates that it is not necessary to increase the credit limit beyond one or two, to achieve good performance. However, it is important to control the other parameters carefully. This is a useful feature of our algorithm because if we had to increase the credit limit too much, it would allow a large number of total free blocks per node, in turn weakening the incentive mechanism.

The Impact of Block Selection Policy Thus far, we have studied the impact of the degree and the credit limit on completion time, assuming the simplest possible block selection policy: Random block selection. We now consider an alternative block selection policy: Rarest-First. This policy has been described in detail in Section 3.2.1.

To study the impact of the block-selection policy, we repeat the earlier experiments of Figure 4.2. However, this time we use Rarest-First block selection instead of Random block selection. The results are shown in Figure 4.4. The degree d is varied on the X-axis, while the completion time is on the Y-axis. Again, we have a solid line and a dotted line. The solid line is for the case where s is fixed at 1. The dotted line is for the case where s is chosen such that $s * d = 100$.

We observe that these results mimic the results for the Random policy in terms of general behavior, but with a crucial difference: the degree threshold at which the algorithm approaches optimal behavior is now around 20, a fourfold improvement over the degree 80 that was required with the Random policy. In comparison, using a degree-20 network with Random block selection results in a completion time more than 20 times worse.

Thus, *the block-selection policy plays a critical role in determining the completion time.* The results depicted here were obtained assuming that nodes have access to perfect statistics about block frequencies. However, the results are almost identical even using simple schemes for estimating frequencies based on the content of nodes' neighbors.

To summarize these results, we found that the randomized barter could be made

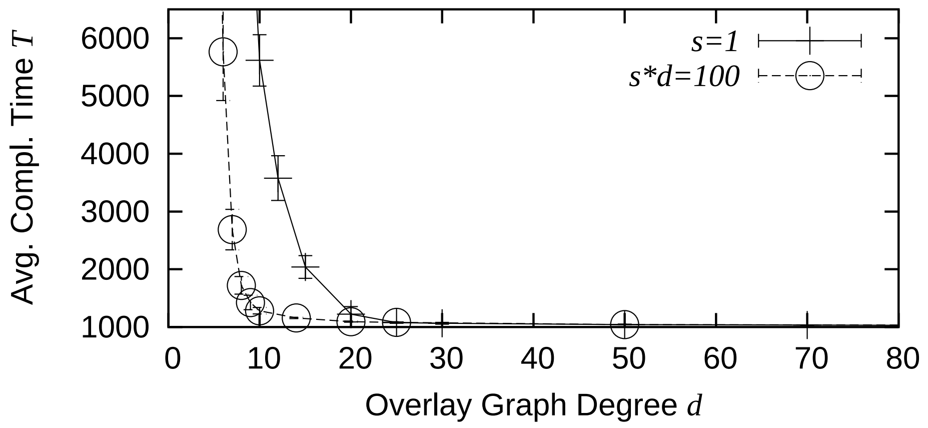


Figure 4.4. T vs. d with $s * d$ kept constant, for Rarest-First block selection

to perform close to its theoretical lower bound, thus reducing the price paid for non-cooperation to almost nothing. However, careful control and selection of factors like graph degree, credit limit and block selection policy were necessary to achieve such performance.

We now proceed to the best known algorithm in this field, namely BitTorrent. We investigate the price it pays for incentivizing cooperation.

4.4 BitTorrent

We have presented our proposal for an incentive scheme, and evaluated its performance. We compared the performance of our algorithms operating with the incentive schemes and without them, estimating the price of non-cooperation. We have yet to address the question of how our incentive schemes perform in comparison to other related incentive schemes and protocols known to the research community.

Among related protocols, the most popular by far is BitTorrent [8]. In fact, BitTorrent served as the inspiration for this dissertation. Our algorithms, in both cooperative and non-cooperative scenarios, resemble BitTorrent in terms of general framework and philosophy. For an example in the context of this chapter's study of non-cooperative clients, and BitTorrent features a tit-for-tat transmission rate-based incentive scheme. This appears similar, on the surface, to the barter-based schemes we have proposed. However, there are differences in detailed design choices and performance goals which significantly distinguish our work from BitTorrent, both in cooperative and non-cooperative scenarios. It is necessary for us to quantify the performance impact of these differences.

The performance of BitTorrent, in terms of completion time parameterized by system size, has not been adequately studied in the research literature to our knowledge. Therefore we undertook a study of BitTorrent's performance under different operating parameters. This study was based on simulations. Since BitTorrent is targeted for a selfish client set, we compared the performance of BitTorrent with our barter-based proposals, for non-cooperative clients. However, since it is among the best-known and best-performing protocols even for the cooperative scenario, we also compare BitTorrent's performance with the optimal hypercube algorithm and the randomized $HRAND$ algorithm from Chapter 3. In fact, we regard the quantification of the gap between BitTorrent's performance and optimal performance to be a significant contribution of our work.

The following sub-sections present the relevant details of BitTorrent's operation and our simulation study.

4.4.1 BitTorrent Incentive Mechanism - Details and Critique

BitTorrent is a widely used peer-to-peer content-distribution system that is meant to operate in an ad-hoc unregulated world of clients. Therefore, it uses an incentive mechanism to encourage these clients to upload data. This incentive mechanism works as follows. A typical BitTorrent client X uploads *simultaneously* to a fixed number of its neighbors on the overlay network, prioritizing neighbors by the rate at which they are uploading to X. While BitTorrent exhibit some "tit-for-tat" behavior thanks to its neighbor priority scheme, there is no well-defined invariant relationship between peers, unlike the mechanisms in Chapter 4. Typical clients almost always upload to a certain minimum number of neighbors at a fixed upload rate, no matter how many neighbors are uploading in return, and at what rate. Selfish clients thus have an incentive to deviate from this behavior and avoid uploading freely to non-reciprocating nodes.

Furthermore, each BitTorrent client X usually uploads to one neighbor *optimistically*, i.e., irrespective of what rate that neighbor uploads data to X. This is meant to allow new clients entering the system to quickly bootstrap themselves to good performance levels. Otherwise, due to the tit-for-tat incentive scheme, a new client Z would never be given priority by its neighbors, since Z would initially have no data to give to its neighbors in return. This is analogous to our credit proposal in Section 4.3. However, the difference between our proposals and BitTorrent is that our incentive scheme is based strictly on an invariant relationship between each pair of nodes. With no such invariants, the BitTorrent protocol is susceptible to exploitation by smart, selfish clients, and would need further safeguards. For example, since there are many circumstances under which a BitTorrent client could receive data from a neighbor without uploading any in return, a selfish client might receive a large amount of data before it is actually forced to upload anything in return. In contrast, our proposals impose a strict bound on the number of free blocks a node can get.

Do we pay a higher price for our incentive schemes than BitTorrent, in terms of performance? That might be true in the highly dynamic environment in which BitTorrent is predominantly used today, since typical deployment scenarios for BitTorrent involve a high rate of churn of BitTorrent clients. Investigation of such an environment remains an open problem, and an interesting area of future work. In this dissertation, however, our target environment involves static, or mostly static, clients. In other words, the rate at which clients join and leave a file distribution network is assumed to be zero, or low enough that it does not affect to distribution time of a single file. It is for this environment that we study the performance of BitTorrent and our proposed algorithms.

4.4.2 BitTorrent Simulation Study

We used an asynchronous simulator to study the performance of the BitTorrent protocol. This simulator assumed typical client behavior in accordance with the BitTorrent specification [8]. We describe the important aspects of this simulator before proceeding to the results.

The synchronous simulator used in Section 3.2.1 cannot adequately capture the time-based actions of the BitTorrent protocol. Specifically, the BitTorrent protocol involves “choke” and “unchoke” messages which might interrupt block transfers, and resume other block transfers. Compare this with our protocols, which do not have any such messages causing interruptions or leading to partial block transfers. Therefore, we had to extend our simulation model to include all the messages from the BitTorrent specification that could possibly affect the completion time of a single file distribution. This led to an asynchronous event-based simulation of the BitTorrent protocol.

We performed these simulations, each for a particular value of k and n. Recall that these are the number of file blocks and nodes respectively. We simulated a wide range of k and n, for the simple homogeneous model. The main result of each simulation was the completion time T. As in Chapter 3, we define a tick to be our basic time unit. As usual, a tick is defined as the time taken for any client to transmit a block at full capacity. Other heterogeneous client-bandwidth models are not considered in this

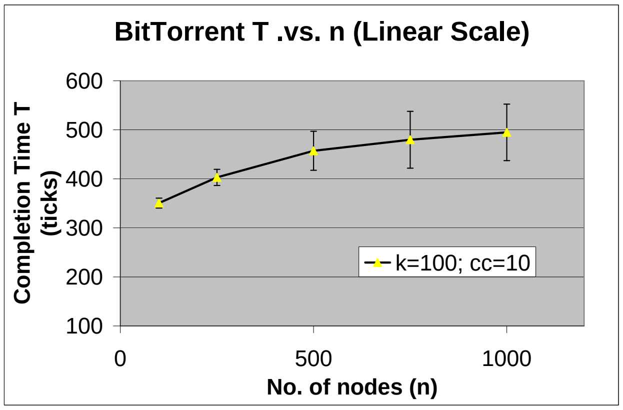


Figure 4.5. BitTorrent Completion Time T vs. n

section. We did investigate the impact of BitTorrent's neighbor-selection policies on a heterogeneous client-set, and these results have already been presented in Section 3.3.

4.4.2.1 BitTorrent Completion Time versus k and n

We aim to estimate BitTorrent's completion time T as a function of k and n to understand the quality of the algorithm. We adopt the same approach we used for the cooperative randomized algorithm of Section 3.2.2. We first vary n across a wide range, while keeping the value of k fixed.

Figure 4.5 plots the mean value of T as a function of n with both axes on linear scales. In this graph, the value of k is fixed at 100. Figure 4.6 plots the same results, but shows the x-axis on a log scale. Comparing these two graphs, we observe that T appears to increase linearly with $\log n$. Replicating the experiment with different values of k produces the same behavior.

Now, we fix the value of n at 100, vary k across a wide range, and show the completion time in Figure 4.7. We see that T increases linearly with k. The same behavior was observed with other values of n too.

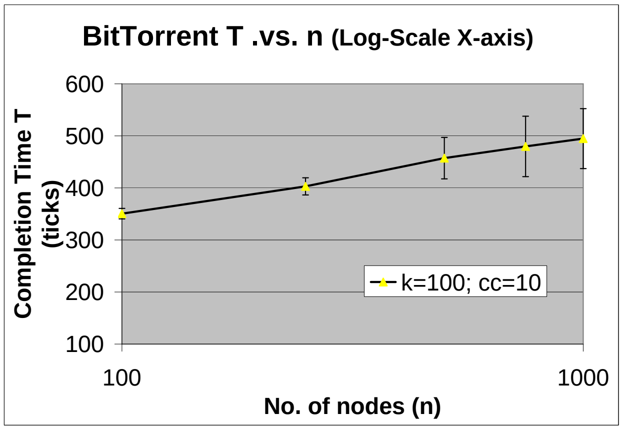


Figure 4.6. BitTorrent Completion Time T vs. n (log-scale X-axis)

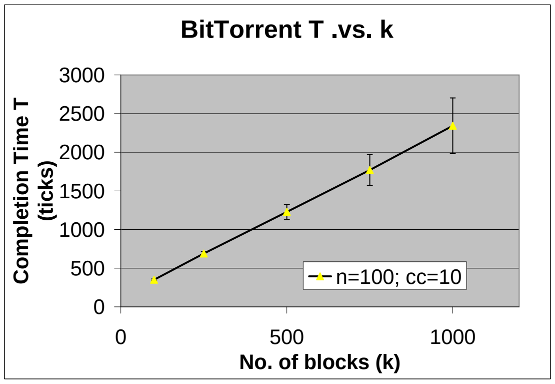


Figure 4.7. BitTorrent Completion Time T vs. k

Given this evidence, we hypothesized that, to a first order of approximation, T is a linear function of k and $\log n$. Using least-square estimation over all data points, we estimate that the expected completion time for BitTorrent is $T = 2.2k + 47.3 \log n - 173.3$. This equation gives us the scaling behavior of BitTorrent with respect to k and n. How does this compare to the scaling behavior of the optimal hypercube algorithm proposed in Section 3.1.3.4, as well as the cooperative randomized algorithm evaluated in Section 3.2.2?

Recall that the optimal algorithm exhibited a completion time of $T = k + \lceil \log n \rceil - 1$ ticks. The basic randomized algorithm's completion time scaled as $T = 1.01k +$

$4.4 \log n + 3.2$. For typical system sizes of $k = 1000$ and $n = 1000$ or even $n = 10000$, the dominant factor in the completion time estimates of all 3 algorithms we consider is the coefficient of k. This implies that BitTorrent is about a factor of 2.2 worse than both the optimal algorithm and the basic randomized algorithm. In addition, for very large values of n, it is possible that BitTorrent's completion time would be even worse, given the large coefficient of n.

The reader might find the difference in completion times of BitTorrent and the basic randomized algorithm surprising, since the two algorithms are similar and based on random overlay graphs. One important reason for this difference is the neighbor prioritization mechanism of BitTorrent, which is implemented by "choking" and "unchoking" neighbors [8], at particular intervals. This leads to many blocks being partially received, and then being completely received only after significant interruptions. Also, BitTorrent nodes upload to four neighbors at a time, which we observed to be slightly worse than uploading to 1 node at a time. The total upload capacity is the same in both cases. Both these differences inherently reduce the rate at which complete blocks are available to serve at nodes, thereby reducing the rate of growth of service capacity in the overlay network. This leads to the poorer completion times of BitTorrent, compared to the basic randomized algorithm, as well as the optimal hypercube algorithm.

4.4.2.2 Dependence on Choking Interval

We have highlighted the performance gap between BitTorrent and the basic randomized algorithm, and attributed it partly to the choking algorithm used in BitTorrent. We now examine if it possible to change or tune this choking mechanism to achieve better performance.

The choking mechanism is characterized by an important parameter, which we shall term the Choking Interval, and denote by the variable CC. This parameter is the length of the time period for which a node transmits data to a certain set of neighbors. At the end of this choking interval, the node chooses anew a set of neighbors to transmit data to, and so on. The length of this interval plays an important role in the performance of this algorithm.

So far, we have only shown the results of simulations where CC is one tick. This is based on the default value of ten seconds indicated for the choking interval in the BitTorrent specification. We translated CC to a value in ticks by assuming that in the typical BitTorrent systems considered by the specification, the upload capacity would be no more than 128 kbps. Assuming a block size of 128KB or 256KB, the time taken to transmit a block at full capacity would be eight seconds or sixteen seconds respectively. This is the length of a tick. Therefore CC would translate to a bit less than one tick in the first case, and a 1.6 ticks in the second case.

We increased the value of CC across a wide range, by more than an order of magnitude. We found that the completion time decreased significantly, until CC rose above 15 ticks. This is the equivalent of 120 seconds. Figure 4.8 shows the completion time on the Y-axis as n is varied on the X-axis, with k fixed at 100, assuming that cc is fifteen ticks. The X-axis is in logarithmic scale. Compare this to Figure 4.6 which shows the same experiments, but assuming CC is one tick. Increasing CC reduces the completion time significantly. T is still seen to increase linearly with $\log n$, however.

Similarly, Figure 4.9 shows the completion time as k is varied on the X-axis, with n fixed at 100, and CC set at 15 ticks. We compare this graph with Figure 4.7, which shows the same experiments, except that CC is set to one tick. Again, we observe a significant reduction in completion time due to increasing CC. This graph also shows that T grows linearly with k.

Repeating our earlier steps, we hypothesized that T is a linear function of k and

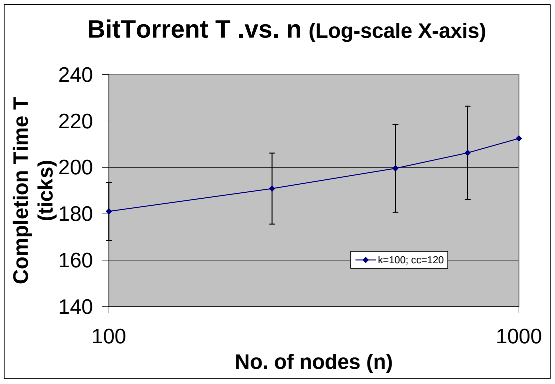


Figure 4.8. BitTorrent's Choking Interval Increased: T vs. n (log-scale X-axis)

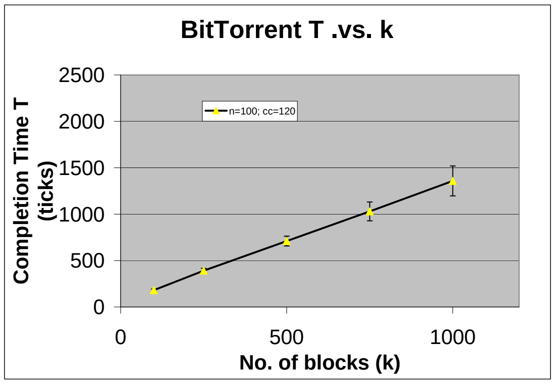


Figure 4.9. BitTorrent's Choking Interval Increased: T vs. k

$\log n$, and used least-squares regression to estimate the completion time for BitTorrent, with $CC = 15$ ticks, as $T = 1.3k + 9.8 \log n - 9$. Since the dominant factor in this equation is likely to be the coefficient of k, this scaling behavior can be characterized as being a factor of 1.3 worse than the optimal algorithm. Recall that we estimated that the default value of $CC = 1$ tick would lead to a completion time that was a factor of 2.2 worse than optimal. Therefore increasing CC leads to significant performance improvement.

However, this improvement is achieved at the risk of weakening the incentive mechanism. The choking interval governs the frequency at which the tit-for-tat mechanism is enforced. Increasing the choking interval means that clients could get away with violating the tit-for-tat principle for a long time. In contrast, our randomized barter algorithm does not suffer from such uncertainties. Tuning this parameter is important to obtain good performance from BitTorrent. Our barter algorithm depends only on the credit-limit parameter s. With the lowest setting of $s = 1$, we can obtain near-optimal performance.

This concludes our discussion of BitTorrent's performance and comparison to our proposed algorithms. After a brief diversion into alternative barter formulations, we will summarize and conclude this chapter on non-cooperative clients.

4.5 Discussion and other Issues in Barter

The barter concept presents interesting extensions and raises important practical issues that we have yet to address. We briefly mention them here, pointing out areas of potentially fruitful future work wherever relevant.

Credit-limited barter is a way of relaxing the strict barter requirement. A more general way to relax the barter requirement is to allow "transitive" use of credit. A node A will upload to B if B is simultaneously uploading to C and C is simultaneously uploading to A. We call this triangular barter. This is more flexible than simple barter, since B can receive data from A even if B does not have data that is useful to A. We could generalize this idea to allow "cyclic barter", involving cycles of any length. But cheat-proof implementation of this generalization is complex, since this nearly converts barter into a cash-based economy. We have already discussed the disadvantages of this class of mechanism, involving a high degree of centralization.

The combination of triangular barter with a credit limit is rather intriguing because it enables provably optimal content distribution with a deterministic algorithm! Observe that Section 3.1.3.5's generalization of the Hypercube algorithm obeys triangular barter with a credit limit $s = 2$. We do not discuss triangular barter any further. The investigation of randomized algorithms for triangular barter, and their potential use in low-degree overlay networks is an area of potential future work.

4.6 Chapter Summary

In this chapter, we have investigated the performance of content distribution algorithms under the assumption that clients are non-cooperative. We motivated the use of barter-based mechanisms to provide incentives for clients to upload data to other clients. We defined two different models of barter. The first was the strict barter model. Our analysis revealed that this model is inherently slow. The second model, credit-limited barter, does not suffer from any such limits. We proposed the incorporation of the credit-limited barter model into the framework of the basic randomized algorithm, and evaluated its performance by simulations. We found that the completion time could be made nearly as low as in the cooperative case, provided that Rarest-First block selection was used. In other words, we could almost completely eliminate the cost of non-cooperation.

Another contribution of this chapter is our estimation of BitTorrent's completion time behavior with respect to the number of blocks and nodes in the system. Via simulations, we found that BitTorrent's completion time was about a factor of 2.2 worse than the optimal cooperative-case completion time. However, the completion time was found to depend on a crucial parameter: the choking interval. By tuning this parameter carefully, and potentially weakening the incentive mechanism, we were able to reduce this factor to 1.3.

This concludes our research in the area of non-cooperative clients. In the next chapter, we will explore a different point of the design space. Until now, we had assumed the file being distributed was bulk data, with no importance attributed to the order in which blocks were delivered. We will now consider scenarios where the order of data delivery is important.

Chapter 5

Application-specific Customization of Block Delivery - *beyond Bulk Data*

This dissertation investigates algorithms for content distribution under different scenarios. Each represents a different part of the design space, which is characterized by the following three dimensions:

- The clients' bandwidth behavior can be either homogeneous or heterogeneous.
- Clients can be either cooperative or non-cooperative;
- The data being distributed is either independent of the order of delivery to the application or not.

So far, we have explored different choices along both the first and second dimension, while making the simplest possible assumption for the third dimension. Thus, the common thread has been our assumption of bulk data delivery. No importance has been attributed to the order in which blocks are delivered to clients, as long as the file transfer latency is low.

In this chapter, we will no longer assume bulk data delivery. We will consider scenarios where the order and time of data delivery is significant to the application. In the context of our graphical roadmap in Figure 5.1, this chapter covers the highlighted area.

Our goal remains the discovery of the best-performing algorithms for ordered delivery. The meaning of performance is now dependent on the semantics of specific applications we might consider. Consequently, the algorithms designed and used might also vary, depending on the application under study. The distribution algorithms best-suited towards these can share some common characteristics, while differing in others. We propose that the shared features be defined and implemented once, while providing an interface to add application-specific features. We present the detailed motivation and proposal of an architecture to achieve this in Section 5.1.

We then consider examples of specific application semantics, and examine the ability of our architecture to handle them, in Section 5.2. We study two candidate applications, which are likely to be important in the future:

- In Section 5.2.1, we motivate and explore the first candidate application, which assumes bulk data, with an ordered list of blocks. While the overall completion time still needs to be minimized, the application prefers that lower-numbered blocks should be delivered before higher-numbered blocks. We name this application "IBULK". We discuss the application-specific algorithm components required to handle these semantics, and evaluate our proposed algorithm via simulations.
- In Section 5.2.2, we explore the second application, which assumes that the data is a constant bit-rate stream. We name this application "ISTREAM". We first explain why this is different from IBULK. We then propose and evaluate the application-specific components of the distribution algorithm for ISTREAM, via simulations.

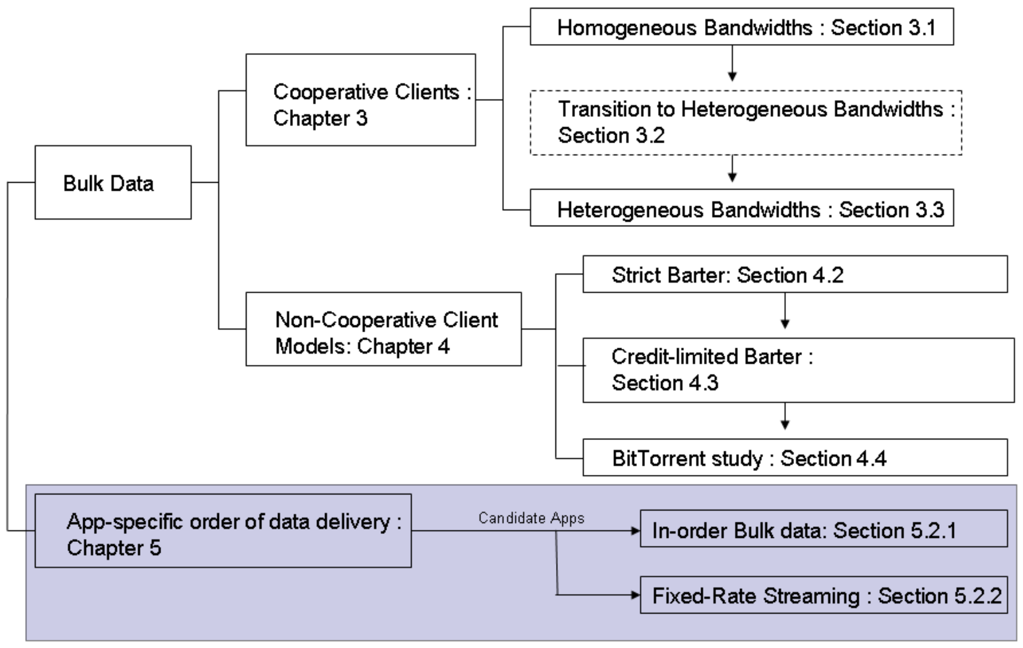

Homogeneous Bandwidths : Section 3.1
Cooperative Clients : Chapter 3
Transition to Heterogeneous Bandwidths : Section 3.2
Bulk Data
Heterogeneous Bandwidths : Section 3.3
Strict Barter: Section 4.2
Non-Cooperative Client Models: Chapter 4
Credit-limited Barter : Section 4.3
BitTorrent study : Section 4.4
App-specific order of data delivery : Chapter 5
Candidate Apps
In-order Bulk data: Section 5.2.1
Fixed-Rate Streaming : Section 5.2.2

Figure 5.1. Design Space and Roadmap of Study: Chapter 5 highlighted

While it is possible to consider more applications in detail, that remains an area of future work. In this dissertation, we restrict ourselves to exploring two applications in detail. Our choices cover different design challenges and serve as illustration of the flexibility of our proposed architecture. Our simulation results indicate that, when the target data rates are near the system capacity, our proposals exhibit significantly better performance than currently known solutions.

This chapter finally concludes with a summary, in Section 5.3.

5.1 Motivation, Proposal and Methodology of our Architecture

5.1.1 Motivation

The bulk content distribution mechanisms we have discussed do not offer any assurances about the order in which blocks are received. This includes our proposal $HRAND$, as well as related work like BitTorrent. Some applications would find useful the ability to specify their block delivery method. For example, a user watching a live video stream of a music concert, without replay ability, would always be interested in blocks at a certain time coordinate of the stream. This user would have very little interest in blocks associated with earlier time coordinates of the stream. In contrast, a user downloading purely bulk data would want to optimize the whole transfer, and not care about the order or time of block delivery. A third possibility might be a hybrid of these two behaviors. There might be users or applications who intend to download a whole file, but wish to start viewing it as soon as possible. A fourth possibility is a live video stream of a sports event or a movie, where the ability to replay is very useful.

Each customization of the downloading protocol could involve a tradeoff. While the block ordering or delivery constraints might be suitably handled, the total completion time might increase. Applications will have different choices in the tradeoff, depending on the intended usage patterns. As a design goal, we provide the ability to customize the delivery method. But at the same time, we need to be able to accommodate a variety of application semantics and requirements.

We have outlined three different applications, and more possibilities exist. Numerous such applications will evolve in the near future, and seek high performance content distribution tailored to their individual requirements. Obviously, it would be cumbersome to develop a new distribution tool for each application. Therefore, it would be useful to possess a basic distribution component that encapsulates the functions common to all these applications. We would then only need to add a few application-specific functions to this basic component.

This basic component must be carefully defined, to ensure that the functionality provided is adequate for a wide range of applications. In addition, the interface to this component dictates the range of application-specific customization that is possible. Therefore this interface needs to be flexible. We propose one definition of the basic component and its interface in the following architectural framework.

5.1.2 The Two-layer Architecture - our Proposal

We propose the following architecture for our content delivery methods. The delivery of the data blocks from the server to the applications residing on the clients goes through two layers. One of these layers is closely tied to application-specific details, and the other layer to application-independent functionality.

The upper layer is the application-specific layer, which contains logic determining what order blocks should arrive in, if some blocks are more urgent than others, and so on. This in turn depends on the intended use of the data. For example, assume that the intended usage implies that a particular set of blocks are more important at the moment than higher-numbered blocks. The upper layer would then convey that preference from the application to the lower layer.

The lower layer is independent of the application behavior. Its function is to provide general methods for high-volume large-scale data delivery. These methods are intended to work reasonably well for all types of application behavior. This layer represents the basic functionality that is common to the distribution methods for many different applications.

The interface between the upper and lower layer needs to be able to convey preferences of the upper layer to the lower delivery layer. It needs to be a "narrow waist", to maintain the application-agnostic nature of the lower layer. At the same time the interface should convey whatever delivery requirements the application needs to convey.

Any interface would have to convey the filename and the source. We propose the following addition, which is simple and general, but powerful. The upper layer associates a priority number with every block. The lower layer needs to be made aware of all blocks' priorities at all times, either explicitly or implicitly. This priority association comprises the "hook" from the upper to the lower layer.

This method provides diverse functionality. For example, streaming video without replay would imply that past blocks have no priority. Upcoming blocks near the application's viewing point have a high priority. If a user wanted to jump to a random point in a video stream, the application would assigning a high priority to blocks near the target point. If a user wanted simple bulk download, all blocks would be equal priority.

At the same time, the interface is simple enough that the lower layer only needs to implement one extra feature in addition to bulk block distribution: it has to consider block priorities when selecting blocks. Therefore, we can use the algorithms discussed in Chapter 3 to distribute data in the lower layer. For example, we would use $HRAND$ if the clients were cooperative and exhibited heterogeneous bandwidth characteristics. The extra priority feature is implemented as a part of the block selection policy. This policy now has to rank the blocks available for transmission in the order of priority and choose the highest priority block. Ties are broken by selecting the rarer block.

Additional messages are required to keep neighbors informed of block priority. We leave open the choice of when these messages will be sent. This represents a tradeoff of the messaging overhead versus how strictly the delivery requirements are obeyed. For example, assume an upper layer with a priority scheme that varies block priorities rapidly compared to the block arrival rate. Then node would have a range of choices regarding the frequency of block priority messages. For example, the node could update neighbors with block priority messages every time a priority changed . An alternative would be sending these messages less frequently, allowing their neighbors to possess stale block priority information. We will discuss this further in the context of specific applications.

We have described our general architecture for in-order application-specific data delivery. We now discuss how the architecture would handle certain specific applications. The following section motivates our choice of applications, and describes our methodology for evaluating each choice.

5.2 Candidate Applications - Methodology and Choices

We have motivated and proposed a two-layer architecture for content distribution. This architecture is flexible enough to develop distribution algorithms that perform well for a wide range of application requirements. We now evaluate the performance of this architecture.

Our approach here is to consider several specific applications with different block delivery requirements. We then ask the following questions, with respect to these applications.

- Can our architecture be customized to provide a solution, or instantiation, that captures the semantics of the application accurately?
- How well does the above solution perform, in terms of the relevant application-specific metrics? How does it compare to the best-known solutions for that application?

For each application we consider, we answer the first question qualitatively. Given a set of application requirements, we propose a priority scheme for the upper layer of our proposed architecture. We examine whether the scheme allows nodes to specify block preferences to their neighbors in a manner that matches the requirements of the application.

The second question has been a constant theme throughout this dissertation. Earlier chapters defined and assumed a sole metric of performance: completion time. Now there is a complication. Completion time is not always sufficient as a metric of performance. Additional metrics might be needed for different applications. For example, a fixed-rate streaming application might place more importance on the loss-rate of the stream than the overall completion time. This is because blocks before the current viewing point are useless to this application, but factor into completion time.

Therefore, for each candidate application, we define our metrics of interest, based on the application semantics. In some cases, these are well known metrics like loss rate. In others, we define a new metric, since the corresponding application requirements are not currently prevalent. We perform simulations to obtain the performance of our algorithms in terms of these metrics. We compare these results to simulations of the best currently known solutions for each candidate application.

The list of applications and variants one can consider is too long to exhaustively investigate. We have already mentioned bulk data and fixed-rate streaming media as possible examples. We could consider different variants of streaming media: (a) a stream of a live event with no jumps from one point of the stream to another; (b) a stream of a past event which allows jumping from point to point; (c) allowing replay but no other kind of jumping; (d) variable rate streams; and (e) layered encoding of streams to use as much capacity as possible.

Apart from streaming media, we could also consider ordered block delivery. This could either be strictly ordered or best-effort ordered delivery. That is, overall completion is sought to be minimized, but blocks should be delivered in order if possible. The latter semantic would be useful when downloading a large movie file that the user wanted to start viewing as soon as possible.

In this dissertation, we restrict ourselves to studying two specific applications in detail. In our opinion, the two applications we choose are the most likely to evolve into popular ones. Indeed, they are extant today, albeit with low-rate data delivery methods. Equally important in our choices was the fact that these two applications illustrate two different design challenges.

The two applications we choose for detailed study are:

- **IBULK:** In-order delivery of bulk data, where the user downloads bulk data, but can obtain additional benefit if the data arrives in order. While overall completion time is still a metric of interest, we will also consider a metric that captures the extent to which the data is ordered. We aim to utilize as much bandwidth as possible from the server and all the clients, and are not satisfied with some fixed rate.
- **ISTREAM:** Fixed-rate streaming media, where the data has to arrive in order at a rate equal to or greater than some minimum rate. We need to minimize the number of losses or gaps in the arriving stream. Given some media rate, it is not beneficial to receive data at a higher rate. This is in contrast to the requirements of IBULK.

We now consider each application in detail, starting with IBULK.

5.2.1 IBULK: In-order delivery of bulk data

We first motivate the choice of this particular application, with examples from popular usage patterns. We will then present our solution for this application's requirements. That is followed by our simulation results and comparison to related work.

5.2.1.1 Motivation and Description of Application

We now consider one specific type of application behavior, and discuss how we'd achieve those semantics with our architecture. We assume that the user downloads bulk data and can eventually benefit from downloading the whole file. We also assume that the user derives additional benefit from downloading the data in order. For example, the user might be downloading a huge high-quality video file like a movie, which she would like to keep on her hard disk for an extended period. But it would certainly be beneficial if the user could start viewing the file as soon as she started the download.

Today, we can observe similar behavior among users of YouTube [64] and Google Video [30]. These are popular websites that allow registered users to upload and store their videos. Subsequently, anyone on the Internet can download and view these videos, provided they have the right software. The format of the video file is typically Flash Video [22]. This allows for the file to be viewed at the same time as it is being downloaded by a client, after a certain initial quantity has been downloaded. Therefore it is important that blocks are downloaded in order at a high rate. Once the video has been downloaded, it can be viewed repeatedly in that session without delay or interruption. In addition the viewer can jump to any point of a fully downloaded file without delay. Therefore downloading the file completely also has its benefits. The default browser plug-ins today typically do not allow the user to save the downloaded Flash Video files from YouTube or Google Video beyond a session. However, it is easy to find extensions [23] that allow the user to save these files onto user-specified locations of the hard-disk, for later viewing.

Usage reports indicate great demand for videos on sites like YouTube, with more than a hundred million videos served per day [65]. However, the size of videos served by YouTube is small, usually a few tens of megabytes. The length of the videos is around a few minutes. While the video sizes can be dictated by the viewer, the video quality is poor when viewed at sizes much larger than a four-inch square. Our goal is to enable the same level of functionality with much larger file sizes, of the order of gigabytes. This would enable longer higher-quality videos that could be viewed full-screen. In addition to formats like Flash Video [22], we also envision using other DVD-quality formats like MPEG-4 [44]. Individual blocks or small subsets of blocks would comprise independently viewable movie segments.

To facilitate the above application, we begin with a standard well-known design point. We provide the application with a buffer that can hold a small number of blocks. When the lower layer receives completed blocks, it sends them to the application buffer. There they are assembled in order. Blocks are to be delivered from the application buffer to the application strictly in order, starting with the lowest numbered block. The blocks might have to wait in the buffer for this purpose, thus introducing delays in the stream of blocks to the application. We assume the availability of enough storage for the entire file in the lower layer.

Since the application buffer is of limited size, long delays between the arrival of consecutive blocks are undesirable. It is also required that blocks are delivered to the application as fast as possible. Therefore the overall completion time still needs to be kept low. We aim to distribute blocks as fast as possible, utilizing as much service capacity from all the clients as possible. This characteristic can be useful when building mechanisms to download video, in cases where the video rate is not known in advance to the designer. This can also be useful when downloading multipart content that is explicitly meant to be viewed or used piece-meal.

Having finished the description of our candidate application, we now describe a distribution algorithm for this application that is based on our two-layer architecture.

5.2.1.2 Solution

The functionality of the lower layer is fixed for all applications. We need to propose an upper layer that is tailored to the IBULK application. This can be achieved by proposing a priority scheme suitable for the application's delivery requirements.

Our proposal is based on a moving window of block priorities. It will henceforth be referred to as the *"priority window"* scheme. The upper layer has a small window of blocks that are the lowest numbered blocks it has not received. The window size is fixed, and this parameter is termed the Priority Window Size. Blocks within the window are higher priority than other blocks. Blocks within the window posses the same high priority value. Blocks outside the window, that have not yet been received, possess the same low priority value.

At a given node X, block priorities can change only upon reception of a block. Assuming the lower layer uses a randomized algorithm like $HRAND$, or even $RAND$ or BitTorrent, block reception results in messages being sent to all neighbors to notify them that X received a new block. Therefore an additional message can be piggybacked indicating a new set of block priorities. Only the blocks whose priorities have changed are indicated in this message. Therefore the increase in messaging overhead is bounded by a low constant factor.

The upper layer fills up the application's block buffer only with blocks in order. Once the buffer is full, blocks can start being delivered to the application. The job of the buffer is to eliminate any waits once the application starts using or viewing the data. It is desirable to reduce the initial time taken to fill up the buffer. That is, if the buffer is of size b_f, the time required for the first b_f blocks to be delivered must be low. In addition the rate of delivery of blocks *in order* subsequently needs to be both high and smooth. Otherwise, the viewer would experience interruptions in his viewing pattern.

We now examine whether this simple proposal can meet these performance goals.

5.2.1.3 Evaluation of Priority Window Scheme

We now evaluate the performance of our priority window scheme by simulations. We use the same synchronous simulator mentioned in Section 3.2.2 to test the priority window scheme. The only area where the randomized algorithm is changed significantly is in the block selection policy. We track two metrics, averaged over all clients:

- The startup time: the time taken to fill up the application buffer initially.
- The maximum sustainable data rate: the best possible rate that can be maintained after a small but reasonable startup delay. To understand this, imagine plotting a graph of block numbers arriving at a client versus time. Take any starting point on the X-axis *under* the graph. The highest slope that a straight line can have while remaining *under* the arrival graph is the best sustainable rate. We desire that our algorithm support a high value of this rate.

When we track these metrics for the simple $RAND$ algorithm, they are so poor that they cannot be shown on the same chart as the Priority Window algorithm! Since $RAND$ deliberately tries to be diverse in selecting blocks, it takes a very long time for the startup buffer to fill up. This is often more than half of the total completion time. The best sustainable rate is meaningless in this scenario. The same argument applies to the default rarest-first algorithm specified by BitTorrent.

We need a non-trivial baseline for comparison, and best known choices, $RAND$ and BitTorrent, are much worse than our proposal. Therefore, we choose an algorithm that is a natural modification of BitTorrent's rarest-first block-selection policy. We call this policy "Rarest-Lowest". This chooses the rarest block, but when there are many equally rare choices, it picks the lowest numbered one, instead of randomizing.

Figure 5.2 shows the maximum sustainable rate, after about 30 ticks, averaged over all nodes. There is one line each plotted for the Priority Window and the Rarest-Lowest algorithms. For the former, the rate is over 90% of the maximum possible unicast transmission rate, for nearly all the data points. This indicates a very high level of performance, since nearly the entire upload capacity of the clients has been utilized.

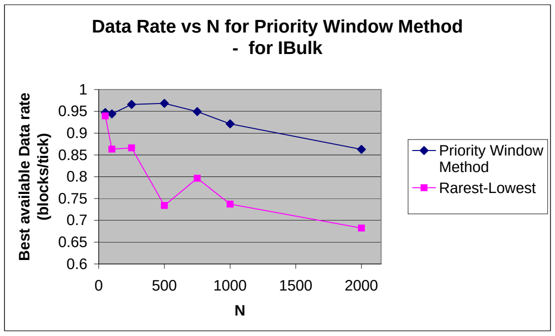


Figure 5.2. Best Sustainable Rate vs. n for IBULK

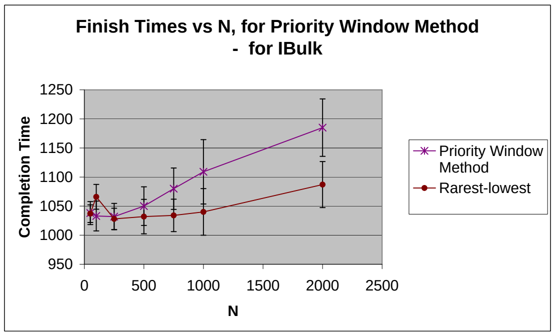


Figure 5.3. Completion Time T vs. n, for IBULK

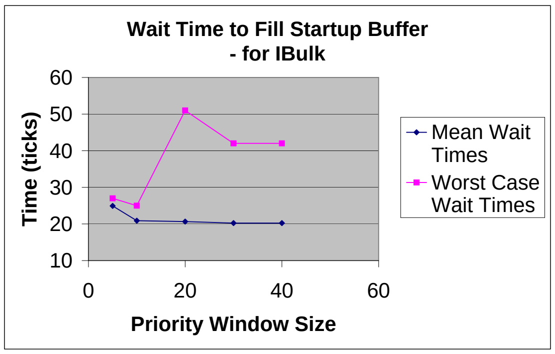


Figure 5.4. Startup Time vs. Priority Window Size

Figure 5.3 shows the average finish time for different values of n for the Priority Scheme and the Rarest-Lowest scheme. We observed that Rarest-Lowest block-selection performed reasonably well compared to $RAND$ and Rarest-First BitTorrent. It does not perform as well as the priority window scheme, in terms of sustainable rate and finish time. The priority window scheme on the other hand performs significantly better in terms of sustainable rate without sacrificing much by way of total completion time.

We can actually observe the tradeoff wherein trying to order blocks results in worse completion times, in Figure 5.3. The finish time of Priority Window is a bit worse than optimal. This is indicated by the Rarest-Lowest scheme doing better. However, the actual percentage increase is less than 5% compared to optimal, given that the Y-axis scale minimum is not zero. The gain in sustainable data rate obtained from this slightly increased finish time is of course seen in Figure 5.2.

Figure 5.4 shows the startup time for different values of the Priority Window size. This is a parameter that affects the functioning of the algorithm. If the Priority Window is too small, there is very little diversity and block availability, and performance is poor. When the priority window gets too large, the algorithm begins to approach simple Random block selection. Therefore it is important to choose the priority window size parameter carefully. This graph illustrates the importance of the priority window. Since we achieve diminishing returns after the X-axis value of 10, we consider a priority window size of 10 blocks to be appropriate. At this setting, both mean and worst-case startup times are less than 25 ticks, which is low given the system size of 1000 nodes and 1000 blocks.

This concludes our investigation of the IBULK application. We now move on to a different application, with different requirements.

5.2.2 ISTREAM: Fixed-rate streaming media

We first motivate the choice of this particular application, with examples from popular usage patterns. We will then present our solution for this application's requirements. That is followed by simulation results and comparison to related work.

5.2.2.1 Motivation and description of Application

We now consider another specific type of application behavior. We assume that the user wishes to view a video stream. She is neither interested in data that is "in the past", nor in storing the data for later viewing. We envision this behavior characterizing people viewing live streams, for example, of concerts or sports events. Several variants, for example, involving replay or TIVO [58] emulation, could also be considered. Such advanced variants are not considered here, and remain future work.

There are several peer-to-peer streaming applications that allow users to watch TV channels online, e.g. TVUPlayer [61],SOPCast [56], [60]. The applications use proprietary technology in several cases. In some cases, they are likely to use methods on multicast or CoolStreaming [67]. These applications currently exhibit the same limitation that YouTube [64] does: small screen size and even poorer quality. The video rates are around 50-500 kbps, depending on the nature of client bandwidths. Higher video rates, and consequently, better quality, is desirable. We seek to increase the video rates that can be supported. To our knowledge, the only published description of algorithms for peer-to-peer television streaming is CoolStreaming. Therefore we compare our methods to the ideas used in CoolStreaming.

To facilitate the above application, we again provide the application with a small block buffer. The buffer is used to hold and assemble blocks in order, waiting if necessary to restore block ordering. Blocks are to be delivered from the buffer to the application in order. Initially the viewer waits until the buffer fills up, and then starts viewing the file. Once viewing commences, blocks are delivered from the buffer to the viewing application at the rate of viewing. The viewing rate is determined by a pre-specified rate of encoding.

If the viewer has arrived at a certain time coordinate of the video stream, and the block for that time has not arrived, it is considered a lost block. We consider such losses tolerable as long as their rate is low. The loss rate is an important application metric.

This application requires that blocks are delivered to the application at a rate higher than some specified minimum rate, which depends on the video encoding parameters. We desire that our system support a high rate of delivery, since that would enable the original video stream to be of high quality. This is another metric for our application: the highest possible rate that can be supported for a given network.

We have finished describing the application requirements. We now proceed to our solution. Like the solution for IBULK, this solution is also based on a sliding priority window.

5.2.2.2 Solution

Since the lower layer and the interface are fixed, all that remains is to define the upper layer. This can be achieved by defining a priority scheme. We propose a window-based priority scheme, which we'll henceforth refer to as *"rate-based window"* or RBW. The upper layer keeps track of what point of the stream the user is currently viewing. It specifies a small window of blocks that are the lowest numbered blocks that have not been received, and that do not lie *before* the current viewing point. The window is fixed size, and this parameter is the Priority Window Size. Blocks in the window are higher priority than all other blocks. Blocks within the window possess the same high priority value: 1. Those outside the window, that are after the viewing point, possess the same lower priority value: 0.5. Blocks that are before the viewing point are at an even lower priority: zero. These will not be downloaded.

The upper layer fills up the applications block buffer. Once the buffer is full, blocks can start being delivered to the application at a fixed rate. The job of the buffer is to reduce any waits once the application starts using or viewing the data. The viewing point in the stream moves forward at a fixed rate once the initial buffer fills up. If the data block corresponding to the current viewing point is not available in the application buffers, it is considered a loss. The total number of losses needs to be minimized for a good user experience. Therefore the loss-rate is our prime metric, given an input viewing rate.

It is also desirable to reduce the initial time taken to fill the buffer. That is, if the buffer is of size b_f, the time required for the first b_f blocks to be delivered needs to be low. Once the buffer fills, the rate of delivery needs to be both high.

Does our proposed solution meet the requirements on these metrics? We investigated this via simulations, and now present the results. As we shall see, our solutions are capable of handling higher rate streams than alternatives based on known techniques.

5.2.2.3 Evaluation of Rate-based Window Scheme

We now evaluate the performance of our rate-based window scheme by simulations. We use the same synchronous simulator that we described earlier in Section 3.2.2 and 5.2.1. We simulate the basic randomized algorithm from Chapter 3 without change, except in the block selection policy. This is where we implement our rate-based window scheme, and other known solutions for comparison.

The primary metric of interest for the ISTREAM application is the total fraction of blocks that failed to be delivered to the viewer by the time she reached that point in the stream. This is the loss rate. We average its value over all clients. This rate needs to be very low for an algorithm's performance to be considered good. This metric is observed for a given viewing rate of the data stream, which is an input parameter.

As in Section 5.2.1, we found that $RAND$ and BitTorrent with its default block selection policy performed so poorly that they did not serve as useful comparison. There are better algorithms to compare our proposal against, namely Rarest-Lowest from Section 5.2.1, and CoolStreaming [67]. CoolStreaming is targeted towards a heterogeneous client-bandwidth scenario. We will consider this shortly. But first, we consider the homogeneous bandwidth scenario. Here neighbor selection is irrelevant, and CoolStreaming is not optimized for this scenario. So we consider the the Rarest-Lowest algorithm.

Figure 5.5 plots the average loss rate of our proposed RBW scheme as well as the Rarest-Lowest block selection policy. This is plotted on the Y-axis, for various values of n on the X-axis. These results are for the simple homogeneous model, and assume

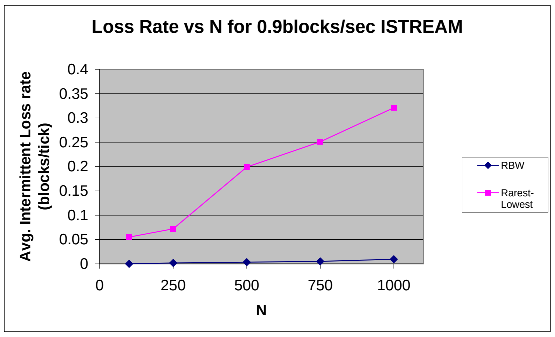


Figure 5.5. Loss Rate vs. N for ISTREAM with rate 0.9 blocks/tick

a data stream viewing rate of 0.9 blocks/tick. We have deliberately chosen a rate higher than aspired to by today's algorithms, since our goal is to improve existing data rates. There are 1000 nodes and 1000 blocks in the system.

Figure 5.5 shows that the Rarest-Lowest policy is not capable of sustaining such a high stream rate without losses. The loss rate climbs to 30%. This would make for a poor viewing experience. On the other hand, our proposal of RBW does much better, with a loss rate that is generally below 3%. While this is not perfect, this leads to a much better viewing experience than Rarest-Lowest. The difference in performance is because Rarest-Lowest distributes a lower numbered block as far as it can before spreading higher-numbered blocks. This leads to too many clients having a lower-numbered block and not enough clients with blocks that are actually in demand. On the other hand, RBW ensures the spread of the entire window of blocks, and therefore allows nodes to have blocks that other nodes want.

We now consider a heterogeneous bandwidth environment. The Neighbor selection policy of any algorithm we consider becomes important. We presented results in Chapter 3 indicating that $HRAND$, in general, was the best choice of Neighbor Selection Policy for heterogeneous environments. Therefore, we incorporate $HRAND$ into the basic functionality of the lower architectural layer. The upper layer, for the ISTREAM application, is *RBW*. We term this algorithmic combination as *"HRAND+"*.

CoolStreaming [67] is the best known peer-to-peer streaming algorithm optimized for heterogeneous bandwidths. Therefore, it is useful to compare the performance of *HRAND+* and CoolStreaming. To incorporate CoolStreaming's principles into our simulation model, we extracted its neighbor and block selection policy, and inserted those components into our randomized algorithm. We then compared the CoolStreaming-derived policies to our proposal, *HRAND+*.

CoolStreaming uses a version of greedy bandwidth-based neighbor selection. Each node ranks its neighbors in decreasing order of bandwidths and chooses the neighbor with the highest rank for requesting block transmission. In addition, the idea of Rarest-First is employed. Blocks with fewer suppliers are preferentially transmitted by CoolStreaming nodes. Only blocks within a sliding window near the viewing point of the stream are considered for this purpose. Therefore, we approximate CoolStreaming by a combination of greedy neighbor selection, and Rarest-first window-based block selection. We term this algorithmic combination *GRW*.

GRW is not an exact model of CoolStreaming. One important difference is that CoolStreaming is a "pull"-based algorithm where nodes request blocks from neighbors. In contrast, our simulation of *GRW* is "push"-based, where nodes with blocks decide which neighbors to serve. In addition, CoolStreaming's use of block deadlines is not perfectly modeled by our simulator, due to the difficulty of porting it to our "push"-based simulations. While a comparison of these algorithms using real-world implementations would be more accurate, that remains an area of future work. However, our simulations of *GRW* yield insight into the aspects of CoolStreaming that perform poorly in certain scenarios, and how they can be improved.

We simulated *HRAND+* and *GRW* on the different heterogeneous models de-

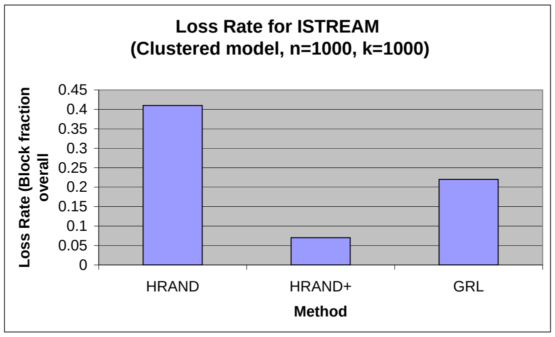


Figure 5.6. Loss Rate vs. N for ISTREAM with rate 0.9 blocks/tick

scribed in Section 3.3. Recall the Clustered BPD that was motivated and described in Section 3.3. We used this heterogeneous model for a simulation of 1000 blocks and 1000 nodes. Figure 5.6 shows the results of these simulations. The clients are divided into logical clusters, where the bandwidth between clients in the same cluster is twice the bandwidth between those in different clusters. We simulate a data stream with a viewing rate of 0.9 blocks/tick. Here, a tick is defined as the time taken for a block to be transmitted between two clients in the same cluster, at full capacity.

Figure 5.6 shows the average loss rate for this system, for both $HRAND+$ and GRW. In addition we also show the performance of $HRAND$, just to show what effect the priority scheme has, isolated from the benefits of $HRAND$. $HRAND$ performs poorly, with around a 40% loss rate. The addition of the RBW priority scheme greatly improves the loss rate, reducing it to around 7% for $HRAND+$. $HRAND+$ is a significant improvement over GRW. This is mainly because GRW's neighbor selection policy is worse than that of $HRAND+$ for this bandwidth model. This was foreshadowed in Section 3.3, where we reported that greedy neighbor selection policies performed poorly for the Clustered BPD. Thus, in combination with an intelligent block selection policy, our proposal outperforms GRW.

We have presented results indicating that our priority-based block-selection schemes work well for the IBULK and ISTREAM application-specific metrics. We now conclude this chapter with a summary.

5.3 Chapter Summary

In this chapter, we considered scenarios where the data and its application depend on the order or time of delivery of individual blocks to the clients. We proposed a two-layer architecture that facilitates the customization of distribution algorithms to a wide range of application requirements. The lower layer is application-agnostic while the upper-layer is application-specific. Our core contribution here was to identify the basic distribution functionality common to all algorithms and the interface provided to this basic functionality. This interface is based simply on a priority associated with every block. We argue that this is a simple but powerful interface.

We demonstrate the flexibility of our architecture by considering two different application examples. For each example, we develop solutions based on the priority interface, and evaluate them via simulations. When simulating systems under the stress of high data rates, these solutions improve significantly on solutions derived from currently known techniques. Typically, our priority window schemes can support a data rate that is around 30% higher than alternative methods.

Chapter 6

Summary and Conclusion

6.1 Summary of Contributions

Content distribution has been a well-studied problem over the last decade. Various algorithms have been proposed by researchers or implemented commercially. The explosion of file-sharing networks around the turn of the century, and multimedia download in particular, has only led to a growth in the pace of research in this area. But prior to our dissertation, the focus was largely on uncoordinated distribution of files to individuals, rather than *coordinated distribution of large files to large communities.*

There have been a few exceptions to this observation, like BitTorrent, which have provided methods for distributing large files to large client sets. However, the usage patterns of BitTorrent typically involve a dynamic ad-hoc set of clients, largely spurred by the promise of free multimedia content. Our target, on the other hand, is a different kind of client base, a well-known static community of clients subscribing to a service. An example of such a service is the timely dissemination of software packages and updates. Completion time is the metric of interest, measuring overall system performance rather than individual client performance. Another example application could involve regular subscribers to popular TV shows or sports broadcasts.

Since this domain has not been targeted by currently known algorithms, our goal has been to devise algorithms optimized for distributing large files to a large number of clients with the lowest completion time. Another goal is to establish the performance of well-known algorithms in this problem domain, so that they could be meaningfully compared. Since our domain still presented a range of possible assumptions, we partitioned it into several sub-domains, depending on client cooperation, bandwidth distribution, and the importance of ordered delivery. We then achieved our goals in each of these sub-domains, establishing the best algorithms, and the principles thereof. Our key results and contributions are as follows, categorized by different scenarios.

Cooperative Clients and Bulk Data:

- For a simple homogeneous client bandwidth model, we analytically obtained a provably optimal algorithm for distributing k blocks to n nodes, for arbitrary values of k and n. This algorithm is based on a hypercube overlay network and guarantees low node degrees. The completion time of this algorithm for k nodes and n blocks is $k + \lceil \log n \rceil - 1$. This is at least twice as fast as multicast-based methods.
- We estimated the completion time behavior of BitTorrent with respect to k and n via simulations, and found that it was about a factor 2.2 times the optimal completion time. We also found that this factor could be reduced to 1.3 with tuning of the choking interval parameter of the BitTorrent algorithm.
- We evaluated the performance of simple randomized algorithm for the homogeneous model and found that the completion time is within 2% optimal for a wide range of k and n.

- We adapted the randomized algorithm for heterogeneous environments using a heuristic aimed at high utilization of high bandwidth nodes' upload capacities.This heuristic performed improved upon the greedy bandwidth-based neighbor selection method employed by BitTorrent. This improvement was as much as a factor of two when the client bandwidth distribution was characteristic of geographic or topological clustering.

Non-Cooperative Clients and Bulk Data:

- We analytically established that the strict barter model is inherently subject to poor scaling behavior. Its completion time grows linearly with k and n.
- We proposed and evaluated the credit-limited barter model via simulation, and found that its completion time was close to the optimal cooperative algorithm's. However, it is crucial to use a high enough graph degree and Rarest-First block selection for this result.

Ordered Data Delivery:

- We then proposed an architecture to allow applications which depend on the order of data delivery to customize our distribution algorithms accordingly. An application-specific upper layer communicates to generic lower distribution layer by indicating a priority for blocks.
- We considered two different candidate applications and show that they can be implemented within our framework. We evaluated our solutions for these applications, based on sliding priority windows, via simulation. We found that our priority schemes can support data stream rates about 25% higher than alternatives derived from BitTorrent, without significant loss rates.

Despite our contributions, several open problems remain. Further, our work is limited in several aspects. This provides a framework and guide for future work, which we cover next.

6.2 Open Problems and Future Work

We now discuss different avenues for future work in large-scale high-volume content distribution.

Real-World Implementation

Using simulations allowed us to explore a large variety of different scenarios and algorithmic variants. However, an implementation would be extremely useful in further development of our algorithms. Firstly, the messaging overhead of these algorithms would be put to the test. Currently, we design algorithms to have low overhead but do not test them further.

Secondly, the processor time required to run our algorithms could be tested using an implementation. None of our algorithms are complex, but the CPU load imposed by them needs to be verified.

Thirdly, a deployment in real-world scenarios could expose our algorithms to realistic client and network bandwidth distributions. We have, so far, evaluated our algorithms on several synthetic models. However, our focus in on large system and file sizes. This is likely to be difficult to achieve in a practical deployment, unless one has content that is sought-after by thousands. A hybrid of these two scenarios is the use of real-world bandwidth traces as a base simulations.

We could achieve a smaller-scale deployment on a wide area testbed like PlanetLab [53]. While this would not stress the scaling behavior, it would still provide proof of practicality. We consider such implementation and deployment as one important direction of immediate future work.

More Application-specific features

As part of our work on allowing applications to customize the order of data delivery, we considered two candidate applications. A natural extension of this work is to consider more candidate applications. In particular, TIVO [58] and similar Digital Video Recorders are rapidly gaining in popularity among television viewers. It would be interesting to prove that our two-layer architecture can successfully provide a distributed emulation of TIVO. For example, a viewer would be able to replay recent segments, skip forward past commercial breaks, pause the stream, etc.

We envision being able to achieve this function simply by using block priority. Blocks behind and ahead of the viewing point, within a pre-defined range would be assigned higher priority. A large application buffer would be required to enable skipping forward past commercials. The following questions arise. If the viewer requests a jump in the viewing stream, how soon can it be satisfied? If multiple jumps are requested in succession, how soon can they be satisfied? Ideally,the answer to these questions would be zero or a short length of time. Preliminary answers to these questions can be obtained via simulation. This is, therefore another direction of work in our near future.

Extensions to $HRAND$

In Section 3.3, we proposed the $HRAND$ algorithm for heterogeneous conditions. This algorithm used DEMAND messages to allow nodes to keep track of their neighbors' values of the "$demand()$" metric. We have assumed that the graph degree is low enough that these messages do not cause significant overhead. Recall that a higher frequency of these messages leads to a higher likelihood that a node will have correct

Figure 6.1. A screen-shot of a game being viewed on TIVO, paused mid-stream (source:Amazon.com)

information about its neighbors. Therefore, the tradeoff between overhead and the algorithm's effectiveness bears further study. In addition, we can consider extending $HRAND$ by propagating a neighbor's DEMAND and block reception messages to other neighbors. This would give each node knowledge about the blocks within a greater radius in the graph. A study of the messaging overheads would be crucial to this research.

Reliability and Dynamicity

We have focused exclusively on static client sets where the rate of arrival and departure of clients into the system is not significant in comparison to file distribution times. When we consider more dynamic situations, BitTorrent is likely to perform well. However, the definition of optimality and a comparison of different algorithms' performance remains an open problem. Therefore, we could pursue some of the goals of this thesis even in the dynamic scenario. While we have not adapted the hypercube algorithm to a general dynamic client set, we foresee being able to run a multi-tiered hypercube algorithm for certain restricted cases of dynamic client sets.

We have assumed throughout this dissertation that no failures occur at nodes or the network. In order to facilitate deployment, additional research into ensuring robustness in the face of such failures would be useful. We anticipate that nodes can respond to the possibility of their neighbors failing simply by increasing the number of neighbors simultaneously connected to, as in BitTorrent. Since our algorithms are based on unstructured graphs, with all neighbors receiving the same state information, we believe that this approach will suffice.

A more challenging problem is that of tracker failure, or failure of connectivity to the tracker. Recall that the tracker is assumed as a central point of rendezvous, allowing nodes to find the server or other client nodes. A failure in the tracker would prevent a node from a joining the system. This issue becomes vastly more important with a dynamic client base.

We could approach this issue by replicating or distributing the information stored at the tracker. For this purpose, we could leverage prior research on robust replication [40, 57]. Alternatively we could combat network and routing failures by providing each node with multiple different paths to reach the tracker.

For the latter approach, we adopt and enhance the principle demonstrated by the Resilient Overlay Network Project [4]. This could be achieved by client nodes acting as overlay-level routers, propagating messages from other clients to the tracker. If some knowledge of the physical network were available or discovered, then the redundant tracker paths can be chosen with the fewest possible physical links in common. Thus, a physical link failure would be unlikely to affect all possible overlay paths from a node to the tracker.

Other Client bases

We have assumed implicitly that the client bases we consider consist of PCs or laptops with reasonable connectivity and bandwidth. The typical client we consider has 128Kbps upload capacity, and at worst 56Kbps from a dial-up connection. Therefore, we focused on files several hundred megabytes large. Smaller file sizes do not pose a challenge. However, if we consider clients with lower bandwidths or intermittent connectivity, content distribution is likely to be a challenge even with lower file sizes. Examples of such clients are mobile phones or sensors. Examples of applications for these clients are video clips and software updates. The investigation of whether our techniques can be applied in the mobile and wireless milieu is a fascinating direction for long-term future work.

Having discussed different open problems and possible approaches to solving them, we now conclude this dissertation.

6.3 Conclusion

Content distribution is an ever-present problem due to its changing nature, a consequence of rising traffic patterns. Growth in the number of home users connected by broadband, and universal connectivity in general, leads to increasingly higher volumes of data consumed. In addition, "killer applications" like Napster and YouTube cause infrequent but tremendous elevations in the overall appetite of the common user for high volume content.

We are one the verge of viable business models being developed for the broadcast of film and television on the Internet. The key to any distribution scheme becoming popular is instant gratification. It is not acceptable to the common user to wait tens of minutes to download a movie. The next frontier is therefore rapid-response large-scale secure distribution of high-quality multimedia content. This dissertation identifies the basic building blocks that lead to high performance, and establishes a common framework to compare different algorithms. Our basic algorithms can be further augmented to provide security and other features. This is our contribution to this arena.

Bibliography

[1] A. Adams, J. Nicholas, and W. Siadak. *Protocol Independent Multicast - Dense Mode (PIM-SM): Protocol Specification (Revised).* Internet Engineering Task Force, draft-ietf-pim-dm-new-v2-01.txt, Feb. 2002. Internet Draft.

[2] Akamai Technologies Inc. Akamai content delivery services. "http://www.akamai.com/".

[3] K. G. Anagnostakis and M. B. Greenwald. Exchange-based incentive mechanisms for peer-to-peer file sharing. In *Proc. 24th International Conference on Distributed Computing Systems*, 2004.

[4] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proc. ACM SOSP '01*, Oct. 2001.

[5] A. Bar-Noy and S. Kipnis. Broadcasting multiple messages in simultaneous send/receive systems. *Discrete Applied Mathematics*, 55(2):95–105, 1994.

[6] A. Bar-Noy, S. Kipnis, and B. Schieber. Optimal multiple message broadcasting in telephone-like communication systems. *Discrete Applied Mathematics*, 100(1-2):1–15, 2000.

[7] D. Bickson, D. Malkhi, and D.Rabinowitz. Efficient large scale content distribution. In *Proc. 6th Workshop on Distributed Data and Structures*, 2004.

[8] Bittorrent protocol specification. http://wiki.theory.org/BitTorrentSpecification.

[9] Cnet news report on bittorrent's studio deal. http://news.com.com/BitTorrent+inks+studio+distribution+deal/2100-1026_3-6070004.html.

[10] Bittorrent.com. http://www.bittorrent.com.

[11] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in a cooperative environment. In *Proc. SOSP*, 2003.

[12] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *USENIX Annual Technical Conference*, pages 153–164, 1996.

[13] Y. Chawathe. *Scattercast: An Architecture for Internet Broadcast Distribution as an Infrastructure Service*. PhD thesis, University of California, Berkeley, Dec. 2000.

[14] B. Chun, Y. Fu, and A. Vahdat. Bootstrapping a distributed computational economy with peer-to-peer bartering. In *Proc. First Workshop on Economics of Peer-to-Peer Systems*, 2003.

[15] Wikipedia entry on coolstreaming. http://en.wikipedia.org/wiki/CoolStreaming.

[16] L. Cox and B. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proc. 19th ACM Symposium on Operating Systems Principles*, 2003.

[17] G. de Veciana and X. Yang. Fairness, incentives and performance in peer-to-peer networks. In *Proc. Allerton Conference on Communication, Control and Computing*, 2003.

[18] S. Deering and D. Cheriton. Multicast Routing in Datagram Internetworks and

Extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[19] Wikipedia entry on fasttrack. http://en.wikipedia.org/wiki/FastTrack.

[20] The fedora core project. http://fedora.redhat.com.

[21] B. Fenner, M. Handley, H. Holbrook, and I. Kouvelas. *Protocol Independent Multicast–Sparse Mode (PIM-SM): Protocol Specification (Revised).* Internet Engineering Task Force, draft-ietf-pim-sm-v2-new-05.txt, Mar. 2002. Internet Draft.

[22] Wikipedia entry on flash video. http://en.wikipedia.org/wiki/FLV.

[23] Video downloader mozilla add-on. https://addons.mozilla.org/firefox/2390/.

[24] A. Fox, S. Gribble, E. Brewer, and E. Amir. Adapting to Network and Client Variability via On-demand Dynamic Distillation. In *Proceedings of ASPLOS-VII*, Cambridge, MA, Oct. 1996.

[25] P. Francis. Yoid: Extending the internet multicast architecture. Unrefereed report, Apr. 2000.

[26] P. Ganesan, Q. Sun, and H. Garcia-Molina. Apocrypha: Making p2p overlays network-aware. Technical report, Stanford University, 2003. http://dbpubs.stanford.edu/pub/2003-70.

[27] C. Gkantsidis and P. Rodriguez-Rodriguez. Network coding for large scale content distribution. In *Proc. IEEE INFOCOM*, 2005.

[28] Gnutella.com. http://www.gnutella.com.

[29] Wikipedia entry on gnutella. http://en.wikipedia.org/wiki/Gnutella.

[30] Google video. http://video.google.com.

[31] R. Han, P. Bhagwat, R. lamaire, T. Mummert, V. Perret, and J. Rubas. Dynamic adaptation in an image transcoding proxy for mobile web browsing. *IEEE Personal Communications Magazine*, pages 8–17, Dec. 1998.

[32] Y. hua Chu, S. Rao, and H. Zhang. A Case For End System Multicast. In *Proceedings of ACM Sigmetrics '00*, Santa Clara, CA, June 2000.

[33] Irc - technical and historical information site. http://www.irc.org.

[34] Apple itunes. http://www.apple.com/itunes.

[35] J. Jannotti, D. K. Gifford, and K. L. Johnson. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI)*, San Diego, CA, Oct. 2000. USENIX.

[36] D. R. Karger, E. Lehman, F. T. Leighton, M. S. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. 29th ACM Symposium on Theory of Computing (STOC 1997)*, pages 654–663, May 1997.

[37] Kazaa.com. http://www.kazaa.com.

[38] Wikipedia entry on kazaa. http://en.wikipedia.org/wiki/Kazaa.

[39] K. Lai, M. Feldman, J. Chuang, and I. Stoica. Incentives for cooperation in peer-to-peer networks. In *Proc. First Workshop on Economics of Peer-to-Peer Systems*, 2003.

[40] M. Leslie, J. Davies, and T. Huffman. Replication strategies for reliable decentralised storage. *ares*, 0:740–747, 2006.

[41] Limewire.com. http://www.limewire.com.

[42] Wikipedia entry on limewire. http://en.wikipedia.org/wiki/limewire.

[43] Movielink. http://movielink.com.

[44] Homepage of the moving picture experts group. http://www.chiariglione.org/mpeg/.

[45] J. Mundinger and R. Weber. Efficient file dissemination using peer-to-peer technology. University Of Cambridge, Statistical Laboratory Research Report 2004-01, 2004.

[46] Napster.com. http://www.napster.com.

[47] Wikipedia entry on napster. http://en.wikipedia.org/wiki/Napster.

[48] Wikipedia entry on nash equilibrium. http://en.wikipedia.org/wiki/Nash_Equilibrium.

[49] T. Ngan, A. Nandi, A. Singh, D. S. Wallach, and P. Druschel. Designing incentives-compatible peer-to-peer systems. In *Proc. 7th International Conference on Electronic Commerce Research*, 2004.

[50] P. Obreiter and J. Nimis. A taxonomy of incentive patterns - the design space of incentives for cooperation. In *Proc. Second International Workshop on Agents and Peer-to-Peer Computing*, 2003.

[51] V. Paxson. End-to-End Routing Behavior in the Internet. *IEEE/ACM Transactions on Networking*, 1997.

[52] Paypal. Website http://www.paypal.com.

[53] The PlanetLab testbed. http://www.planet-lab.org.

[54] D. Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *Proc. SIGCOMM*, 2004.

[55] The fedora project by red hat - list of mirrors. http://fedora.redhat.com/download/mirrors.html.

[56] Sopcast. http://www.sopcst.org.

[57] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *In Proc. ACM Sigcomm '01*, Aug. 2001.

[58] Tivo homepage. http://www.tivo.com.

[59] D. A. Turner and K. W. Ross. A lightweight currency paradigm for the P2P resource market. In *Proc. 7th International Conference on Electronic Commerce Research*, 2004.

[60] Tvants. http://www.tvants.com.

[61] Tvu networks. http://www.tvunetworks.com.

[62] D. Waitzman, C. Partridge, and S. Deering. *Distance Vector Multicast Routing Protocol (DVMRP)*, Nov. 1988. RFC-1075.

[63] X. Yang and G. de Veciana. Service capacity of peer to peer networks. In *Proc. IEEE INFOCOM*, 2004.

[64] Youtube. http://www.youtube.com.

[65] Bbc news report on youtube. http://news.bbc.co.uk/2/hi/technology/5186618.stm.

[66] X. Zhang, J. Liu, B. Li, and T.-S. Yum. CoolStreaming/DONet: A Data-driven Overlay Network for Live Media Streaming. In *Proceedings of IEEE Infocom 2005*, Miami, FL, Mar. 2005.

[67] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. Coolstreaming/donet: A data-driven overlay network for live media streaming. In *Proc. IEEE INFOCOM*, 2005.