Scaling Data Servers via Cooperative Caching

by

Siddhartha Annapureddy

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
January 2008

Prof. David Mazières

नमो तस्स भगवतो अरहतो सम्मासम्बुद्धस्स ।

चत्तारि अरियसच्चानिः

दुक्खम् अरियसच्चम् ।

दुक्खसमुदयो अरियसच्चम् ।

दुक्खनिरोधो अरियसच्चम् ।

दुक्खनिरोधगामिनीपटिपदा अरियसच्चम् ।

Homage to the Blessed, Worthy, Perfectly Enlightened One.

Four Noble Truths:

The Noble Truth of dukkha.

The Noble Truth of the origin of dukkha.

The Noble Truth of the cessation of dukkha.

The Noble Truth of the path leading to the cessation of dukkha.

Dedicated to my Parents

Annapureddy Laxma Reddy

Katta Vijayalakshmi

Acknowledgements

imasmim sati idam hoti, imassuppaadaa idam uppajjati

imasmim asati idam na hoti, imassa nirodhaa idam nirujjhati.

This being, that becomes; this arising, that arises

This not being, that does not become; This ceasing, that ceases.

This thesis has been made possible by many. Firstly, I would like to thank my advisor David Mazières for encouraging me to pursue my lines of interest. Working with him has taught me to approach research with rigour, and to present research with clarity. I would also like to thank Pablo Rodriguez and Christos Gkantsidis for introducing me to the problem of video streaming, for their camaraderie in and out of work, and for their support over the years.

I would also like to thank my labmates, Michael Freedman, Jinyuan Li, Antonio Nicolosi, and Nickolai Zeldovich for making graduate life painless. Mike has helped with Shark, Antonio provided his computer as well as stimulating conversation, and Nickolai gave me patient feedback on my research on countless occasions, apart from educating me on berries and the American west coast. Thanks also to Steve Muir (PlanetLab) and Robert Ricci (Emulab) for their help with experiments on testbeds at crucial times.

I would like to thank my academic advisor, Vijay Karamcheti, for his support and advice during the early years of grad school. Thanks also to my teachers Kamala Krithivasan from undergrad for her support, Ramaiah sir, Surendranath sir, Madhusudan sir, Koteshwar Rao sir from pre-college days for inculcating a love of Science, and Parthasarathy Sir from high school days for encouraging me to pursue Mathematics.

I would like to thank Omkar for the many discussions we had on philosophy. These discussions rejuvenated my interest in Indian philosophy, and made me more rigorous and articulate in debate. I would also like to thank my friends, Vipin for his timely help on numerous occasions, Shashi for most of my photo collection, Raghavan and Rajesh for making my move to Stanford easy, Nishith, David, and Purushotham for all the conversation.

Finally, I would like to thank my Parents for everything that is memorable in my life. For, as the Buddha would say, “Even if one were to establish one’s Mother and Father in lordship and sovereignty over the great earth and abundant treasures, even so one may not repay the Mother and Father. What is the reason for this? Mother and Father do much for their children, they bring them up, nurture them, and show them this world.”

Abstract

In this thesis, we present design techniques – and systems that illustrate and validate these techniques – for building data-intensive applications over the Internet. We enable the use of a traditional bandwidth-limited server in these applications. A large number of cooperating users contribute resources such as disk space and network bandwidth, and form the backbone of such applications. The applications we consider fall in one of two categories. The first type provide user-perceived utility in proportion to the data download rates of the participants; bulk data distribution systems is a typical example. The second type are usable only when the participants have data download rates above a certain threshold; video streaming is a prime example.

We built Shark, a distributed file system, to address the first type of applications. It is designed for large-scale, wide-area deployment, while also providing a drop-in replacement for local-area file systems. Shark introduces a novel locality-aware cooperative-caching mechanism, in which clients exploit each other's file caches to reduce load on an origin file server. Shark also enables sharing of data even when it originates from different servers. In addition, Shark clients are mutually distrustful in order to operate in the wide-area. Performance results show that Shark greatly reduces server load and reduces client-perceived latency for read-heavy workloads both in the wide and local areas.

We built RedCarpet, a near-Video-on-Demand (nVoD) system, to address the second type of applications. nVoD allows a user to watch a video starting at any point after waiting for a small setup time. RedCarpet uses a mesh-based peer-to-peer (P2P) system to provide the nVoD service. In this context, we study the problem of scheduling the dissemination of chunks that constitute a video. We show that providing nVoD is feasible with a combination of techniques that include network coding, avoiding resource starvation for different chunks, and overlay topology management algorithms. Our evaluation, using a simulator as well as a prototype, shows that systems that do not optimize in all these dimensions could deliver significantly worse nVoD performance.

Contents

[*]This work was started while interning at Microsoft Research Cambridge (MSRC), during the summer of 2005.

List of Figures

Chapter 1

Introduction

Efficient distribution of large files over the Internet is key to enabling new functionality. Applications are becoming increasingly more complex and deal with large amounts of data. Good examples of this trend are data distribution for grids and video distribution over the Internet. A lot of new and interesting functionality has been made possible through data mining on big data sets like the one used to power Google maps [30], or the census data for the US [55, 71]. Similarly, while movies were previously distributed through tapes, DVDs etc., video distribution over the Internet is now becoming increasingly popular. Distribution of large files to meet the demands of these new applications is thus an interesting and relevant problem.

The performance of such applications depends largely on the data download rates of the participants. However, this dependence generally falls in one of two categories. With the first type of applications, performance improves proportionally with the download rate. An example of such an application is a grid where a number of clients do some computation on the census data (from the above example). Given the normal case that the client is not limited by other resources

(CPU etc.), the performance of such an application improves with the download rate of the census data. The second type of applications are useful only when the download rate is above a certain threshold. Video streaming is a good example of this type. A user finds streaming video useful only when the client is able to download data at a rate higher than the encoding rate of the video.

In addition, applications that deal in large files display two basic access patterns – random and sequential. Taking the above examples, queries on the census data usually hit random parts of the file. These parts then need to be supplied to the application keeping the latency as low as possible. While fetching the relevant parts of the file, it would be useful to transfer the rest of the file in the background, so as to field further queries from the local cache. On the other hand, in the example of video distribution, an application accesses the file sequentially (though not necessarily from the beginning). The initial latency for fetching the first part should be kept low.

Traditionally, a central server was used to distribute large files for such applications. As the data volume increased, individual servers have been replaced by server farms [9]. But such server farms do not scale well to large numbers of users because of huge operating costs. An alternative approach has been to use Content Distribution Networks (CDNs) [3]. While CDNs scale better than server farms, they are still quite expensive for the average content provider [56].

A relatively new approach towards distributing large files, in a scalable and economical fashion, is peer-to-peer (P2P) systems. The Internet has scaled rapidly to include a growing population of home users. The idea is to exploit the unused resources of these machines (also referred to as nodes, users, clients) spread over the wide area, to support a variety of applications. In such P2P systems, the large file is usually divided into a number of chunks (also called blocks), and are disseminated across the nodes [20, 44, 21, 54]. The nodes themselves are organized into a connected graph, so that each node can find the chunks it requires from amongst the other nodes.

There are a number of challenges in distributing large files over P2P networks.

- An interesting problem is scheduling the dissemination of chunks of a file in an efficient manner. Note that nodes only have partial information about the rest of the system (typically, a small subset of all the nodes), for reasons of scalability. Determining what chunks to fetch in order to benefit the entire system, while having only partial information is a challenging problem.

- Another interesting problem is topology management, i.e., arranging the nodes into an efficient network. The challenge here is to construct an efficient topology based on the characteristics of the underlying network (such as locality), of the nodes (such as upload bandwidth), and of the application (such as which nodes perform well together).

- Another important aspect of P2P systems is security. Given that the nodes are spread across the world, they are mutually distrustful. The challenge is in ensuring data integrity in such an untrusted environment, and also providing privacy and authentication for some applications.

In sections 1.1, 1.2, we will introduce two systems that we built, Shark and RedCarpet, to give an overview of our approach towards addressing these problems. Shark is a distributed file system that addresses the first type of applications, while RedCarpet is a system that provides a near-Video-on-Demand (nVoD) service and addresses the second type of applications.

1.1 Shark: High-performance read-heavy filesystem

In this section, we give a brief overview of Shark, which illustrates distributed applications that require a high-performance, read-mostly filesystem interface. We provide the motivation for such applications, point out the challenges in designing such systems, the techniques used to overcome these challenges, and the advantages provided by Shark over existing systems.

Users of distributed computing environments often launch similar processes on hundreds of machines almost simultaneously. Running jobs in such an environment can be significantly complicated, both because of data-staging concerns and the increased difficulty of debugging. Batch-oriented tools, such as Condor [23], can provide I/O transparency to help distribute CPU-intensive applications. However, these tools are ill-suited to tasks like distributed web hosting and network measurement, in which software needs low-level control of network functions and resource allocation. An alternative is frequently seen on network test-beds such as RON [5] and PlanetLab [60]: users replicate their programs, along with some minimal execution environment, on every machine before launching a distributed application.

Replicating execution environments has a number of drawbacks. First, it wastes resources, particularly bandwidth. Popular file synchronization tools do not optimize for network locality, and they can push many copies of the same file across slow network links. Moreover, in a shared environment, multiple users will inevitably copy the exact same files, such as popular OS add-on packages with language interpreters or shared libraries. Second, replicating run-time environments requires dedicated resources, a scarce resource in a shared test-bed. Programs need sufficient disk space, yet idle environments continue to consume disk space, in part because the owners are loathe to waste the bandwidth and effort required for re-distribution. Third, replicated run-time environments differ significantly from an application's development environment, in part to conserve bandwidth and disk space. For instance, users usually distribute only stripped binaries, not source or development tools, making it difficult to debug running processes in a distributed system.

Shark is a network file system specifically designed to support widely distributed applications. Rather than manually replicate program files, users can place a distributed application and its entire run-time environment in an exported file system, and simply execute the program directly from the file system on all nodes. In a chrooted environment such as PlanetLab, users can even make `/usr/local` a symbolic link to a Shark file system, thereby trivially making all local software available on all test-bed machines.

The big challenge faced by Shark, of course, is scalability. With a normal network file system, if hundreds of clients simultaneously execute a large, 40MB C++ program from a file server, the server quickly saturates its network uplink and delivers unacceptable performance. Shark, however, scales to large numbers of clients through a locality-aware cooperative cache. When reading an uncached file, a Shark client avoids transferring the file or even chunks of the file from the server, if the same data can be fetched from another, preferably nearby, client. For world-readable files, clients will even download nearby cached copies of identical files—or even file chunks—originating from different servers.

Shark leverages a locality-aware, peer-to-peer distributed index [26] to coordinate client caching. Shark clients form self-organizing clusters of well-connected machines. When multiple clients attempt to read identical data, these clients locate nearby replicas and stripe downloads from each other in parallel. Thus, even modestly-provisioned file servers can scale to hundreds, possibly thousands, of clients making read-mostly accesses.

There have been serverless, peer-to-peer file systems capable of scaling to large numbers of clients, notably Ivy [54]. Unfortunately, these systems have highly non-standard models for administration, accountability, and consistency. For example, Ivy spreads hard state over multiple machines, chosen based on file system data structure hashes. This leaves no single entity ultimately responsible for the persistence of a given file. Moreover, peer-to-peer file systems are typically noticeably slower than conventional network file systems. Thus, in both accountability and performance they do not provide a substitute for conventional file systems. Shark, by contrast, exports a traditional file-system interface, is compatible with existing administrative procedures like backup and restore, provides competitive performance on the local area network, and also scales easily to many clients in the wide area.

For workloads with no read sharing between users, Shark offers performance that is competitive with traditional network file systems. However, for shared read-heavy workloads in the wide area, Shark greatly reduces server load and improves client latency. Compared to both NFSv3 [12] and SFS [51], a secure network file system, Shark can reduce server bandwidth usage by nearly an order of magnitude and can provide a 4x-6x improvement in client latency for reading large files, as shown by both local-area experiments on the Emulab [75] test-bed and wide-area experiments on the PlanetLab [60] test-bed.

By providing scalability, efficiency, and security, Shark enables network file systems to be employed in environments where they were previously desirable but impractical.

1.2 RedCarpet: Providing high-quality near-VoD

In this section, we give a brief overview of RedCarpet, which illustrates distributed applications that require a guaranteed data download rate. We present video streaming as a prime example of such an application (a user can watch a video only if the download rate is higher than the encoding rate of the video.) We provide the motivation for such applications, point out the shortcomings of previous approaches, describe the challenges involved, and highlight our contributions towards solving these challenges.

Audio and video together form arguably the most popular form of content. Their popularity can be easily gauged from the amount of audio-video content generated each year. In 2002, there were $47,776$ radio stations generating 70 million hours of original programming, and $21,264$ TV stations generating 31 million hours of original programming, in the United States (US) alone [47]. Given the convenience and the economy of the Internet, it is not surprising that much effort has gone into bringing audio-visual content to the Internet.

There have been a number of paradigms for the dissemination of audio-video over the Internet. One is the live streaming model, where users tune into a server at a particular time for obtaining a specific audio-video, much like regular radio or TV. Another is the blockbuster model, where users obtain the video in its entirety, and then start watching it, much like getting a DVD from a nearby blockbuster store. A third model is Video-on-Demand (VoD), where a user can start watching a video at any time, at the click of a button. Note that the VoD paradigm combines the utility of the DVD to watch a video at any time, with the convenience of the TV in being able to watch at the click of a button. In this thesis, we thus focus only on providing VoD over the Internet in a scalable and efficient manner.

On the Internet, peer-to-peer (P2P) systems have been immensely successful for large scale content distribution. Peer-to-peer systems provide for scalable content distribution without infrastructure support. Current peer-to-peer applications generate a large percentage of the traffic over the Internet and, not surprisingly, a large fraction of that traffic relates to distributing video content [58]. However, with such systems, the users need to download the complete file, and as a result incur a long delay before they can watch the video. Recently systems such as CoolStreaming and others [79, 61, 25] have been very successful in delivering live media content to a large number of users using mesh-based P2P technology.

In this thesis, we will explore the feasibility of using such P2P technologies to provide a VoD service. A P2P VoD service is more challenging to design than a P2P live streaming system (when there are no hard real-time constraints; note that without this constraint, P2P live streaming becomes a special case of the VoD scenario), because the system should allow users arriving at arbitrary times to watch (arbitrary parts of) the video, in addition to providing a low start up delay. That different users might be watching different parts of the video at a given time can greatly impact the efficiency of a swarming protocol. The lack of synchronization among users reduces the block sharing opportunities, and careful design of the block transmission algorithms is required to achieve good performance.

There are two fundamental approaches to building P2P systems – tree-based (push) systems where a tree (or a forest of trees) is usually constructed for dissemination of data [13, 18, 40], and mesh-based (pull) systems where peers exchange random blocks [20, 29, 42]. Recently, mesh-based systems have become very popular with the success of BitTorrent. In comparison with the tree-based approaches which are usually complex, they are much simpler to design. Also, the mesh-based approaches are robust to high rates of churn unlike the tree-based approaches. On the other hand, mesh-based systems usually incur higher control overhead than tree-based systems [78]. Given that the control overhead is moderate in comparison with the amount of data distributed in the VoD scenario, we chose a mesh-based approach for our study.

Note that while mesh-based P2P systems have proved to be efficient for bulk file dissemination, it has been an open question if they are also efficient for providing VoD. The challenge lies in the fact that users of the VoD service need to receive blocks "sequentially" (and not in random order) in order to watch the movie while downloading, and, unlike live streaming, the users may be interested in different parts of the video, and there is competition for system resources between these different parts. The goal then is to design a P2P system which meets these VoD requirements, while maintaining a high utilization of the system resources.

In this thesis, we study algorithms that provide users with a high-quality VoD service, while ensuring efficient utilization of the system resources. We evaluate our algorithms using extensive simulations as well as real-world experiments under different user arrival patterns, and heterogeneous user capacities. We show that naïve, greedy scheduling algorithms provide bad throughput (number of blocks disseminated per unit time). Applying Network Coding[28, 27, 2] over small time-windows of the video (e.g. a *segment* with a few seconds worth of video frames) minimizes the inefficiency resulting from uploading of duplicate content, and reduces the variance in the performance of the nodes.

While network coding solves the scheduling problem within a segment, scheduling across segments (spanning the entire video file) requires algorithms that avoid under-represented video portions. We present an algorithm that avoids the occurrence of such rare segments. The combination of this algorithm with network coding provides good system *throughput* while allowing nodes to download blocks "pseudo-sequentially" (so that the user does not experience interruptions in playback).

The performance of the system, which measures both the utilization of the system resources as well as user experience (low startup delay and sustained playback), depends critically on adaptively constructing proper mesh topologies using efficient peer-matching algorithms. Such algorithms should take into account the content available at each peer as well as their bandwidth. Our algorithm clusters nodes that are interested in the same part of the video, and ensures that a high throughput translates into good user experience.

RedCarpet is a peer-to-peer system that incorporates this combination of network coding, segment scheduling, and peer-matching algorithms to provide a nVoD service.

The rest of this thesis is organized as follows. In chapter 2, we discuss the design and implementation of Shark, and evaluate its performance benefits in wide- and local-area networks. In chapter 3, we discuss the algorithms mentioned above in detail, and evaluate their benefits with a prototype implementation of RedCarpet. We discuss related research in chapter 4, and conclude in chapter 5.

Chapter 2

Shark

Shark is a distributed file system designed to scale a file server by leveraging a locality-aware cooperative cache formed by mutually distrustful clients. In this chapter, we first present an overview of the design of Shark. We then explore the file system interface and consistency model, and go on to discuss the functioning of the cooperative cache formed by the users of the system. We address the security concerns posed by Shark, given that it operates in the wide-area and relies on sharing of data between clients. We also present the use of LBFS-style chunks designed to exploit file commonalities, as part of the cooperative caching mechanism. Finally, we describe the implementation of Shark, and evaluate the performance benefits offered in wide- and local-area networks.

2.1 Shark Design

Shark's design incorporates a number of key ideas aimed at reducing the load on the server and improving client-perceived latencies. Shark enables clients to securely mount remote file systems and efficiently access them. When a client

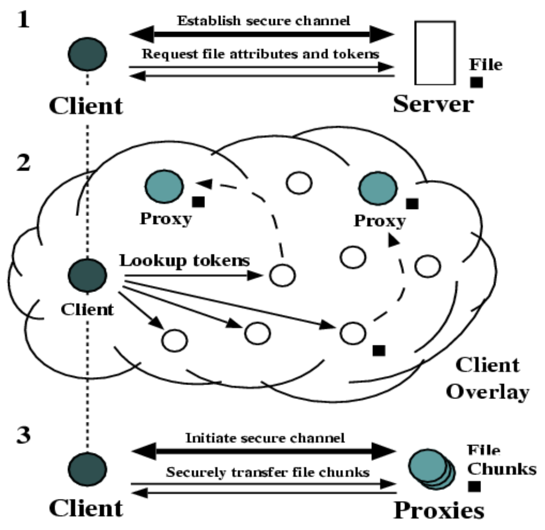


Figure 2.1: *Shark System Overview.* A client machine simultaneously acts as a client (to handle local application file system accesses), as a proxy (to serve cached data to other clients), and as a node (within the distributed index overlay). In a real deployment, there may be multiple file servers that each host separate file systems, and each client may access multiple file systems. For simplicity, however, we show a single file server.

is the first to read a particular file, it fetches the data from the file server. Upon retrieving the file, the client caches it and registers itself as a *replica proxy* (or *proxy* for short) for the "chunks" of the file in the distributed index. Subsequently, when another client attempts to access the file, it discovers proxies for the file chunks by querying the distributed index. The client then establishes a secure channel to multiple such proxies and downloads the file chunks in parallel (Note that the client and the proxy are mutually distrustful.) Upon fetching these chunks, the client also registers itself as a proxy for these chunks.

Figure 2.1 provides an overview of the Shark system. When a client attempts to read a file, it queries the file server for the file's attributes and some opaque tokens (Step 1 as shown). One token identifies the contents of the whole file, while other tokens each identify a particular *chunk* of the file. A Shark server divides a file into chunks by running a Rabin fingerprint algorithm on the file [53]. This technique splits a file along specially chosen boundaries in such a way that preserves data commonalities across files, for example, between file versions or when concatenating files, such as building program libraries from object files.

Next, a client attempts to discover replica proxies for the particular file via the Shark's distributed index (Step 2). Shark clients organize themselves into a key/value indexing infrastructure, built atop a peer-to-peer structured routing overlay [26]. For now, we can visualize this layer as exposing two operations, *put* and *get*: A client executes *put* to declare that it has something; *get* returns the list of clients who have something. A Shark client uses its tokens to derive *indexing keys* that serve as inputs to these operations. It uses this distributed index to register itself and to find other nearby proxies caching a file chunk.

Finally, a client connects to several of these proxies, and it requests various chunks of data from each proxy in parallel (Step 3). Note, however, that the clients themselves are mutually distrustful, so Shark must provide various mechanisms to guarantee secure data sharing: (1) Data should be encrypted to preserve confidentiality and should be decrypted only by those with appropriate read permissions. (2) A malicious proxy should not be able to break data integrity by modifying content without a client detecting the change. (3) A client should not be able to download large amounts of even encrypted data without proper read authorization.

Shark uses the opaque tokens generated by the file server in several ways to handle these security issues. (1) The tokens serve as a shared secret (between client and proxy) with which to derive symmetric cryptographic keys for transmitting data from proxy to client. (2) The client can verify the integrity of retrieved data, as the token acts to bind the file contents to a specific verifiable value. (3) A client can "prove" knowledge of the token to a proxy and thus establish read permissions for the file. Note that the indexing keys used as input to the distributed index are only derived from the token; they do not in fact expose the token's value or otherwise destroy its usefulness as a shared secret.

Shark allows clients to share common data segments on a sub-file granularity. As a file server provides the tokens naming individual file chunks, clients can share data at the granularity of chunks as opposed to whole files.

In fact, Shark provides *cross-file-system sharing* when tokens are derived solely from file contents. Consider the case when users attempt to mount `/usr/local` (for the same operating system) using different file servers. Most of the files in these directories are identical and even when the file versions are different, many of the chunks are identical. Thus, even when distinct subsets of clients access different file servers to retrieve tokens, one can still act as a proxy for the other to transmit the data.

In this section, we first describe the Shark file server (Section 2.1.1), then discuss the file consistency provided by Shark (2.1.2). Section 2.1.3 describes Shark's cooperative caching, its cryptographic operations, and client-proxy protocols. Finally, we present Shark's chunking algorithm (2.1.4) and its distributed index (2.1.5) in more depth.

2.1.1 Shark file servers

Shark names file systems using self-certifying pathnames, as in SFS [51]. These pathnames *explicitly* specify all information necessary to securely communicate with remote servers. Every Shark file system is accessible under a pathname of the form:

$$/\texttt{shark}/@\mathit{server}, \mathit{pubkey}$$

A Shark server exports local file systems to remote clients by acting as an NFS loop-back client. A Shark client provides access to a remote file system by automounting requested directories [51]. This allows a client-side Shark NFS loop-back server to provide unmodified applications with seamless access to remote Shark file systems. Unlike NFS, however, all communication with the file server is sent over a secure channel, as the self-certifying pathname includes sufficient information to establish a secure channel.

System administrators manage a Shark server identically to an NFS server. They can perform backups, manage access controls with little difference. They can configure the machine to taste, enforce various policies, perform security audits etc. with *e*xisting tools. Thus, Shark provides system administrators with a familiar environment and thus can be deployed painlessly.

2.1.2 File consistency

Shark uses two network file system techniques to improve read performance and decrease server load: leases [31] and AFS-style whole-file caching [36]. When a user attempts to read any portion of a file, the client first checks its disk cache. If the file is not already cached or the cached copy is not up to date, the client fetches a new version from Shark (either from the cooperative cache or directly from the file server).

Whenever a client makes a read RPC to the file server, it gets a read lease on that particular file. This lease corresponds to a commitment from the server to notify the client of any modifications to the file within the lease's duration. Shark uses a default lease duration of five minutes. Thus, if a user attempts to reads from a file—and if the file is cached, its lease is not expired, and no server notification (or *callback*) has been received—the read succeeds immediately using the cached copy.

If the lease has already expired when the user attempts to read the file, the client contacts the file server for fresh file attributes. The attributes, which include file permissions, mode, size, etc., also provide the file's modification and inode change times. If these times are the same as the cached copy, no further action is necessary: the cached copy is fresh and the client renews its lease. Otherwise, the client needs to fetch a new version from Shark.

While these techniques reduce unnecessary data transfers when files have not been modified, each client needs to refetch the entire file after any modification from the server. Thus, large numbers of clients for a particular file system may overload the server and offer poor performance. Two techniques alleviate the problem: Shark fetches only modified chunks of a file, while its cooperative caching allows clients to fetch data from each other instead of from the server.

While Shark attempts to handle reads within its cooperative cache, all writes are sent to the origin server. When any type of modification occurs, the server must invalidate all unexpired leases, update file attributes, recompute its file token, and update its chunk tokens and boundaries.

We note that a reader can get a mix of old and new file data if a file is modified *while* the reader is fetching file attributes and tokens from the server. (This condition can occur, for example, when fetching the file tokens requires multiple RPCs, as described next.) However, this behavior is no different from NFS, but it could be changed using AFS-style whole-file overwrites [36].

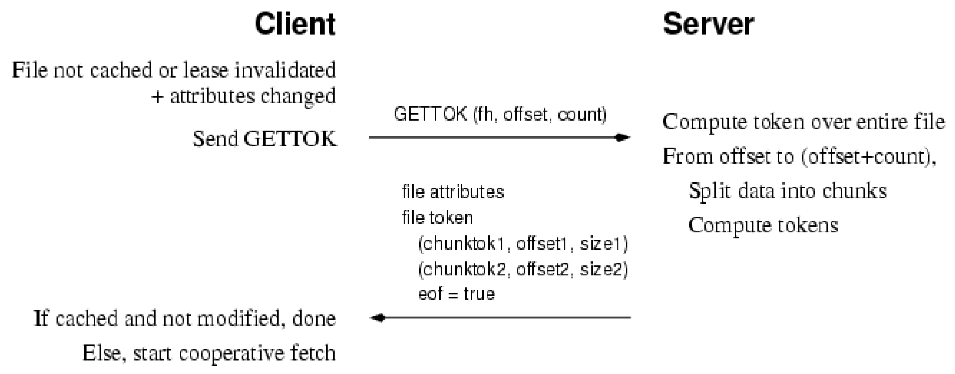

Client
Server
File not cached or lease invalidated
+ attributes changed
Send GETTOK
GETTOK (fh, offset, count)
Compute token over entire file
From offset to (offset+count),
Split data into chunks
Compute tokens
file attributes
file token
(chunktok1, offset1, size1)
(chunktok2, offset2, size2)
eof = true
If cached and not modified, done
Else, start cooperative fetch

Figure 2.2: Shark GETTOK RPC

2.1.3 Cooperative caching

File reads in Shark make use of one RPC procedure not in the NFS protocol, GETTOK, as shown in Figure 2.2.

GETTOK supplies a file handle, offset, and count as arguments, just as in a READ RPC. However, instead of returning the actual file data, it returns the file's attributes, the *file token*, and a vector of *chunk descriptions*. Each chunk description identifies a specific extent of the file by offset and size, and includes a *chunk token* for that extent. The server will only return up to 1,024 chunk descriptions in one GETTOK call; the client must issue multiple calls for larger files.

The file attributes returned by GETTOK include sufficient information to determine if a local cached copy is up-to-date (as discussed). The tokens allow a client (1) to discover current proxies for the data, (2) to demonstrate read permission for the data to proxies, and (3) to verify the integrity of data retrieved from proxies. We will now specify how Shark's various tokens and keys are derived.

Symbol	Description	Generated by ...	Only known by ...
F	File		Server and approved readers
F_i	ith file chunk	Chunking algorithm	Parties with access to F
r	Server-specific randomness	$r = \mathsf{PRNG}()$ or $r = 0$	Parties with access to F
T	File/chunk token	$tok(F) = \mathsf{HMAC}_r(F)$	Parties with access to F/F_i
$\mathsf{I}, \mathsf{E}, \mathsf{A}_C, \mathsf{A}_P$	Special constants	System-wide parameters	Public
I	Indexing key	$\mathsf{HMAC}_T(\mathsf{I})$	Public
r_C, r_P	Session nonces	$r_C, r_P = \mathsf{PRNG}()$	Parties exchanging F/F_i
$Auth_C$	Client authentication token	$\mathsf{HMAC}_T(\mathsf{A}_C, C, P, r_C, r_P)$	Parties exchanging F/F_i
$Auth_P$	Proxy authentication token	$\mathsf{HMAC}_T(\mathsf{A}_P, P, P, r_P, r_C)$	Parties exchanging F/F_i
K_E	Encryption key	$\mathsf{HMAC}_T(\mathsf{E}, C, P, r_C, r_P)$	Parties exchanging F/F_i

Table 2.1: Notation used for Shark's tokens, keys, and other values

Content-based naming. Shark names content with cryptographic hash operations, as given in Table 2.1.

A *file token* is a 160-bit value generated by a cryptographic hash of the file's contents F and some optional per-file randomness r that a server may use as a key for each file (discussed later):

$$T_F = tok(F) = \mathsf{HMAC}_r(F)$$

Throughout our design, HMAC is a keyed hash function [10], which we instantiate with SHA-1. We assume that SHA-1 acts as a collision-resistant hash function, which implies that an adversary cannot find an alternate input pair that yields the same T_F.[1]

[1]While our current implementation uses SHA-1, we could similarly instantiate HMAC with SHA-256 for greater security.

The *chunk token* T_{F_i} in a chunk description is also computed in the same manner, but only uses the particular chunk of data (and optional randomness) as an input to SHA-1, instead of the entire file F. As file and chunk tokens play similar roles in the system, we use T to refer to either type of token, as the context necessitates.

The *indexing key* I used in Shark's distributed index is simply computed by $\mathsf{HMAC}_T(\mathsf{I})$. We key the HMAC function with T and include a special character I to signify indexing. More specifically, I_F refers to the indexing key for file F, and I_{F_i} for chunk F_i.

The use of such server-selected randomness r ensures that an adversary cannot guess file contents, given only I. Otherwise, if the file is small or stylized, an adversary may be able to perform an offline brute-force attack by enumerating all possibilities.

On the flip-side, omitting this randomness enables cross-file-system sharing, as its content-based naming can be made independent of the file server. That is, when r is omitted and replaced by a string of 0s, the distributed indexing key is dependent only on the contents of F: $I_F = \mathsf{HMAC}_{\mathsf{HMAC}_0(F)}(\mathsf{I})$. Cross-file-system sharing can improve client performance and server scalability when nearby clients use different servers. Thus, the system allows one to trade-off performance improvement with additional security guarantees. By default, we omit this randomness for world-readable files, although configuration options can override this behavior.

The cooperative-caching read protocol. We now specify in detail the cooperative-caching protocol used by Shark. The main goals of the protocol are to reduce the load on the server and to improve client-perceived latencies. To this end, a client tries to download chunks of a file from multiple proxies in parallel. At a high level, a client first fetches the tokens for the chunks that comprise a file. It then contacts nearby proxies holding each chunk (if such proxies exist) and downloads them accordingly. If no other proxy is caching a particular chunk of interest, the client falls back on the server for that chunk.

The client sends a GETTOK RPC to the server and fetches the whole-file token, the chunk tokens, and the file's attributes. It then checks its cache to determine whether it has a fresh local copy of the file. If not, the client runs the following cooperative read protocol.

The client always attempts to fetch k chunks in parallel. We can visualize the client as spawning k threads, with each thread responsible for fetching its assigned chunk.[2] Each thread is assigned a *random* chunk F_i from the list of needed chunks. The thread attempts to discover nearby proxies caching that chunk by querying the distributed index using the primitive $get(I_{F_i} = \mathsf{HMAC}_{T_{F_i}}(\mathsf{I}))$. If this *get* request fails to find a proxy or does not find one within a specified time, the client fetches the chunk from the server. After downloading the entire chunk, the client announces itself in the distributed index as a proxy for F_i.

If the *get* request returns several proxies for chunk F_i, the client chooses one with minimal latency and establishes a secure channel with the proxy, as described later. If the security protocol fails (perhaps due to a malicious proxy), or the connection to the proxy fails, or a newly specified timeout occurs, the thread chooses another proxy from which to download chunk F_i. Upon downloading F_i, the client verifies its integrity by checking whether $T_{F_i} \overset{?}{=} tok(F_i)$. If the client fails to successfully download F_i from any proxy after a fixed number of attempts, it falls back onto the origin file server.

[2]Our implementation is structured using asynchronous events and callbacks within a single process, we use the term "thread" here only for clarity of explanation.

Reusing proxy connections. While a client is downloading a chunk from a proxy, it attempts to reuse the connection to the proxy by *negotiating* for other chunks. The client picks α random chunks still needed. It computes the corresponding α indexing keys and sends these to the proxy. The proxy responds with those γ chunks, among the α requested, that it already has. If $\gamma = 0$, the proxy responds instead with β keys corresponding to chunks that it does have. The client, upon downloading the current chunk, selects a new chunk from among those negotiated (*i.e.*, needed by the client and known by the proxy). The client then proves read permissions on the new chunk and begins fetching the new chunk. If no such chunks can be negotiated, the client terminates the connection.

Client-proxy interactions. We now describe the secure communication mechanisms between clients and proxies that ensure confidentiality and authorization. We already described how clients achieve data integrity by verifying the contents of files/chunks by their tokens.

To prevent adversaries from passively reading or actively modifying content while in transmission, the client and proxy first derive a symmetric encryption key K_E before transmitting a chunk. As the token T_{F_i} already serves as a shared secret for chunk F_i, the parties can simply use it to generate this key.

Figure 2.3 shows the protocol by which Shark clients establish a secure session. First, the parties exchange fresh, random 20-byte nonces r_C and r_P upon initiating a connection. For each chunk to be sent over the connection, the client must signal the proxy which token T_{F_i} to use, but it can do so without exposing information to eavesdroppers or malicious proxies by simply sending I_{F_i} in the clear. Using these nonces and knowledge of T_{F_i}, each party computes authentication tokens as

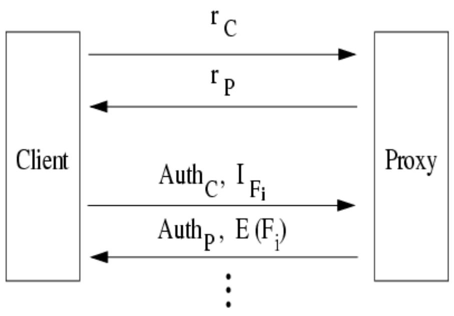


Figure 2.3: Shark session establishment protocol

follows:

$$\begin{aligned} Auth_C &= \mathsf{HMAC}_{T_{F_i}}(\mathsf{A}_C, C, P, r_C, r_P) \\ Auth_P &= \mathsf{HMAC}_{T_{F_i}}(\mathsf{A}_P, P, C, r_P, r_C) \end{aligned}$$

The $Auth_C$ token proves to the proxy that the client actually has the corresponding chunk token T_{F_i} and thus read permissions on the chunk. Upon verifying $Auth_C$, the proxy replies with $Auth_P$ and the chunk F_i after applying E (see below) to it.

In our current implementation, E is instantiated by a symmetric block encryption function, followed by a MAC covering the ciphertext. However, we note that $Auth_P$ already serves as a MAC for the *content*, and thus this additional MAC is not strictly needed. [3] The symmetric encryption key K_E for E is derived in a similar manner as before:

$$K_E = \mathsf{HMAC}_{T_{F_i}}(\mathsf{E}, C, P, r_C, r_P)$$

[3]The results of Krawczyk [43] speaking on the *generic* security concerns of "authenticate-and-encrypt" are not really relevant here, as we already expose the raw output of our MAC via I_{F_i} and thus implicitly assume that HMAC does not leak any information about its contents. Thus, the inclusion of $Auth_P$ does not introduce any *additional* data confidentiality concerns.

An additional MAC key can be similarly derived by replacing the special character E with M. Shark's use of fresh nonces ensure that these derived authentication tokens and keys cannot be replayed for subsequent requests.

Upon deriving this symmetric key K_E, the proxy encrypts the data within a chunk using 128-bit AES in counter mode (AES-CTR). For each 16-byte AES block, we use the block's offset within the chunk/file as its counter.

The proxy protocol has READ and READDIR RPCs similar to NFS, except they specify the indexing key I and $Auth_C$ to name a file (which is server independent), in place of a file handle. Thus, after establishing a connection, the client begins issuing read RPCs to the proxy; the client decrypts any data it receives in response using K_E and the proper counter (offset).

While this block encryption prevents a client without T_{F_i} from decrypting the data, one may be concerned if some unauthorized client can download a large number of encrypted blocks, with the hope of either learning K_E later or performing some offline attack. The proxy's explicit check of $Auth_C$ prevents this. Similarly, the verifiable $Auth_P$ prevents a malicious party that does not hold F_i from registering itself under the public I_{F_i} and then wasting the client's bandwidth by sending invalid blocks (that later will fail hash verification).

Thus, Shark provides strong data integrity guarantees to the client and authorization guarantees to the proxy, even in the face of malicious participants.

2.1.4 Exploiting file commonalities

We now describe the chunking method by which Shark can leverage file commonalities. This method (used by LBFS [53]) avoids dependence on file-length changes by setting chunk boundaries, or *breakpoints*, based on file contents, rather than on offset position. If breakpoints were selected only by offset—for instance, by breaking a file into aligned 16KB chunks—a single byte added to the front of a file would change all breakpoints and thus all chunk tokens.

To divide a file into chunks, we examine every overlapping 48-byte region, and if the low-order 14 bits of the region's Rabin fingerprint [62] equals some globally-chosen value, the region constitutes a breakpoint. Assuming random data, the expected chunk size is therefore $2^{14} = 16$KB. To prevent pathological cases (such as long strings of 0), the algorithm uses a minimum chunk size of 2KB and a maximum size of 64KB. Therefore, modifications within a chunk will minimize changes to the breakpoints: either only the chunk will change, one chunk will split into two, or two chunks will merge into one.

Note that although our implementation supports only Rabin fingerprints, it is only one technique to divide a file into chunks; the key idea is to obtain chunks which remain intact on simple modifications to a file. Thus, specialized techniques could be used for files which show definite patterns.

Content-based chunking enables Shark to exploit file commonalities: Even if proxies were reading different versions of the same file or different files altogether, a client can discover and download common data chunks, as long as they share the same chunk token (and no server-specific per-file randomness is used). As the fingerprint value is global, this chunking commonality also persists across multiple file systems.

2.1.5 Distributed indexing

Shark seeks to enable data sharing between files that contain identical data chunks, both on the same file system and across different file systems. This functionality is not supported by the simple server-based approach of indexing clients, whereby the file server stores and returns information on which clients are caching which chunks. Thus, we use a *global* distributed index for all Shark clients, even those accessing different Shark file systems.

Shark uses a structured routing overlay [67, 63, 65, 80, 49] to build its distributed index. The system maps opaque keys onto nodes by hashing their value onto a semantic-free identifier (ID) space; nodes are assigned identifiers in the same ID space. It allows scalable key lookup (in $O(\log(n))$ overlay hops for n-node systems), reorganizes itself upon network membership changes, and provides robust behavior against failure.

While some routing overlays optimize routes along the underlay, most are designed as part of distributed hash tables to store immutable data. In contrast, Shark stores only small references about which clients are caching what data: It seeks to allow clients to locate *copies* of data, not merely to find network efficient *routes* through the overlay. In order to achieve such functionality, Shark uses Coral [26] as its distributed index.

System overview. Coral exposes two main protocols: *put* and *get*. A Shark client executes the *get* protocol with its indexing key I as input; the protocol returns a list of proxy addresses that corresponds to some *subset* of the unexpired addresses *put* under I, taking locality into consideration. *put* takes as input I, a proxy's address, and some expiry time.

Coral provides a *distributed sloppy hash table* (DSHT) abstraction, which offers weaker consistency than traditional DHTs. It is designed for soft-state where multiple values may be stored under the same key. This consistency is well-suited

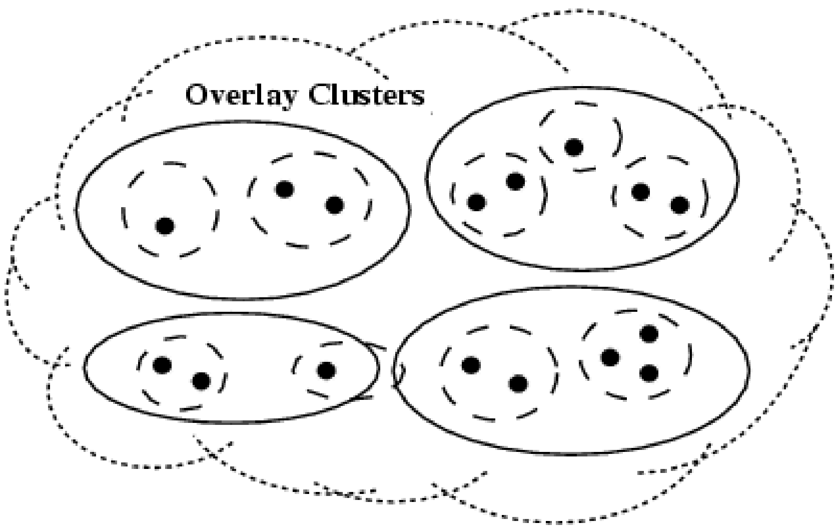


Figure 2.4: *Coral's three-level hierarchical overlay structure.* Nodes (solid circles) initially query others in their same high-level clusters (dashed rings), whose pointers reference other proxies caching the data within the same small-diameter cluster. If a node finds such a mapping to a replica proxy in the highest-level cluster, the *get* finishes. Otherwise, it continues among farther, lower-level nodes (solid rings), and finally, if need be, to any node within the system (the dotted cloud).

for Shark: A client need not discover *all* proxies for a particular file, it only needs to find *several*, *nearby* proxies.

Coral caches key/value pairs at nodes whose IDs are close (in terms of identifier space distance) to the key being referenced. To lookup the client addresses associated with a key I, a node simply traverses the ID space with RPCs and, as soon as it finds a remote peer storing I, it returns the corresponding list of values. To insert a key/value pair, Coral performs a two-phase operation. In the "forward" phase, Coral routes to nodes successively closer to I and stops when happening upon a node that is both *full* (meaning it has reached the maximum number of values for the key) and *loaded* (which occurs when there is heavy write traffic for a particular key). During the "reverse" phase, the client node attempts to insert the value at the closest node seen. Please see [26] for more details.

To improve locality, these routing operations are not initially performed across the entire global overlay: Each Coral node belongs to several distinct routing structures called *clusters*. Each cluster is characterized by a maximum desired network round-trip-time (RTT) called the *diameter*. The system is parameterized by a fixed hierarchy of diameters, or *levels*. Every node belongs to one cluster at each level, as shown in Figure 2.4. Coral queries nodes in fast clusters before those in slower clusters. This both reduces the latency of lookups and increases the chances of returning values stored on nearby nodes.

Handle concurrency via "atomic" put/get. Ideally, Shark clients should fetch each file chunk from a Shark server only once. However, a DHT-like interface which exposes two methods, *put* and *get*, is not sufficient to achieve this behavior. For example, if clients were to wait until completely fetching a file before referencing themselves, other clients simultaneously downloading the file will start transferring file contents from the server. Shark mitigates this problem by using Coral to request *chunks*, as opposed to whole files: A client delays its announcement for only the time needed to fetch a chunk.

Still, given that Shark is designed for environments that may experience abrupt flash crowds—such as when test-bed or grid researchers fire off experiments on hundreds of nodes almost simultaneously and reference large executables or data files when doing so—we investigated the practice of clients optimistically inserting a mapping to themselves upon initiating a request. A production use of Coral in a web-content distribution network takes a similar approach when fetching whole web objects [26].

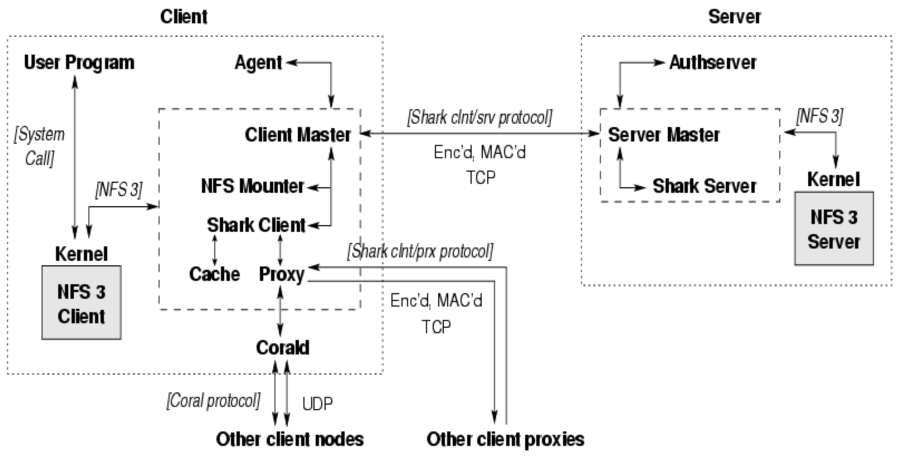


Figure 2.5: The Shark system components

Even using this approach, we found that an origin server can see redundant downloads of the same file when initial requests for a newly-popular file occur synchronously. We can imagine this condition occurring in Shark when users attempt to simultaneously install software on all test-bed hosts.

Such redundant fetches occur under the following race condition: Consider that a mapping for file/chunk F (and thus I_F) is not yet inserted into the system. Two nodes both execute $get(I_F)$, then perform a put. On the node closest to I_F, the operations serialize with both *get*s being handled (and thus returning no values) before either put.

Simply inverting the order of operations is even worse. If multiple nodes first perform a put, followed by a get, they can discover one another and effectively form cycles waiting for one another, with nobody actually fetching the file from the server.

To eliminate this condition, we extended store operations in Coral to provide return status information (like test-and-set in shared-memory systems). Specifically, we introduce a single put/get RPC which atomically performs both operations.

The RPC behaves similar to a *put* as described above, but also returns the first values discovered in *either* direction. (Values in the forward *put* direction help performance; values in the reverse direction prevent this race condition.)

While of ultimately limited use in Shark given small chunk sizes, this extension also proved beneficial for other applications seeking a distributed index abstraction [26].

2.2 Implementation

Shark consists of three main components, the server-side daemon `sharksd`, the client-side daemon `sharkcd` and the coral daemon `corald`, as shown in Figure 2.5. All three components are implemented in C++ and are built using the SFS toolkit [50]. The file-system daemons interoperate with the SFS framework, using its automounter, authentication daemon, etc. `corald` acts as a node within the Coral indexing overlay; a full description can be found in [26].

`sharksd`, the server-side daemon, is implemented as a loop-back client which communicates with the kernel NFS server. `sharksd` incorporates an extension of the NFSv3 protocol—the GETTOK RPC—to support file- and chunk-token retrieval. When `sharksd` receives a GETTOK call, it issues a series of READ calls to the kernel NFS server and computes the tokens and chunk breakpoints. It caches these tokens for future reference. `sharksd` required an additional 400 lines of code to the SFS read-write server.

`sharkcd`, the client-side daemon, forms the biggest component of Shark. In addition to handling user requests, it transparently incorporates whole-file caching and the client- and server-side functionality of the Shark cooperative cache. The code is 12,000 lines.

`sharkcd` comprises an NFS loop-back server which traps user requests and forwards them to either the origin file server or a Shark proxy. In particular, a read for a file block is intercepted by the loop-back server and translated into a series of READ calls to fetch the entire file in the background. The cache-management subsystem of `sharkcd` stores all files that are being fetched locally on disk. This cache provides a thin wrapper around file-system calls to enforce disk usage accounting. Currently, we use the LRU mechanism to evict files from the cache. The cache names are chosen carefully to fit in the kernel name cache.

The server side of the Shark cooperative cache implements the proxy, accepting connections from other clients. If this proxy cannot immediately satisfy a request, it registers a callback for the request, responding when the block has been fetched. The client side of the Shark cooperative cache implements the various fetching mechanism discussed in Section 2.1.3. For every file to be fetched, the client maintains a vector of objects representing connections to different proxies. Each object is responsible for fetching a sequence of chunks from the proxy (or a range of blocks when chunking is not being performed and nodes query only by file token).

An early version of `sharkcd` also supported the use of **xfs**, a device driver bundled with the ARLA [74] implementation of AFS, instead of NFS. However, given that the PlanetLab environment, on which we performed our testing, does not support **xfs**, we do not present those results in this paper.

During Shark's implementation, we discovered and fixed several bugs in both the OpenBSD NFS server and the **xfs** implementation.

2.3 Evaluation

This section evaluates Shark against NFSv3 and SFS to quantify the benefits of its cooperative-caching design for read-heavy workloads. To measure the performance of Shark against these file systems, without the gain from cooperative caching, we first present microbenchmarks for various types of file-system access tests, both in the local-area and across the wide-area. We also evaluate the efficacy of Shark's chunking mechanism in reducing redundant transfers.

Second, we measure Shark's cooperative caching mechanism by performing read tests both within the controlled Emulab LAN environment [75] and in the wide-area on the PlanetLab v3.0 test-bed [60]. In all experiments, we start with cold file caches on all clients, but first warm the server's chunk token cache. The server required 0.9 seconds to compute chunks for a 10 MB random file, and 3.6 seconds for a 40 MB random file.

We chose to evaluate Shark on Emulab, in addition to wide-area tests on PlanetLab, in order to test Shark in a more controlled, native environment: While Emulab allows one to exclusively reserve machines, individual PlanetLab hosts may be executing tens or hundreds of experiments (slices) simultaneously. In addition, most PlanetLab hosts implement bandwidth caps of 10 Mb/sec across all slices. For example, on a local PlanetLab machine operating at NYU, a Shark client took approximately 65 seconds to read a 40 MB file from the local (non-PlanetLab) Shark file server, while a non-PlanetLab client on the same network took 19.3 seconds. Furthermore, deployments of Shark on large LAN clusters (for example, as part of grid computing environments) may experience similar results to those we report.

The server in all the microbenchmarks and the PlanetLab experiments is a

1.40 GHz Athlon at NYU, running OpenBSD 3.6 with 512 MB of memory. It runs the corresponding server daemons for SFS and Shark. All microbenchmark and PlanetLab clients used in the experiments ran Fedora Core 2 Linux. The server used for Emulab tests was a host in the Emulab test-bed; it did not simultaneously run a client. All Emulab hosts ran Red Hat Linux 9.0.

The Shark client and server daemons interact with the respective kernel NFS modules using the *loopback* interface. On the Red Hat 9 and Fedora Core 2 machines, where we did our testing, the loopback interface has a maximum MTU of 16436 bytes and any transfer of blocks of size >= 16 KB results in IP fragmentation which appears to trigger a bug in the kernel NFS code. Since we could not increase the MTU size of the loopback interface, we limited both Shark and SFS to use 8 KB blocks. NFS, on the other hand, issued UDP read requests for blocks of 32 KB over the *ethernet* interface without any problems. These settings could have affected our measurements.

2.3.1 Alternate cooperative protocols

This section considers several alternative cooperative-caching strategies for Shark in order to characterize the benefits of various design decisions.

First, we examine whether clients should issue requests for chunks sequentially (*seq*), as opposed to choosing a *random* (previously unread) chunk to fetch. There are two additional strategies to consider when performing sequential requests: Either the client immediately *pre*-announces itself for a particular chunk upon requesting it (with an “atomic” *put*/*get* as in Section 2.1.5), or the client waits until it finishes fetching a chunk before announcing itself (via a *put*). We consider such sequential strategies to examine the effect of disk scheduling latency: for single clients in the local area, we expect the random strategy to limit the throughput to that imposed by the file server's disk seek time. But when multiple clients operate concurrently, we expect that the random strategy allows all clients to fetch independent chunks from the server and later trade these chunks among themselves. Using a purely sequential strategy, the clients all advance only as fast as the few clients that initially fetch chunks from the server.

Second, we disable the *negotiation* process by which clients may reuse connections with proxies and thus download multiple chunks once connected. In this case, the client must query the distributed index for each chunk.

2.3.2 Microbenchmarks

For the local-area microbenchmarks, we used a local machine at NYU as a Shark client. Maximum TCP throughput between the local client and server, as measured by `ttcp`, was 11.14 MB/sec. For wide-area microbenchmarks, we used a client machine located at the University of Texas at El Paso. The average round-trip-time (RTT) between this host and the server, as measured by `ping`, is 67 ms. Maximum TCP throughput was 1.07 MB/sec.

Access latency. We measure the time necessary to perform four types of file-system accesses: (1) to read 10 MB and (2) 40 MB large random files on remote hosts, and (3) to read large numbers of small files. The small file test attempts to read 1,000 1 KB files evenly distributed over ten directories.

We performed single-client microbenchmarks to measure the performance of Shark. Figure 2.6 shows the performance on the local- and wide-area networks for these three experiments, We compare SFS, NFS, and three Shark configurations, *viz.* Shark without calls to its distributed indexing layer (*nocoral*), fetching chunks

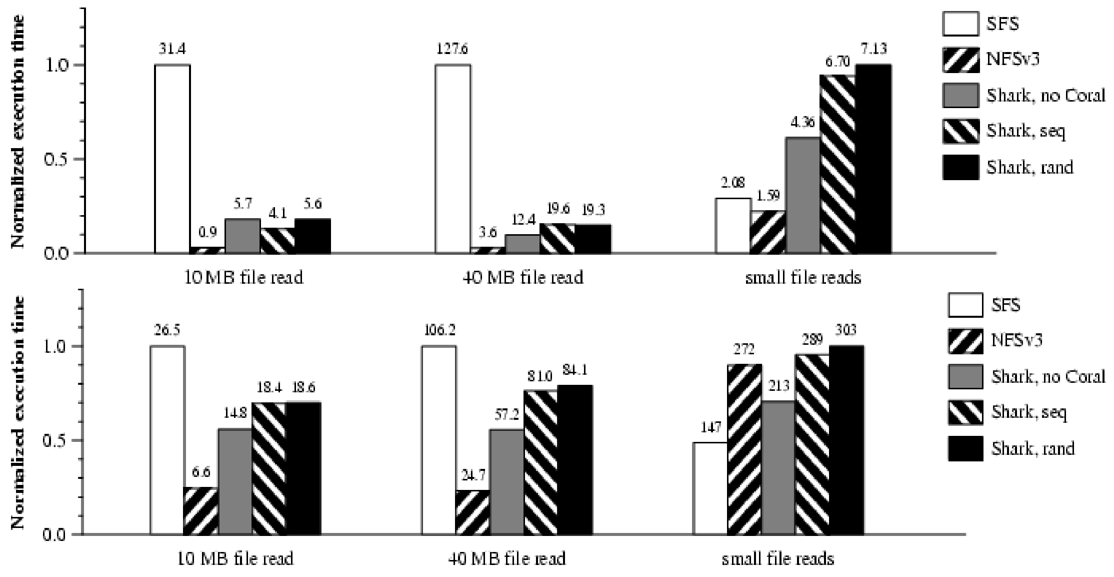


Figure 2.6: *Local-area* (top) *and wide-area* (bottom) *microbenchmarks.* Normalized application performance for various types of file-system access. Execution times in seconds appear above the bars.

from a file sequentially (*seq*), and fetching chunks in random order (*rand*). Shark issues up to eight outstanding RPCs (for *seq* and *rand*, fetching four chunks simultaneously with two outstanding RPCs per chunk). SFS sends RPCs as requested by the NFS client in the kernel.

For all experiments, we report the normalized *median* value over three runs. We interleaved the execution of each of the five file systems over each run. We see that Shark is competitive across different file system access patterns and is optimized for large read operations.

Chunking. In this microbenchmark, we validate that Shark's chunking mechanism reduces redundant data transfers by exploiting data commonalities.

We first read the `tar` file of the entire source tree for emacs v20.6 over a Shark file system, and then read the `tar` file of the entire source tree for emacs v20.7. We note that of the 2,083 files or directories that comprise these two file archives, 1,425 have not changed between versions (*i.e.*, they have the identical `md5` sum), while 658 of these have changed.

Figure 2.7 shows the amount of bandwidth savings that the chunking mechanism provides when reading the newer emacs version. When `emacs-20.6.tar` has been cached, Shark only transfers 33.8 MB (1416 chunks) when reading `emacs-20.7.tar` (of size 56.3 MB).

2.3.3 Local-area cooperative caching

Shark's main claim is that it improves a file server's scalability, while retaining its traditional benefits. We now study the end-to-end performance of reads in a cooperative environment with many clients attempting to simultaneously read the same file(s).

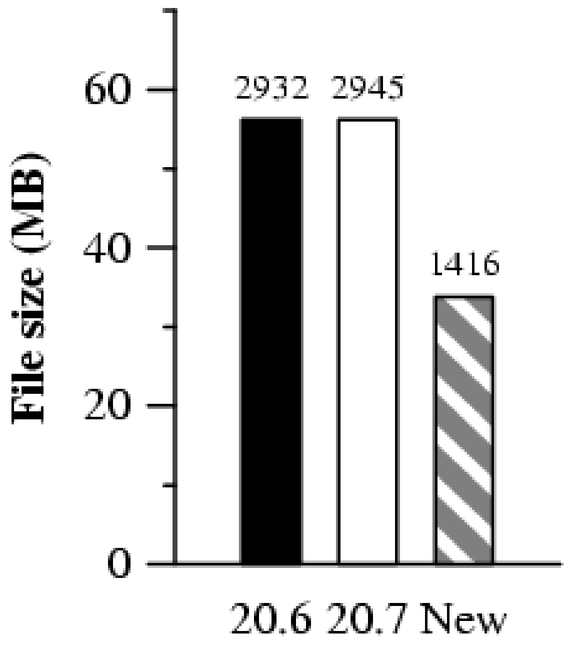
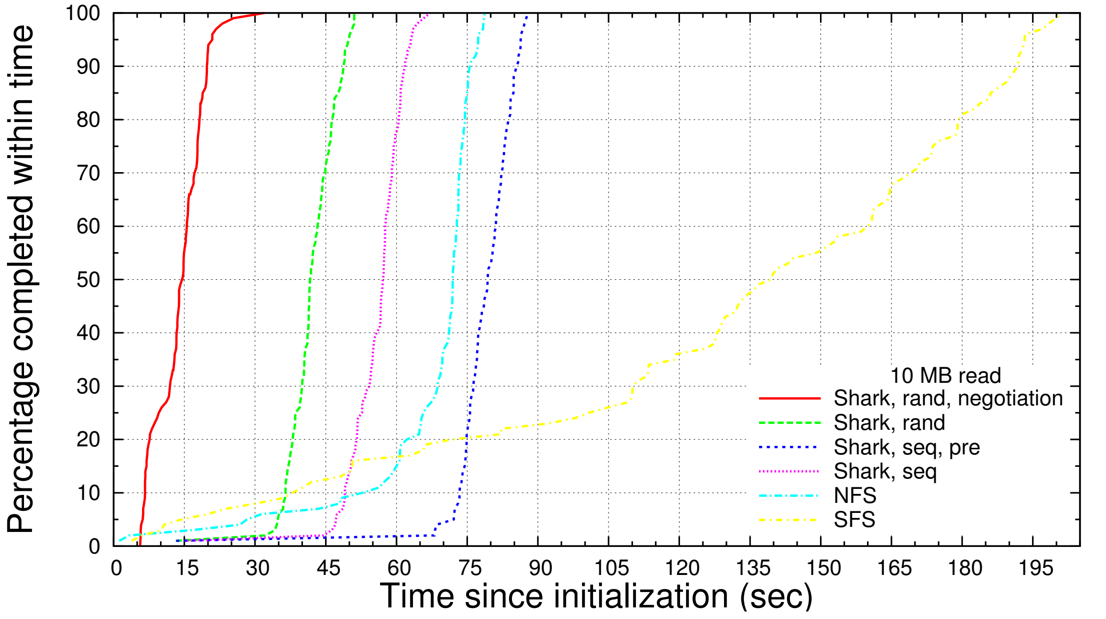


Figure 2.7: *Bandwidth savings from chunking.* "New" reflects the number of megabytes that need to be transferred when reading emacs 20.7, given a cached copy of emacs 20.6. Number of chunks comprising each transfer appears above the bars.

In this section, we evaluate Shark on Emulab [75]. These experiments allowed us to evaluate various cooperative strategies in a better controlled environment. In all the configurations of Shark, clients attempt to download a file from four other proxies simultaneously.

Figure 2.8 shows the cumulative distribution functions (CDFs) of the time needed to read a 10 MB and 40 MB (random) file across 100 physical Emulab hosts, comparing various cooperative read strategies of Shark, against vanilla SFS and NFS. In each experiment, all hosts mounted the server and began fetching the file simultaneously. We see that Shark achieves a median completion time $< \frac{1}{4}$



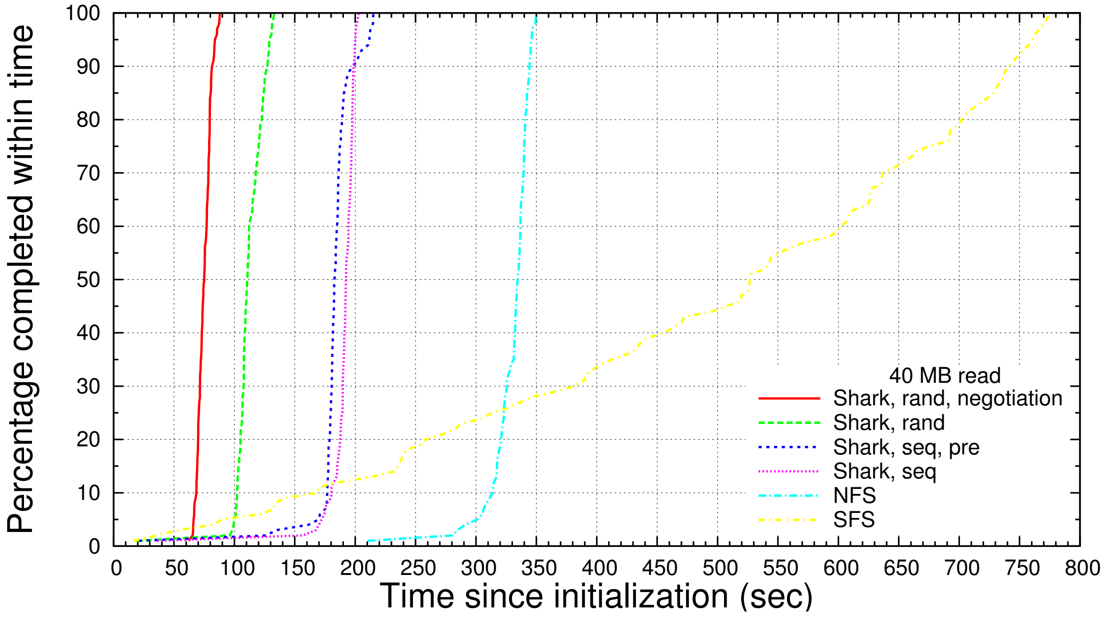


Figure 2.8: *Client latency.* Time (in seconds) required for 100 LAN hosts to read a 10 MB (top) and 40 MB (bottom) file.

that of NFS and $< \frac{1}{6}$ that of SFS. Furthermore, its 95th percentile is almost an order of magnitude better than SFS.

Shark's fast, almost vertical rise (for nearly all strategies) demonstrates its cooperative cut-through routing: Shark clients effectively organize themselves into a distribution mesh. Considering a single data chunk, a client is part of a chain of nodes performing cut-through routing, rooted at the origin server. Because clients may act as root nodes for some chunks and act as leaves for others, most finish at almost synchronized times. The lack of any degradation of performance in the upper percentiles demonstrates the lack of any heterogeneity, both in terms of network bandwidth and underlying disk/CPU load, amongst the Emulab hosts.

Interestingly, we see that most NFS clients finish at loosely synchronized times, while the CDF of SFS clients' times has a much more gradual slope, even though both systems send all read requests to the file server. Subsequent analysis of NFS over TCP (instead of NFS over UDP as shown) showed a similar slope as SFS, as did Shark without its cooperative cache. One possible explanation is that the heavy load on (and hence congestion at) the file server imposed by these non-cooperative file systems drives some TCP connections into back-off, greatly reducing fairness.

We find that a *random* request strategy, coupled with inter-proxy *negotiation*, distinctly outperforms all other evaluated strategies. A sequential strategy effectively saw the clients furthest along in reading a file fetch the leading (four) chunks from the origin file server; other clients used these leading clients as proxies. Thus, modulo possible inter-proxy timeouts and synchronous requests in the non-*pre*-announce example, the origin server pushed at most four new chunks concurrently. Using a *random* strategy, more distinct chunks are fetched from the server simultaneously and thus propagate more quickly through the clients' dissemination mesh.

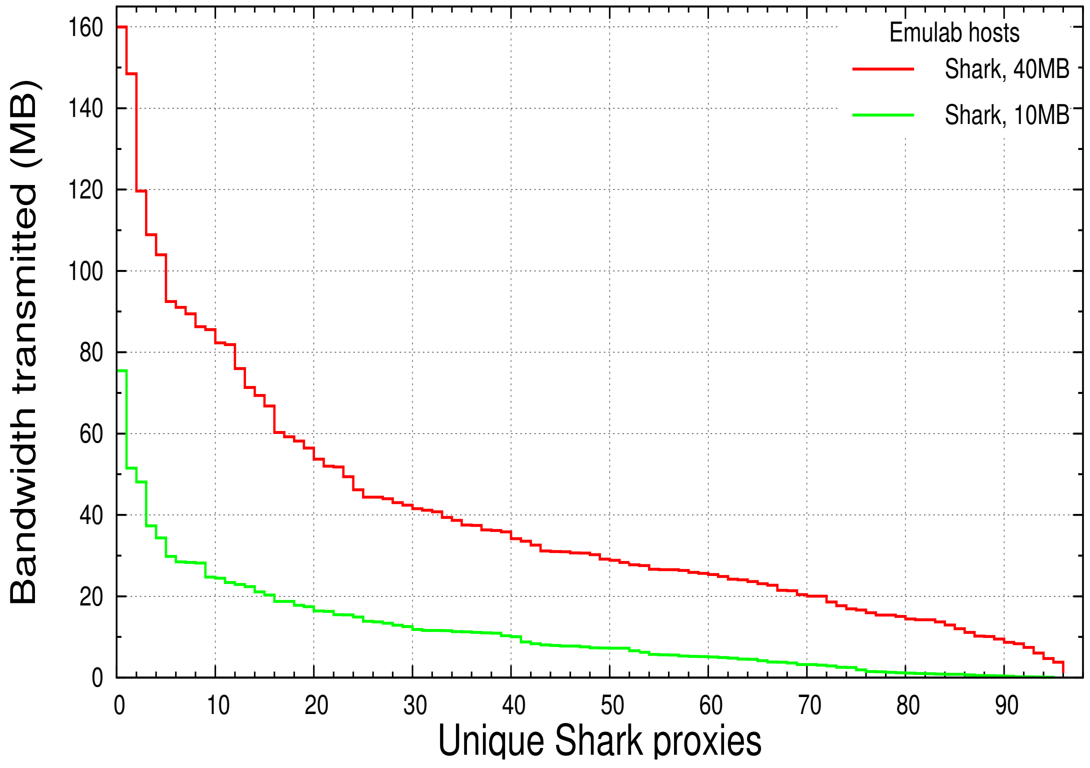


Figure 2.9: *Proxy bandwidth usage.* Amount of data in MB served by each Emulab proxy when reading 40 MB and 10 MB files.

Figure 2.9 shows the total amount of bandwidth served by each proxy as part of Shark's cooperative caching, when using a random fetch strategy with inter-proxy negotiation for the 40 MB and 10 MB experiments. We see that the proxy serving the most bandwidth contributed four and seven times more upstream bandwidth than downstream bandwidth, respectively. During these experiments, the Shark file server served a total of 92.55 MB and 15.48 MB, respectively. Thus, we conclude that Shark is able to significantly reduce a file server's bandwidth utilization, even when distributing files to large numbers of clients. Furthermore, Shark ensures that any one cooperative-caching client does not incur excessive bandwidth cost.

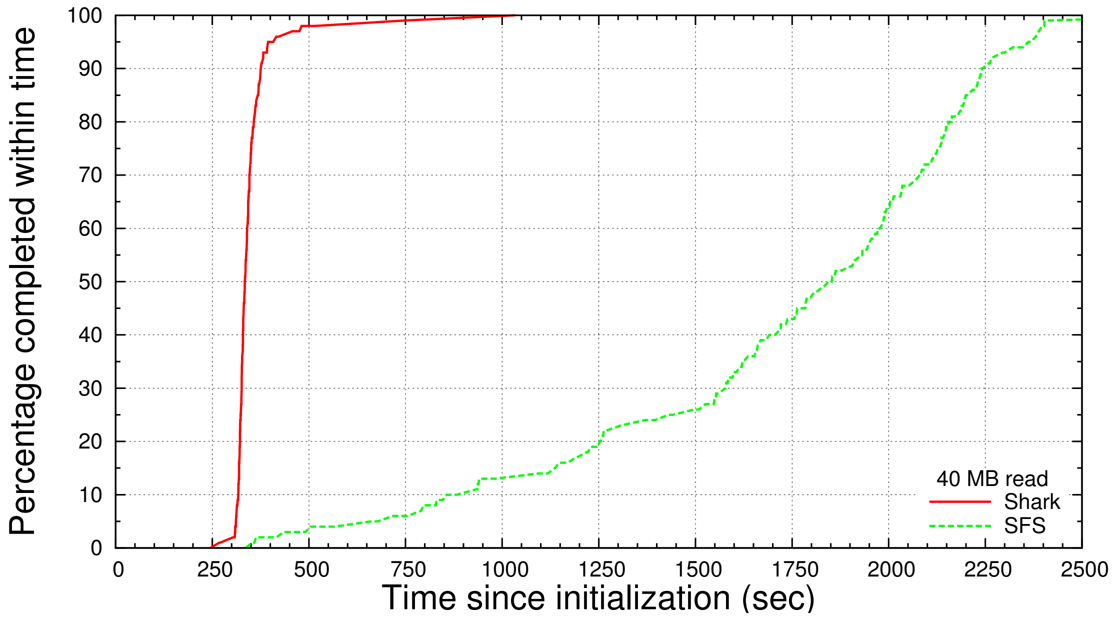


Figure 2.10: *Client latency.* Time (in seconds) for 185 hosts to finish reading a 40 MB file using Shark and SFS.

2.3.4 Wide-area cooperative caching

Shark seeks to improve a file server's scalability while retaining its traditional usage. In our cooperative caching experiment, we study the end-to-end performance of attempting to perform reads within a large, wide-area distributed test-bed.

On approximately 185 PlanetLab hosts, well-distributed across North America, Europe, and Asia, we attempted to simultaneously read a 40 MB random file. All hosts mounted the server and began fetching the file simultaneously.

Figure 2.10 shows a CDF of the time needed to read the file on all hosts, comparing Shark with SFS.

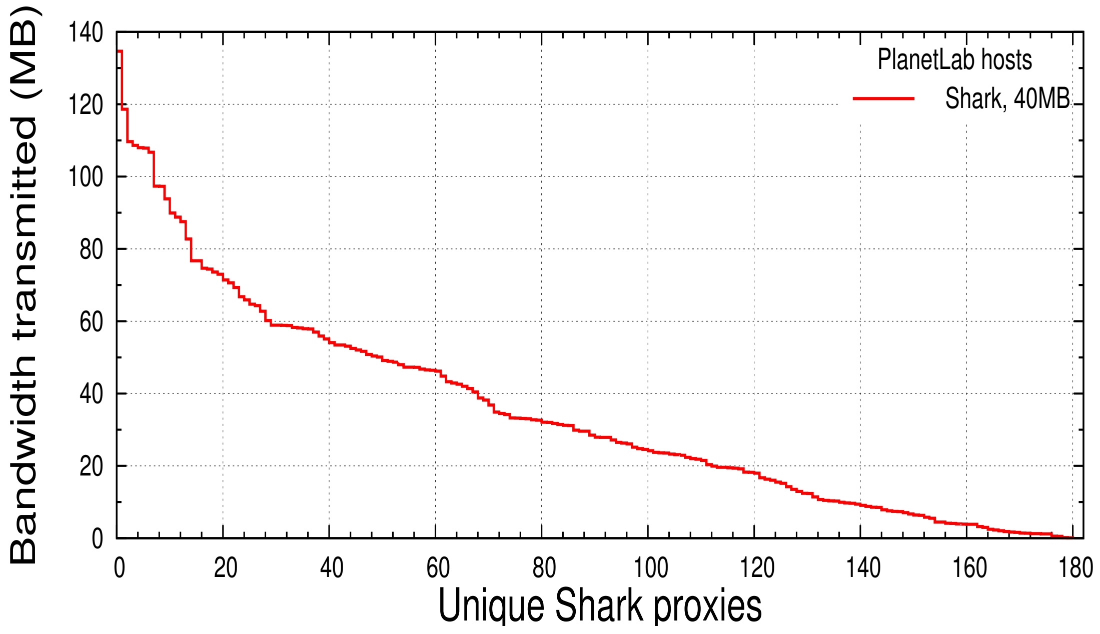


Figure 2.11: *Proxy bandwidth usage.* Amount of data in MB served by each PlanetLab proxy when reading 40 MB files.

% done in (sec)	50%	75%	90%	95%	98%
Shark	334	350	375	394	481
SFS	1848	2129	2241	2364	2396

We see that, between the 50th and 98th percentiles, Shark is five to six times faster than SFS. The graph's sharp rise and distinct knee demonstrates Shark's cooperative caching: 96% of the nodes effectively finish at nearly the same time. Clients in SFS, on the other hand, complete at a much slower rate.

Wide-area experiments with NFS repeatedly crashed our file server (*i.e.*, it caused a kernel panic). We were, therefore, unable to evaluate NFS in the wide area.

Figure 2.11 shows the total amount of bandwidth served by each proxy during

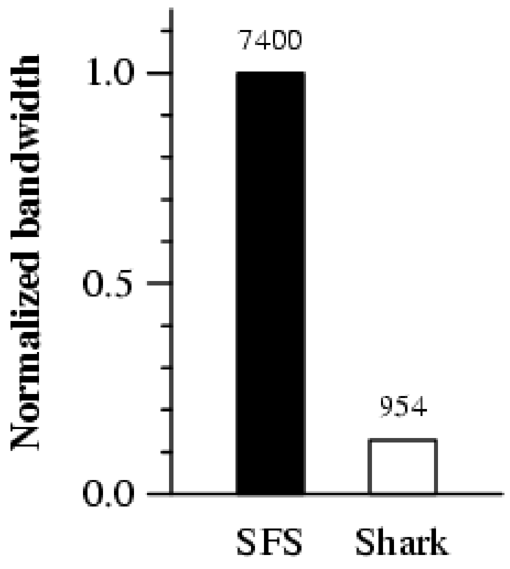


Figure 2.12: *Server bandwidth usage.* Amount of data in MB read from server as a 40 MB file is fetched by 185 hosts.

this experiment. We see that the proxy serving the most bandwidth contributed roughly three times more upstream than downstream bandwidth.

Figure 2.12 shows the number of bytes read from our file server during the execution of these two experiments. We see that Shark reduces the server's bandwidth usage by an order of magnitude. Thus, Sharkmeets the goal of improving the scalability of a traditional file server.

Chapter 3

RedCarpet[*]

In this chapter, we describe the problem of providing a near-Video-on-Demand (nVoD) service to a large population of users using a server with limited resources, and define the metrics of interest in such a system. We then describe the simulator we have built to evaluate different points in the design space. We initially consider naive policies for scheduling the dissemination of data blocks in the system, point out their shortcomings, and address them to arrive at policies which work well in a variety of scenarios. We then describe our prototype implementation of the system, and show that the results from our simulator tally with those from the implementation. We then present an algorithm that introduces structure into the placement of nodes in the system, which considerably improves performance. Finally, we show that our techniques work well even in the presence of heterogeneous nodes (nodes with differing bandwidth capacities).

[*]This work was started while interning at Microsoft Research Cambridge (MSRC), during the summer of 2005.

3.1 Problem Setting

We have a large number of users interested in some video content, which initially exists on a special peer that we call a *server*. The users could arrive at any point in time and watch the video from the beginning (fast-forward functionality is beyond the scope of this thesis, though we believe that our techniques apply to this case as well). In other words, we assume *linear viewing*, but we allow the users to join at arbitrary times. The resources (especially network bandwidth) of the server are limited, and the users contribute their own resources to the system.

The goal of our system is to ensure a *low setup time* (or initial buffering time before playback starts), and a *sustainable data download rate* (which is higher than the video encoding rate) for all users, regardless of their arrival time. For each user, we plot the number of consecutive video blocks from the beginning that the user has downloaded as a function of time (see Fig. 3.1). These blocks can be played without interruption in playback. For a given setup time (i.e. amount of initial buffering), we calculate the sustainable data download rate as the maximum slope of a line that does not exceed the y-coordinate (number of contiguous blocks) at any time. We call that rate the *goodput*. We typically use the 95^{th} percentile goodput over all nodes as a measure of system performance, as this number indicates the data download rate that 95% of the nodes can support; when appropriate, we also report the minimum and maximum values.

The users organize themselves into an unstructured overlay mesh which resembles a random graph (as mentioned earlier), and download blocks of video content from each other. A client joins the system by contacting a central tracker (whose address is obtained by an independent bootstrap mechanism). This tracker gives the client a subset of nodes already present in the system. The client then contacts

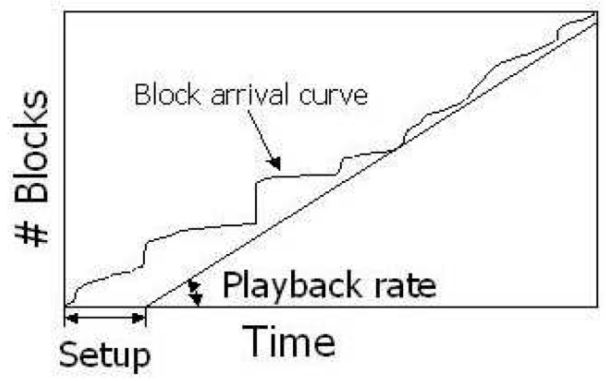


Figure 3.1: This example graph shows the calculation of sustainable playback rate, given the setup time. The y-axis shows the number of *consecutive* blocks, while the x-axis shows the time.

each of these nodes, and incorporates itself into the overlay mesh. Thus, each node is oblivious of the other nodes in the system except for this small subset, which we designate as its *neighborhood*. Each node can exchange content, as well as control messages, only with its immediate neighbors (or with the tracker/server).

Another metric of interest in such a mesh network with data exchange is the total number of blocks exchanged amongst all the nodes in the system per unit of time, which we call *throughput*. This is a measure of the utilization of system resources. Similarly, we also define the node throughput as the amount of data downloaded by a node in a unit of time. We note that not all block exchanges increase the goodput of the nodes, as some of the blocks might be useful only for future playback. Hence, our objective is to **maximize** *throughput* for high system efficiency, while providing high *goodput* to ensure sustainable playback for all nodes.

When a node loses a neighbor (for example, when a neighbor crashes) or wishes to increase its download rate, it can request additional neighbors from the tracker. Note that we assume fail-stop behaviour from the clients, i.e., they either function correctly or they cease to be a part of the system. In particular, they are not actively malicious; malicious clients are outside the scope of this thesis.

We assume that the clients themselves are resource-constrained (esp. network bandwidth). Thus, the download and the upload rates of a client are limited by its capacity. We consider scenarios where clients have asymmetric links, i.e., their upload and download capacities are different.

The file is divided into a number of *blocks*. A number of consecutive blocks could be grouped into a *segment* to improve efficiency. The system is agnostic to the media codec, and hence we do not rely on a knowledge of the media format to recover from errors and packet losses; as a result, an important constraint on our system is to ensure that all blocks are received by the clients without any errors. Note that blocks can only be played if they are received within their playback deadline. We assume that the users have enough storage capacity to keep a copy of all the blocks downloaded up to that point in time. We feel that this is a pragmatic assumption, given that disk space has become very inexpensive recently, as shown in Figure 3.2 ([68]). Note that blocks are available for other peers to download only while the user is watching the video content or shortly after he/she is done.

In our system, while we consider various arrival patterns, most of our motivating examples focus on flash-crowd scenarios where most users arrive closely in time after the video is published, and the file initially resides only at the server with limited upload capacity. We note that typically, in steady-state, it is possible to find a few nodes that have already downloaded the content (either entirely or in part) and can act as servers; the resulting increase in serving capacity thus eases the problem of content scheduling. Hence, we chose to focus on the flash-crowd

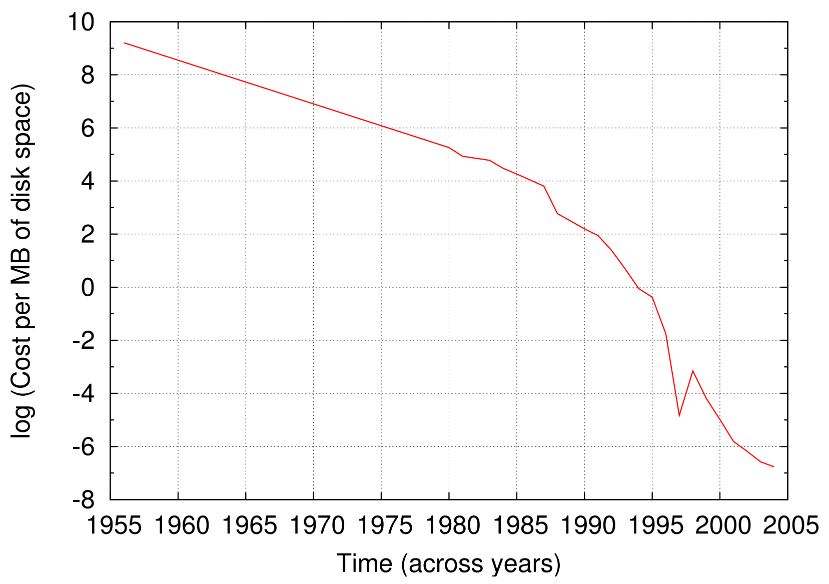


Figure 3.2: This graph shows the price of an MB of hard disk space in log scale on the y-axis, across the years shown on the x-axis.

arrival pattern, because this configuration exercises the system the most.

3.2 Simulator

We have done extensive simulations, and measurements using a prototype, to understand the various factors that affect the performance of VoD over P2P networks, and also to evaluate the performance of our algorithms. The simulator models important elements of P2P networks like access capacities, supports different block scheduling algorithms, and allows us to experiment with large networks; we describe it in detail below (the prototype will be described later).

The simulator takes as input the size of the video file in units of *blocks* (typically 250 in our simulations), the number of nodes (typically 500), their capacities, and the times at which nodes join/depart the system. The simulator operates in discrete intervals of time called *rounds*. A client's upload/download capacity is measured as the number of blocks that the client can transmit/receive in one round (typically 1 block/round). We note that with this setting, the maximum goodput that can be achieved at a node is 1 block/round (i.e., the download capacity). Each node connects to a small number of neighbors (typically 3-6). The topology changes during the simulation as a result of node arrivals and departures, and as the nodes try to find new neighbors to increase their download rates.

At every round, each node contacts its neighbors to identify those that have useful blocks. Thus, there is a random matching of peers that can exchange content. All block transfers, both between the peers and from the server, happen simultaneously, and the system then, as a whole, moves to the next round.

We now illustrate, with an example, what the above units (i.e., blocks, rounds etc.) mean in a realistic setting. There are two independent parameters used in the simulator – the unit of time (round) and the unit of bandwidth (blocks/round). A round is equivalent to 10 s, and a block/round to 512 kbps in realistic terms. Thus, if a strategy were to deliver a 95th percentile goodput of 0.6 blocks/round with a setup time of 35 rounds, it implies that a node could wait for about 6min (35 rounds $*$ 10 $=$ 350 s), and watch a video whose size is about 70 min ((250 blocks) / (0.6 blocks/round) $\approx$ 416 rounds $\approx$ 70 min), and is encoded at a rate of about 300 kbps (0.6 blocks/round $*$ 512 kbps $=$ 307.2 kbps) without interruption.

Note that while our simulator does not model realistic P2P networks in all their details (e.g. network delays, locality properties etc.), it does capture some of the important properties of mesh-based P2P networks. Hence, we feel that many of our results are applicable to the design of real mesh-based systems.

3.3 Naive approaches

In this section, we present some naïve algorithms for scheduling the dissemination of blocks in a simulated network of 500 nodes all arriving at the same time (flash crowd scenario). We will show that the naïve approaches perform poorly. We then investigate the factors responsible for their poor performance, and address the issues on the way to developing new algorithms that perform significantly better.

Our first algorithm is based on current swarming systems that distribute the blocks of the file in *random* order. This strategy results in high block diversity amongst the nodes in the system, resulting in good system throughput. However, since the nodes receive the blocks in random order, they may not be useful for sustaining a high goodput. In Fig. 3.3 we plot the 95% percentile goodput as a function of the setup time. The goodput is given as a fraction of the access link capacity, which is a natural upper bound on the maximum sustainable goodput. Indeed, the goodput is 0 blocks/round at a setup time of 35 rounds, even though the system throughput is quite high with an average of 339.91 block exchanges per round (out of a maximum of 500 blocks). Thus, despite the high throughput, the *random* strategy results in a low goodput for the nodes.

Another naïve approach would be for a node to download the blocks in the order required for playback, i.e., to use a *sequential* strategy. Indeed, Fig. 3.3 shows that this policy performs better than the *random* strategy, and is able to deliver a sustained goodput of ≈ 0.21 blocks/round at a setup time of 35 rounds. With this strategy, though, the peers all have very similar blocks (as they are all interested in downloading the same blocks), and hence there are fewer chances to find and exchange *innovative* blocks. Indeed, the throughput of the system is very low at an average of 95.93 block exchanges per round, which in turn caps the

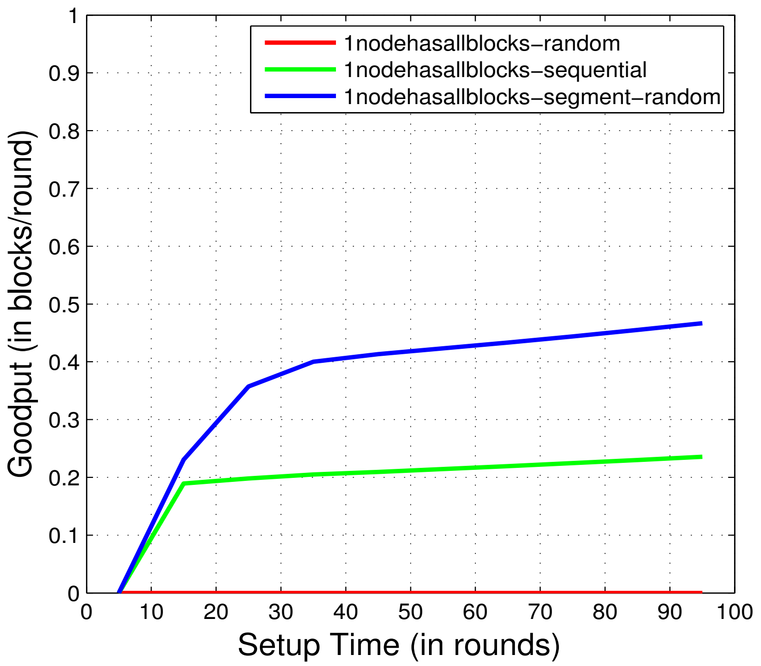
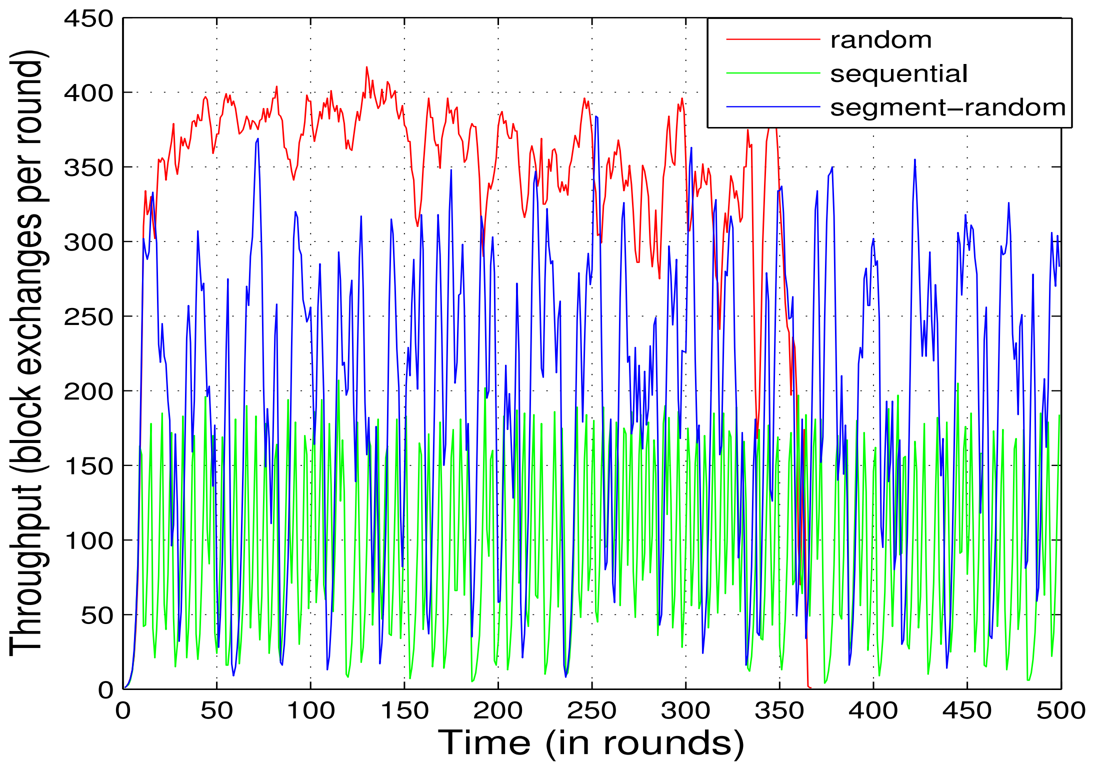


Figure 3.3: Comparison of random, sequential, and segment-random policies

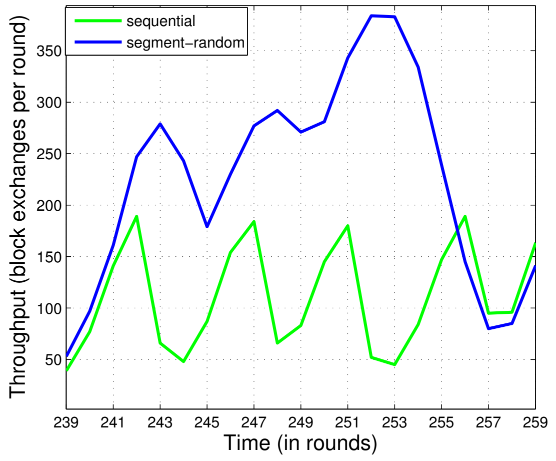
goodput that can be achieved.

The *segment-random* policy attempts to combine the high throughput of the *random* strategy with the high goodput of the *sequential* strategy. This technique divides the file into *segments* which are groups of consecutive blocks; for example a file of 250 blocks is divided into 25 segments of 10 blocks each. The peers request blocks within a segment in random order, but request the segments themselves in sequential order. Fig. 3.3 shows that the *segment-random* strategy achieves a goodput of 0.40 blocks/round at a setup time of 35 rounds, and has a reasonable throughput of 195.47 block exchanges per round; thus, the performance is better than with the above naïve algorithms.

We now look into the throughput of the system with the above policies, to



(a)



(b)

Figure 3.4: Figure 3.4(a) shows system throughput over time, with the naïve policies. Figure 3.4(b) is a blown-up graph showing system throughput over a period of 20 rounds, with the sequential and segment-random policies.

investigate the bottlenecks. The results are shown in Figure 3.4(a). As expected, the random policy takes a small time to ramp up initially, and consistently maintains a high throughput thereafter, which diminishes only towards the end of the dissemination. This high throughput is, as pointed above, due to the diversity of the cooperative cache formed by the nodes. With the sequential policy, we observe a distinct "peaks and troughs" pattern spread over ≈ 5 rounds in the throughput curve. We feel that the rise/fall in throughput corresponds to the availability/lack of new blocks in the system (It is an interesting sidenote that the average throughput with the sequential policy is only 95.93 blocks per round with a mesh structure, as compared to ≈ 250 blocks per round that we would expect with a binary tree.)

With the segment-random policy, the "peaks and troughs" pattern is again obvious, though spread over ≈ 20 rounds. We feel that the rise in this case corresponds to the availability of new segments, while the fall in throughput is due to the lack of new segments as well as the rarity of certain blocks in existing segments. This observation suggests two techniques to increase the system throughput beyond that obtained with the segment-random policy, viz. to avoid the formation of rare blocks in existing segments (in the system) and to increase the availability of new segments.

To avoid rare blocks in existing segments, we use a segment-rarest policy [20] at the clients. With this policy, the client identifies a segment it is interested in, and downloads a block within this segment that is least *popular* in its neighbourhood. Popularity of a block in a neighbourhood is the number of nodes that have a copy of that block available. With this policy in place, the throughput of the system increased to 208.61 block exchanges per round, a 6.72% improvement.

To increase the availability of new segments in the system, we introduce the technique of *pre-fetching* where we probabilistically fetch a block from a segment that is required later than the segment which is currently of interest. The idea with pre-fetching is that nodes download blocks from the future segments with a small probability; even though these block downloads are not immediately useful, and could be considered as "wasted" downloads from a client's standpoint, they still hold an overall benefit for the system. The reason is that nodes doing pre-fetching for future segments, act as launching pads for the content that they pre-fetch. In essence, pre-fetching provides easy access to blocks when they are actually needed by creating additional sources ahead of time, thus minimizing the latency incurred for block propagation. The end result is a smoother transition across segments.

We have considered a number of policies some of which pre-fetch from all the required segments, some of which only consider a few segments into the future, and policies with different probabilities of pre-fetching. The policy which performed consistently well across various scenarios is the following: each peer considers the first two segments of blocks that it needs; the peer then chooses between the segments using a biased coin, typically it picks the first segment with 80% probability and the second segment with 20% probability (80-20 policy). Within each segment, it downloads a particular block using one of the block policies described earlier; typically we use the rarest policy. The throughput of the system under this policy is 212.21 block exchanges per round, an improvement of 8.56%. The impact of the segment-rarest policy and pre-fetching on the goodput of the nodes is shown in Figure 3.5.

In Figures 3.6, 3.7, we highlight the benefits of the segment-rarest policy and pre-fetching vis-a-vis the segment-random policy. We observe that the peaks are higher with the segment-rarest policy because of the avoidance of rare blocks, enabling more block exchanges. Also, the depth of the valleys is considerably decreased with pre-fetching because of smoother segment transitions. The trade-

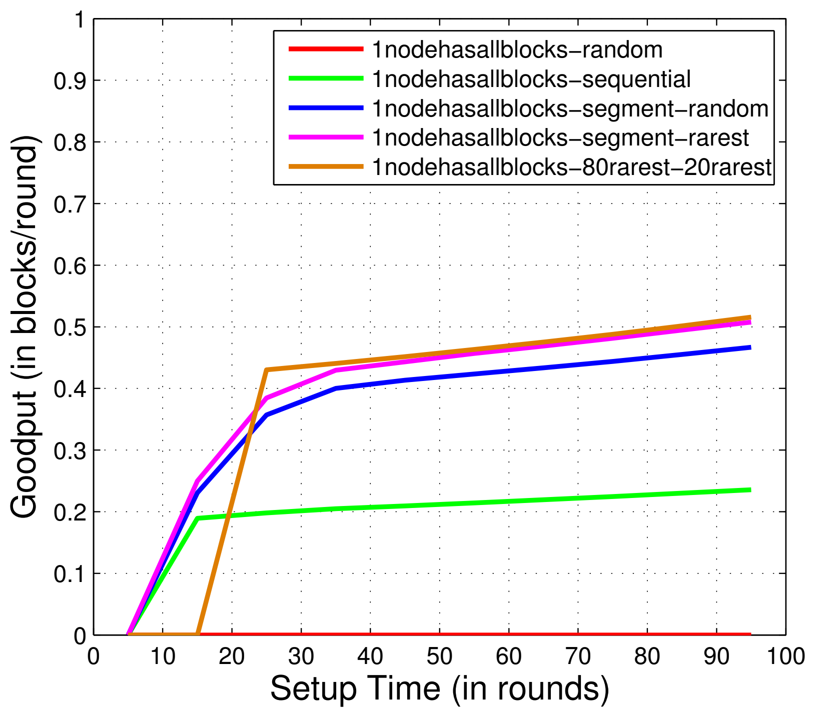


Figure 3.5: The above graph demonstrates the benefits of pre-fetching.

off with pre-fetching is that while it hurts the propagation of the current segment at a few nodes, it also improves the height of the valleys, and we note that the advantages far outweigh the disadvantages.

In addition to the approaches described above, we experimented with a number of block scheduling policies (some leveraging even global knowledge of the system) to discover good heuristics. A consistent observation was that greedy policies performed quite badly, while algorithms which required the peers to download blocks that are not of immediate interest to them but are from segments under-represented in the network, improved the overall performance of the system. We will leverage this observation for *segment scheduling* which is described later (Sec 3.6).

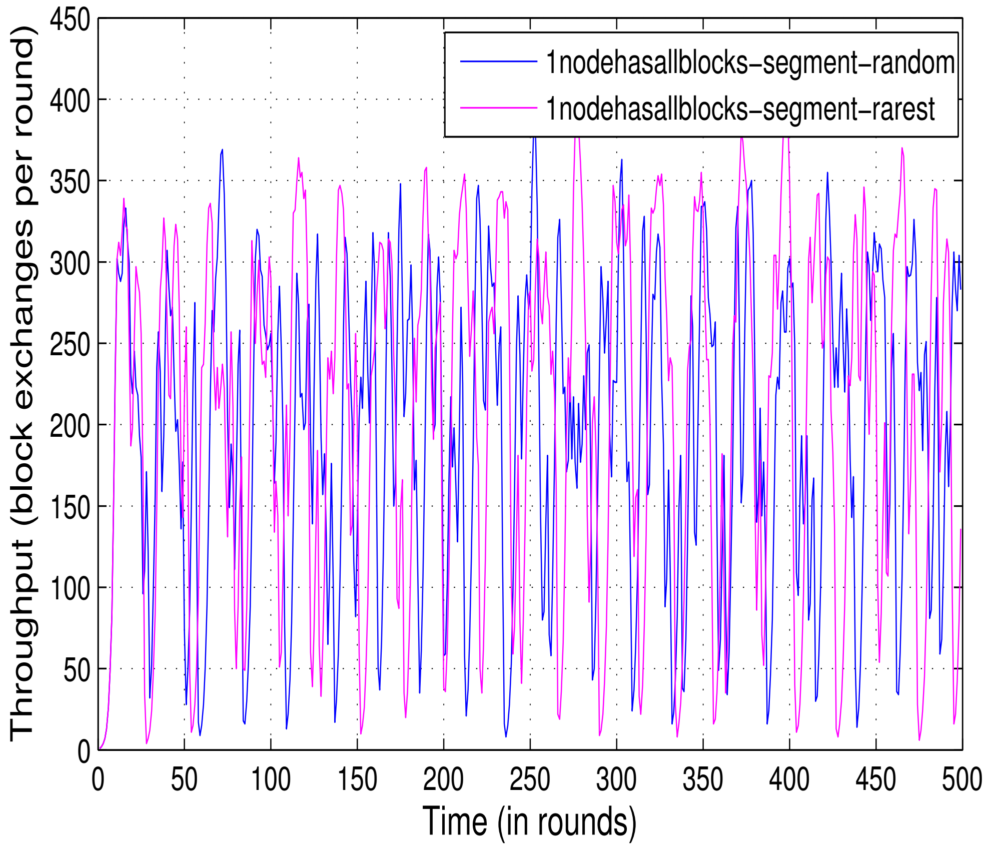


Figure 3.6: This graph shows the system throughput over time with the segment-random and segment-rarest policies.

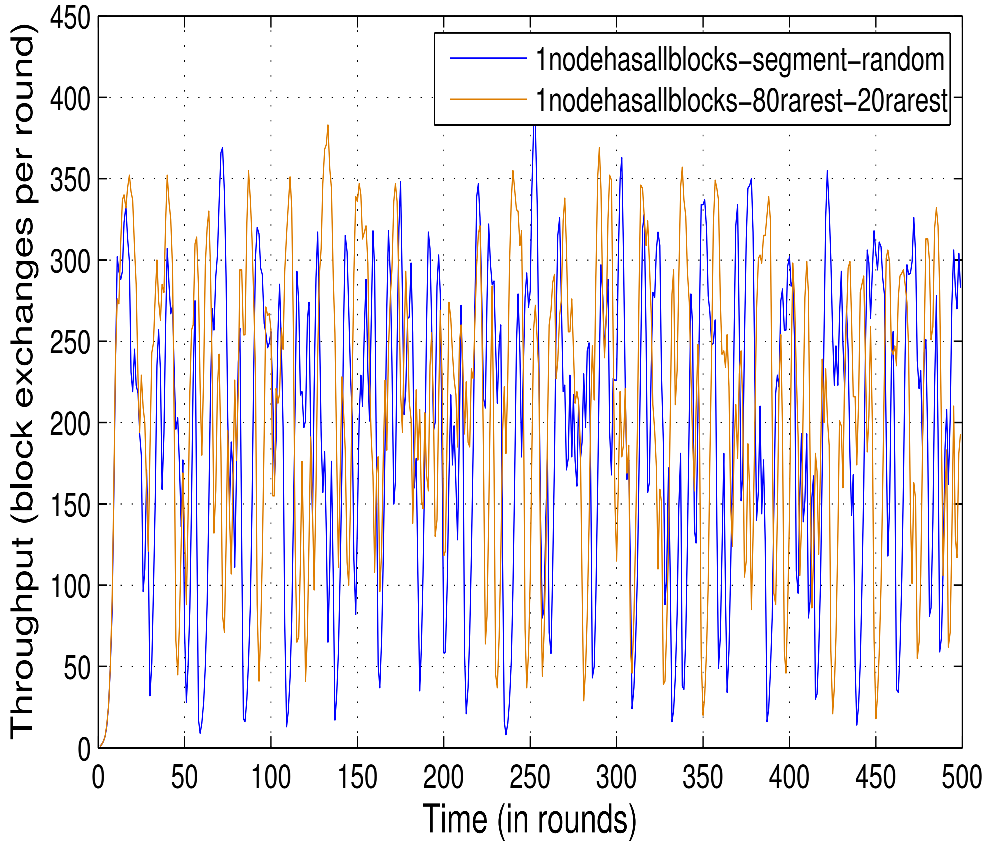


Figure 3.7: This graph shows the system throughput over time with the segment-random and the 80-20 policies.

3.4 Network Coding

In this section, we study network coding techniques to optimize the goodput of the nodes, and the throughput of the system. Network coding has been proposed for improving the throughput of a network in bulk data transfer situations [2, 17, 28]. Network coding makes optimal use of bandwidth resources, and bypasses the block-scheduling problem by allowing all nodes to produce encoded data blocks. A good overview of network coding can be found in [52].

We now present a short description of the benefits and the mechanics of network coding. We illustrate the benefits of network coding with a simple example (Figure 3.8). Assume that node A has already received blocks 1 and 2. Without a scheduler having global knowledge, node B will download block 1 or 2 with equal probability. Simultaneously, let's say node C independently downloads block 1. If node B were to download block 1, the link between B and C would be rendered useless.

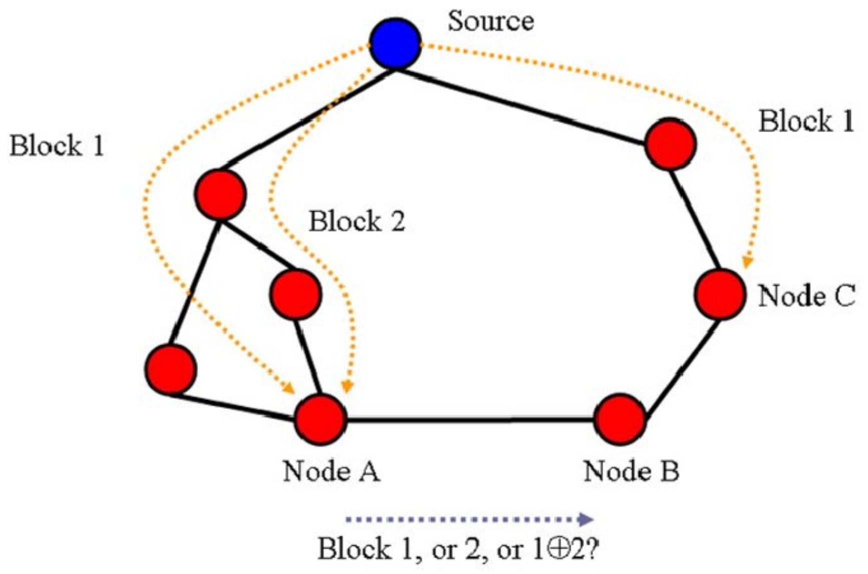


Figure 3.8: This example shows the benefits of network coding when nodes only have local knowledge.

With network coding, however, node A routinely sends a linear combination of the blocks it has (shown in the figure as $1 \oplus 2$) to node B, which can then be used with node C. Note that without a knowledge of the block transfers in other parts of the network, it's not easy for node B to determine the right block to download (and hence the challenge in the scheduling problem). But with network coding, this task becomes trivial.

We now detail how network coding can be used in a peer-to-peer system to disseminate a large file (we will adapt it to our nVoD scenario later). In Figure 3.9, the file exists initially only at the server. When node A contacts the server, the server combines all the blocks of the file to create an encoded block $E1$. The server picks random coefficients $c_1, c_2, \cdots, c_n$, and generates $E1 = \sum_{i=1}^{n} c_i.b_i$, where each b_i represents a block. The server then sends node A, $E1$ and the *coefficient vector* $\overrightarrow{c} = (c_i)$. Note that all the coefficients are chosen from and the operations done in a finite field [1] When node A sends a block to node B, A similarly combines the already encoded blocks it has (namely $E1$ and $E2$) and sends node B, the encoded block $E3 = c_1^{``}.E1 + c_2^{``}.E2$, and the new coefficient vector $c_1^{``}.\overrightarrow{c} + c_2^{``}.\overrightarrow{c'}$.

With network coding, any block that a node receives is useful with very high probability. The downside is that a node often has to wait until it downloads the whole file before it can start decoding the blocks. This is not acceptable in the context of VoD systems where a node wants to play the blocks soon after the download begins. We solve this problem by restricting network coding to segments. A node only needs to wait until it downloads a complete segment before it can start decoding. This limits the benefits of coding since an encoded block is only useful to other nodes interested in a particular segment (rather than being useful to all the nodes). Moreover, this imposes an initial buffering time which is at least one segment size. (Note that non-uniform segment sizes can be used to minimize this start-up delay.) However, coding prevents the occurrence of rare blocks within a

[1]If the finite field is small in size, there could be "collisions" where two nodes pick the same set of coefficients, thereby degrading the performance [35]. Typically, field sizes of 2^{16} provide enough diversity.

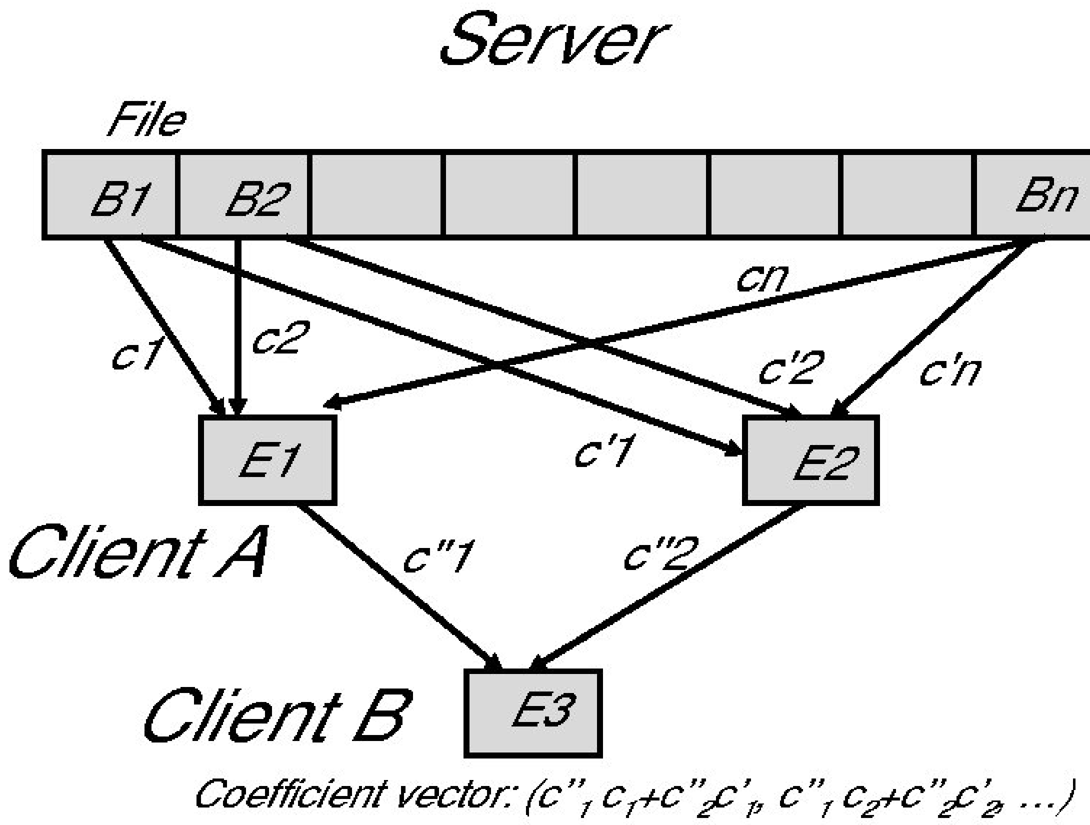
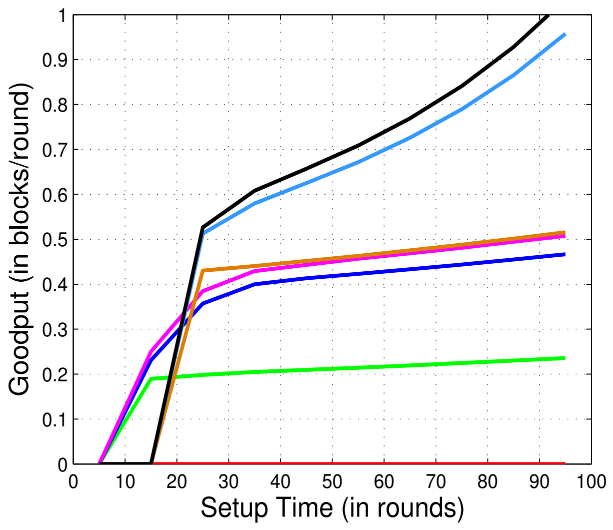

Server
File
B1
B2
Bn
c1
c2
cn
c'1
c'2
c'n
E1
E2
Client A
c"1
c"2
Client B
E3
Coefficient vector: (c"$_1$ c$_1$+c"$_2$c'$_1$, c"$_1$ c$_2$+c"$_2$c'$_2$ ...)

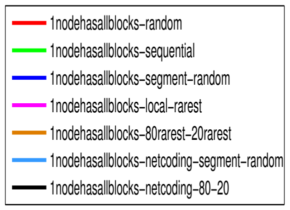
Figure 3.9: An illustration of network coding in RedCarpet.

segment, and ensures that bandwidth is not wasted in distributing the same block multiple times. In essence, coding minimizes the risk of making a wrong upload decision.

We have evaluated the efficacy of network coding with our simulator and our prototype. Fig. 3.10) compares network coding against non-coding heuristics. (Please refer to [7] for the terminology used.) We note that network coding achieves a goodput of ≈ 0.58 blocks/round, while the best rate without using network coding (achieved with pre-fetching) is ≈ 0.44 blocks/round when the setup time is 35 rounds, representing a 31% improvement. (Network coding used along with pre-fetching delivered a goodput of ≈ 0.61 blocks/round). Also, the throughput of the system with network coding is 261.26 block exchanges per round, as compared to 212.21 without coding (Again, network coding along with pre-fetching delivered a throughput of 271.20 block exchanges per round).



(a)



(b)

Figure 3.10: The above graph demonstrates the benefits of network coding vis-a-vis non-coding techniques.

3.5 Implementation

With the insights obtained from our experiments with the simulator, we have developed a prototype in C# to validate our results in a realistic setting. Our system resembles other typical peer-to-peer systems [27], and consists of three types of participants: peers, a tracker, and a logger. The *server* is a special peer that seeds content into the system. The tracker is based on BitTorrent [20] – it maintains a list of all the peers in the system, and when a peer joins the system, the tracker supplies it with a random subset of active peers so that the new peer can form its neighbourhood. The active peers periodically report to the tracker (e.g. information about their content, rates etc.), and can also request the tracker to provide them with a new subset if they have too few neighbors. The logger is an aggregation point for peer and tracker trace messages. Every peer in the system reports detailed statistics to the logger; using those statistics we were able to perform an in-depth evaluation of the various system parameters. We rate-limit the upload and download capacities of the peers using a token bucket based algorithm.

In our prototype implementation, each peer maintains 6-8 connections to other peers. The peers periodically connect to other peers at random and drop connections in an attempt to find better neighbors and increase their download rates. The encoding and decoding operations, for experiments involving network coding, were done over a Galois Field GF(2^{16}). In most of our experiments, the video file was 128MB in size, and was divided into 100 "original" blocks each of which was 1.28MB in size. (we have also experimented with larger number of blocks obtaining similar results).

The implementation is about 25000 lines of code in C#. We have used our implementation to study small scale scenarios, in order to highlight some of the techniques which are indispensable towards building an efficient peer-to-peer nVoD system.

3.5.1 Evaluation against non-coding techniques

We will now present the results from our implementation, and evaluate the benefits of network coding. Consider a flash-crowd where 20 clients B_n join the network. The server has the entire file (100 blocks) divided into 10 segments; network coding is applied over all the blocks in a segment. A segment is decoded on-the-fly as soon as 10 linearly independent blocks are received for each segment.

We compare this to a *global-rarest* policy which does not use network coding. In the global-rarest scheme, a client requests the globally rarest block in the target segment of its interest, either from the server or from its neighborhood. We note that this scheme requires global information which is not available in such a system, and is considered only for comparison purposes. We also note that this policy works the best amongst non-coding policies; in particular, it works better than pre-fetching which, of course, does not rely on global system information.

In Figure 3.11, we show the average number of all blocks as well as useful blocks that are available at the nodes in the system with the global-rarest and network coding policies. The bars mark the maximum and minimum values. Given that global-rarest uses global information about the system, we would expect that it performs optimally. However, this is not the case, and network coding provides a higher throughput than the global-rarest scheme ($\approx$ 14% better). And more importantly, it results in significantly less variance and more predictable download times.

In summary, network coding minimizes the risk of uploading duplicate blocks within a segment, resulting in high system throughput/goodput. We note that while we used network coding within a segment, the nodes still fetch the segments in sequential order (or use pre-fetching). To further improve the performance of

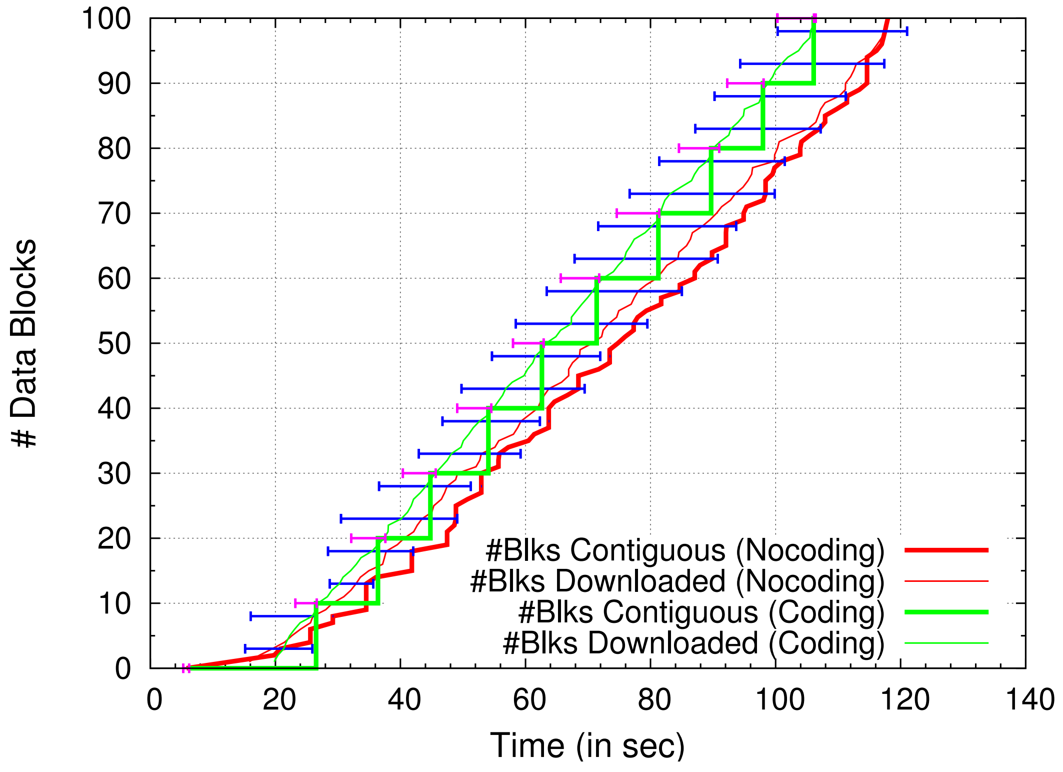


Figure 3.11: The above graph shows the average number of all blocks as well as useful blocks available at the nodes with the global-rarest policy and with network coding over segments.

the system, the next section considers better algorithms for scheduling the fetching of segments.

3.6 Segment scheduling policies

Segment scheduling forms the analogue to the block scheduling problem ([7]) at the segment granularity. As with naïve block scheduling, we show that a naïve segment policy where clients greedily request blocks from their earliest incomplete segments adversely affects the system throughput. And while block scheduling inside a segment is amenable to network coding, coding cannot be used *across segments* since the entanglement it creates prevents streaming VoD. Instead, we propose a heuristic-based solution that schedules segments according to how poorly they are seeded in the network. This approach is similar in spirit to the segment-rarest approach that we discussed earlier.

Segment policy affects the overall throughput of the system when some of the segments are poorly represented in the network. We will examine a representative scenario where a bandwidth-constrained node, that contains blocks from both under-represented and popular segments, uploads blocks from the under-represented segment; this scenario usually occurs when a flash-crowd arrives in the middle of an ongoing download. Consider a server that has the entire file, which is divided into 10 segments containing 10 blocks each. The block policy used within a segment is network coding as described in Section 3.4. One client A has downloaded 75% of the file, when a flash-crowd of 20 clients B_n join the network. For simplicity, we configured the nodes to have equal upload and download capacities.

With naïve segment scheduling where each node greedily requests from the segment it requires, the server's upload capacity is shared between client A, which requests blocks from segments near the end of the file, and the different clients $B_{1..n}$ (number depending on the outbound degree of the server), which request blocks from the first segment(s). Since only the server has the end of the file required by A, and the flash-crowd causes the server's available upload bandwidth to be used in sending blocks from the early segments, the throughput of A is decreased. Also, since the clients $B_{1..n}$ all request from the same initial segment, the server potentially wastes its bandwidth in repeatedly uploading blocks from the same initial segment, i.e., it could serve more than the minimum of 10 blocks required to reconstruct the initial segment within the cooperative cache formed by the clients; this hurts the performance of $B_{1..n}$. Thus, both A as well as $B_{1..n}$ are worse off with naïve segment scheduling.

Figure 3.12 shows the 95th percentile time required to download a given number of blocks, both by A and the newly joined clients $B_{1..n}$. Error bars mark the maximum and minimum values. We observe, from the figure, that with a naïve segment scheduling policy, A initially enjoys a good throughput, rate-limited only by the server's upload bandwidth, until the flash-crowd joins. After this point, A's throughput is severely reduced as the server re-uploads the initial parts of the file to some B_is. The server's upload bandwidth is thus wasted in uploading these segments which are already represented in the network (at A).

The overall throughput of the system would improve if all the nodes contribute to improving the diversity of segments in the network. If we change the segment policy to upload a block from a lesser represented segment whenever possible (*worst-seeded-first policy*), throughput improves significantly for both A and the new nodes $B_{1..n}$ as seen in Figure 3.12. Note that A's throughput is noticeably increased towards the end of the file, because the server continues to serve blocks from later segments to B_i, and A subsequently retrieves these blocks from B_i.

Algorithm 1 describes our worst-seeded-first segment scheduling policy in pseudo-code. We assume that every source node has knowledge of the rarity of segments aggregated across the peers in the system; the aggregation can be performed either centrally at the tracker, or can be approximated in a distributed fashion by gossiping between neighboring nodes. In our implementation, we perform the aggregation at a central tracker.

We will now briefly describe our heuristic to improve segment diversity in the system. Amongst the candidate segments available at the source node and not

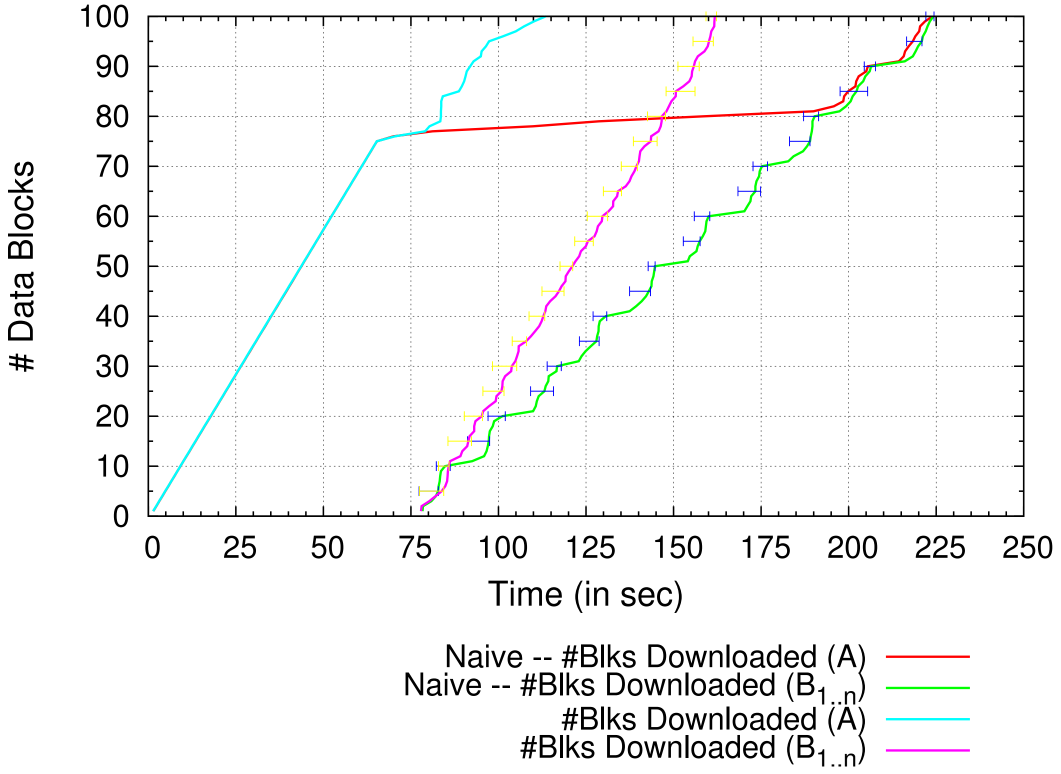


Figure 3.12: The above graph shows the 95th percentile time (shown on the x-axis) required to download a given block (shown on the y-axis). The scenario is when A has already downloaded 75% of the file and a flash-crowd $B_{1..20}$ joins the system. The results for both the naïve and the worst-seeded segment scheduling policies are shown.

Algorithm 1 SelectSegment(s,d)

```
Require: s is source node
Require: d is destination node
 1: $A_S \leftarrow$ AvailableSegments(s)
 2: $C_D \leftarrow$ CompletedSegments(d)
 3: P $\leftarrow$ SortSegmentsWorstSeededFirst($A_S \setminus C_D$)
 4: $C_S \leftarrow$ CompletedSegments(s)
 5: $I_D \leftarrow$ EarliestIncompleteSegment(d)
 6: for $i = 0 \ldots$ Count(P) do
 7:     if $P_i = I_D$ or $P_i \in C_S$ then
 8:         return $P_i$
 9:     end if
10: end for
11: return $P_1$
```

available in full at the destination node (lines 1–3), we pick the segment that is least well-represented (lines 3,6) subject to the conditions on line 7, to send to the destination node. If the poorly seeded segment is immediately of interest to the destination then it is uploaded (clause 1, line 7); otherwise, the source uploads blocks from segments it has completed downloading (clause 2, line 7), in order to ensure that the block is globally innovative with high probability; else, the source simply uploads blocks from the segment which it considers to be the rarest, even though it has only partially downloaded the segment.

A caveat with always uploading the worst-seeded segment is that even when the segment is represented ”widely enough”, a new client is often forced to chase segments that offer minimal benefits both to itself as well as the system. This problem arises when a client joins a mature system where enough copies of the entire file exist. Hence, the computation of worst-seeded segments (line 3) uses a threshold mechanism whereby two segments that are represented at a fraction of nodes greater than a given threshold are considered equally well-seeded. The sorting algorithm, when comparing equally well-seeded segments, prefers the earlier amongst the two. The threshold is set adaptively; in our implementation, it is set to 20% of the active clients. Thus, a new client can receive blocks from segments that are of immediate interest to itself without being penalized.

Note that our algorithm hinges crucially on the availability of a *good enough* estimate for the popularity of the segments in the system. This estimate should include nodes that have the complete segment, as well as those that have only partially downloaded the segment. In our implementation, the tracker aggregates information about the rarity of segments in the system; the clients report the fraction of blocks they have received from each segment to the tracker periodically. These fractions are used to estimate the popularity of the segments; for example, a segment is considered under-represented if the vast majority of nodes have very few blocks from that segment.

We have shown that naïve greedy segment scheduling policies do not work well in certain scenarios. We have proposed the worst-seeded-first segment scheduling policy to improve the throughput of the system. This policy achieves the goal by preventing the formation of rare segments, which increases the diversity of the cooperative cache, and hence the throughput of the system.

3.7 Topology Management

In this section, we will investigate the role of system topology (the arrangement of the nodes w.r.t. each other, viz., the state of the system as represented by an adjacency graph) in improving the goodput of the system. We again use the flash-crowd scenario from Section 3.6 as our running example, in order to study the behaviour of the system.

We will first illustrate how the lack of topology management significantly impacts the goodput of individual clients. For instance, consider Figure 3.13 that plots the 95th percentile time required to download a given number of blocks, by an early node A that has downloaded 75% of the file, and a collection of late arrivals $B_{1..20}$, when using the worst-seeded-first segment scheduling policy (with thresholds) described in Section 3.6. Note that A and $B_{1..20}$ are interested in downloading different parts of the video and, hence, have competing interests. The steps in the graph indicate when a node finishes downloading all the coded blocks from a segment, and hence can decode and watch that segment. We observe that A sees a consistent goodput (note that the goodput curve for A regularly intercepts the throughput curve), implying that A always downloads blocks that are immediately useful to it. In contrast, B_i have worse goodput as can be observed from the graph where the throughput and the goodput curves for B_i are apart.

The problem with this scenario is two-fold. First, the B_is do not benefit from the blocks they receive from the server, since the server gives them blocks from segments near the end of the file (which happen to be the worst-seeded). Thus, in this case, throughput does not translate into goodput. Second, A itself receives some of the blocks that it needs from B_is, thereby incurring the delay for an extra hop through a B_i. Both of these problems could be solved if A forms one cloud

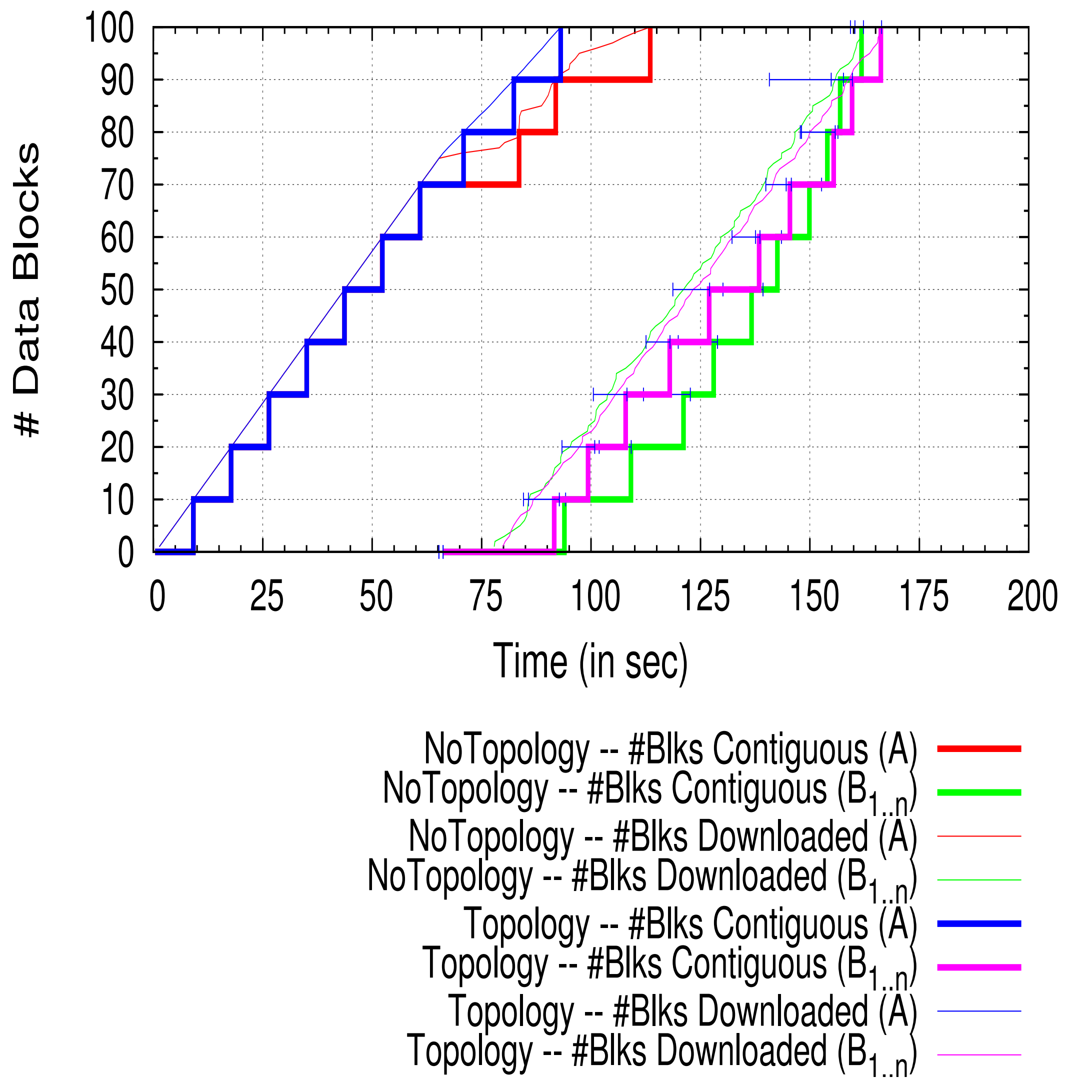


Figure 3.13: The above graph shows the 95th percentile time (shown on the x-axis) required to download a given block (shown on the y-axis). The scenario is when A already has 75% of the file and a flash-crowd $B_{1..20}$ joins the system. The results with and without topology management are shown.

and the B_is form another; and the blocks from the server to A, and from A to the B_is. The goal of topology management then is to create (overlay) connections that improve the overall goodput of the system, without compromising the throughput benefits achieved through segment scheduling. In effect, topology management policies impose a beneficial structure on the mesh network.

We now briefly describe our topology management in Algorithm 2. The algorithm essentially tries to retain connections that demonstrate a high goodput. Thus, it encourages peers targeting the same parts of the video to communicate with each other, resulting in a high goodput for the entire duration of the download. A node S is allowed to upload to another node D if S has not already saturated its upload bandwidth (line 1). If S's upload capacity is used up, then S will accept D only if D is currently targeting the *worst-seeded* segment as calculated by S (lines 4–6). If by accepting D, S crosses its configured limit for the number of connections, S drops an existing connection that provides the least goodput to the corresponding neighbour (lines 9–10). Periodically, nodes that have spare download capacity request a set of random nodes from the tracker and attempt to download from them; of course, subject to that node's acceptance as per the algorithm. These connection attempts cause nodes in the network to re-evaluate their goodput, and opportunistically improve it. This induced connection-churn also encourages neighbor diversity, and prevents the creation of isolated clusters.

The efficacy of topology management can be seen in Figure 3.13. Our algorithm first restricts B_is from connecting to the server. Instead, B_ns are steered to A as well as other B_is. In this process, the server continues to seed innovative content to A unimpeded increasing A's goodput (as compared with no topology management). At the same time, A serves the beginning of the file to some B_i, which in turn disseminate the initial blocks amongst the other B_is. Once A finishes downloading,

B_is can connect to both A and the server. It is clear from Figure 3.13 that the proposed topology management policy improves the goodput both of the new nodes (B_is) as well as that of the old ones (A).

Algorithm 2 ShouldUploadTo(s,d)

```
Require: s is the source node
Require: d is the destination node
 1: if HasSpareUploadCapacity(s) then
 2:    return true
 3: end if
 4: Id ← EarliestIncompleteSegment(d)
 5: Sd ← SelectSegment(s,d)
 6: if Id ≠ Sd then
 7:    return false
 8: end if
 9: if TooManyUploadConnections(s) then
10:    KickUploadWithWorstPeerGoodput(s)
11: end if
12: return true
```

3.8 Heterogeneous Capacities

In this section, we will revisit our segment scheduling and topology management policies, and refine them to handle heterogeneous networks where nodes have asymmetric upload and download capacities (i.e., the capacities are different). As a motivation for our final enhancement to our algorithms, we present a scenario where a capacity-oblivious segment policy falters.

Consider a network with a fast server, a slow node A that has downloaded 25% of the file and is connected to the server, and a flash-crowd $B_{1..20}$ with a range of bandwidth capacities (i.e., both fast and slow nodes) joins the network. The distribution of the capacities of the nodes $B_{1..20}$ is based on [11]. In this scenario, consistent with the foregoing discussion, some B_i are allowed to connect to the server given that it has spare capacity (Alg. 2, line 2). However, they are forced to download segments that are not of immediate use to them (e.g. segments near the middle or end of the file); they can only receive blocks from their immediate segment of interest through node A which, however, happens to be slow.

Figure 3.14 shows that heterogeneity-oblivious algorithms (as presented above) affect not only the goodput, but also the throughput of fast nodes. In particular, we note that the throughput of the fast nodes is reduced to that of the slow nodes (viz., that of node A). This is because fast nodes are initially forced to depend on a slow node (A), which is the only node giving them ‘good’ blocks.

A small change in the computation of the worst-seeded segment solves this problem. The change involves assigning a weight, to the contribution of each node towards the aggregate seeding of each segment, in proportion to that node’s upload bandwidth. This is a small but significant change w.r.t. traditional block rarest-first policies, since it takes into account not only the number of copies of a given segment, but also how efficiently they can be uploaded into the network. Thus, nodes keep downloading a segment until that segment has an overall upload capacity equal to that of the other “well-represented” segments. Our scheme thus prevents the formation of rare segments.

The result of this change, as shown in Figure 3.14, is a high throughput and goodput for both the fast and the slow nodes, and low start-up latencies. In fact, from Figure 3.14, we can see that the fast nodes can target playback rates that are

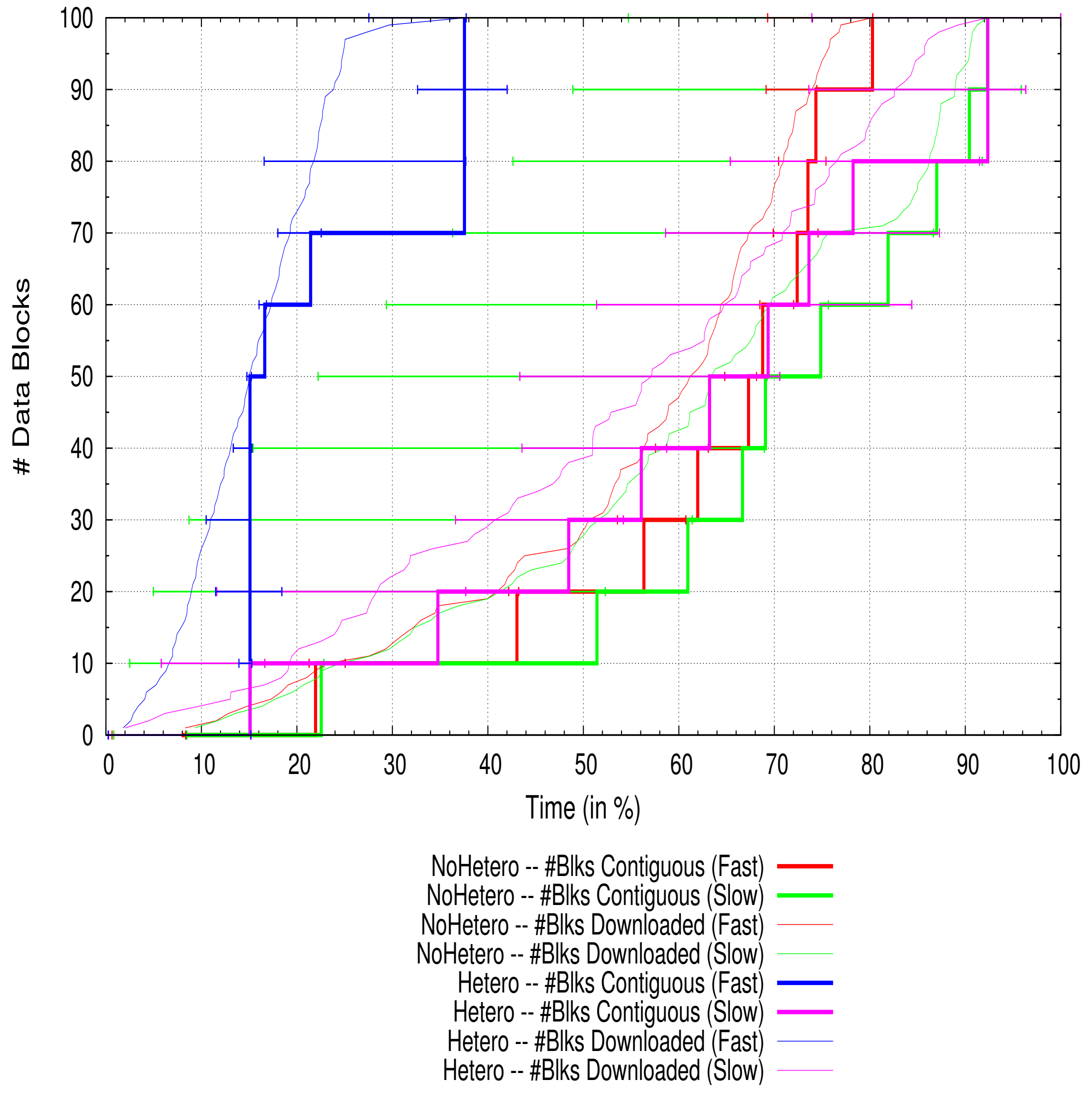


Figure 3.14: The above graph shows the 95th percentile time (shown on the x-axis as a percentage) required to download a given block (shown on the y-axis). The scenario is when A already has 25% of the file and a flash-crowd, $B_{1..20}$ with varying bandwidth capacities, joins the system. The results with and without the heterogeneity-aware algorithms are shown.

very close to their bandwidth, and are not limited by the slow nodes which also experience better performance.

Chapter 4

Related Work

A lot of research has gone into understanding and building scalable data-intensive applications which utilize the resources provided by participants. In this chapter, we will examine this prior work and distinguish our contributions.

Scalable file servers. JetFile [32] is a wide-area network file system designed to scale to large numbers of clients, by using the Scalable Reliable Multicast (SRM) protocol, which is logically layered on IP multicast. JetFile allocates a multicast address for each file. Read requests are multicast to this address; any client which has the data responds to such requests. In JetFile, any client can become the manager for a file by writing to it—which implies the necessity for conflict-resolution mechanisms to periodically synchronize to a storage server—whereas all writes in Shark are synchronized at a central server. Note that while JetFile is intended for read-write workloads, Shark is explicitly designed for read-heavy workloads.

High-availability file systems. Several local-area systems propose distributing functionality over multiple collocated hosts to achieve greater fault-tolerance and availability. Zebra [33] uses a single meta-data server to serialize meta-data operations (*e.g.* i-node operations), and maintains a per-client log of file contents striped across multiple network nodes. Harp [46] replicates file servers to ensure high availability; one such server acts as a primary replica in order to serialize updates. These techniques are largely orthogonal to, yet possibly could be combined with, Shark's cooperative caching design.

Serverless file systems. Serverless file systems are designed to offer greater local-area scalability by replicating functionality across multiple hosts. xFS [6] distributes data and meta-data across all participating hosts, where every piece of meta-data is assigned a host at which to serialize updates for that meta-data. Frangipani [69] decentralizes file-storage among a set virtualized disks, and it maintains traditional file system structures, with small meta-data logs to improve recoverability. A Shark server can similarly use any type of log-based or journaled file system to enable recoverability, while it is explicitly designed for wide-area scalability.

Farsite [1] seeks to build an enterprise-scale distributed file system. A single primary replica manages file writes, and the system protects directory meta-data through a Byzantine-fault-tolerant protocol [15]. When enabling cross-file-system sharing, Shark's encryption technique is similar to Farsite's convergent encryption, in which files with identical content result in identical ciphertexts.

Peer-to-peer file systems. A number of peer-to-peer file systems—including PAST [66], CFS [21], Ivy [54], and OceanStore [44]—have been proposed for wide-area operation and similarly use some type of distributed-hash-table infrastructure ([65, 67, 80], respectively). All of these systems differ from Shark in that they provide a serverless design: While such a decentralized design removes any central point of failure, it adds complexity, performance overhead, and management difficulties.

PAST and CFS are both designed for read-only data, where data (whole files in PAST and file blocks in CFS) are *stored* in the peer-to-peer DHT [65, 67] at nodes closest to the key that names the respective block/file. Data replication helps improve performance and ensures that a single node is not overloaded. In contract, Shark uses Coral to *index* clients caching a replica, so data is only cached where it is needed by applications and on nodes who have proper access permissions to the data.

Ivy builds on CFS to yield a read-write file system through logs and version vectors. The head of a per-client log is stored in the DHT at its closest node. To enable multiple writers, Ivy uses version vectors to order records from different logs. It does not guarantee read/write consistency. Also managing read/write storage via versioned logs, OceanStore divides the system into a large set of untrusted clients and a core group of trusted servers, where updates are applied atomically. Its Pond prototype [64] uses a combination of Byzantine-fault-tolerant protocols, proactive threshold signatures, erasure-encoded and block replication, and multicast dissemination.

Large file distribution. BitTorrent [19] is a widely-deployed file-distribution system. It uses a central server to track which clients are caching which blocks; using information from this meta-data server, clients download file blocks from other clients in parallel. Clients access BitTorrent through a web interface or special software.

Compared to BitTorrent, Shark provides a file-system interface supporting read/write operations with flexible access control policies, while BitTorrent lacks authorization mechanisms and supports read-only data. While BitTorrent centralizes client meta-data information, Shark stores such information in a global distributed index, enabling cross-file-system sharing (for world-readable files) and taking advantage of network locality.

Video streaming over the Internet has seen a lot of research over the past decade, for example, see [16, 76, 8, 37]. Most relevant to our work are video distribution systems that can support a large number of users which are discussed below.

Video streaming using Multicast Multicasting has been proposed to provide a scalable video streaming service, even in the presence of heterogeneous receivers [41, 59, 45]. Multicasting is a natural paradigm for live video streaming. It has also been extended for supporting near-Video-on-Demand services. The simplest approach is to periodically start a new broadcast cycle [4]. More elaborate schemes propose to divide the video into segments and distribute each segment in different multicast channels [72, 38, 37]. The obvious drawback of such systems is that there is no support for native multicasting in the Internet today.

Peer-to-peer systems for live streaming There have been many proposals to use overlay multicast distribution for streaming live events [77, 24, 39, 70, 34, 14]. Such systems unfortunately only support live streaming and not near-Video-on-Demand. It is an open question whether such overlay multicasting approaches can be extended to support nVoD, maybe by using similar approaches as in [72, 38]. Inspired by the success of unstructured P2P networks, the authors in [79, 48, 57] propose to use mesh-based P2P networks for live video streaming. While similar to our approach, these systems do not support nVoD.

Peer-to-peer systems for nVoD The systems which are most relevant to our work are the BASS [22] and BiToS [73] systems. BASS extends the current BitTorrent system [19] to provide a nVoD service [22]. BASS uses a streaming server which all nodes connect to though the nodes use the P2P network to help each other and alleviate the load on the server. Even though BASS reduces the load at the server by a significant amount, the design of the system is still server oriented, and hence the bandwidth requirements at the server increase linearly with the number of users. Note that in RedCarpet, the nodes rely only on the P2P network to retrieve the content; hence, we place a greater emphasis on the performance of the block scheduling algorithms. Also, since we depend on a dynamic and fluctuating P2P network, we use deeper buffering compared to BASS.

The BiToS system is also based on BitTorrent [73]. The main idea is to divide the missing blocks into two sets, "high priority set" and "remaining piece set", and request with higher probability blocks from the high priority set (Note that BiToS too is video-agnostic). While the emphasis is on careful scheduling of the video blocks in BiToS, network coding and an effective topology obviate this problem in our system. Also, we note that BiToS does not deal with issues of topology management and instead adheres to the standard BitTorrent algorithms.

Chapter 5

Conclusions

This thesis has addressed the problem of designing scalable systems for new data-intensive applications. We have presented design techniques to enable a low-end server to support such applications by leveraging the resources of participating users such as disk space, network bandwidth. We have identified two types of such applications – those whose performance increases with the data download rates of the users and usually involve random accesses to large files (e.g. file systems, bulk data distribution systems) and those whose utility requires data download rates which are above a certain threshold and typically involve sequential access to large files (e.g. video streaming). We have built Shark and RedCarpet to illustrate and validate our techniques for building applications of the above two types (respectively).

Shark is a distributed file system designed for large scale usage in the wide-area, and supports read-heavy workloads. Shark clients construct a locality-optimized cooperative cache by forming self-organizing clusters of well-connected machines. They efficiently locate nearby copies of data using a distributed index and stripe downloads from multiple clients in parallel. We have shown that this reduces the load on file servers and delivers significant performance improvements for the clients.

RedCarpet provides near-Video-on-Demand (nVoD) service by organizing the participants into a mesh-based peer-to-peer (P2P) network. In particular, we have addressed the problem of scheduling the dissemination of chunks of a video, in order to achieve a low setup time and a sustained download rate (which is higher than the video encoding rate). We have proposed network coding, worst-seeded-first segment scheduling, and topology management algorithms that render nVoD feasible in such a mesh-based P2P system. We have shown, using simulations as well as a prototype, that these techniques are crucial towards maximizing network utilization and delivering an acceptable nVoD performance.

Bibliography

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec 2002.

[2] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network information flow. *IEEE Trans. on Information Theory*, 46:1204–1216, 2000.

[3] Akamai Technologies, Inc. `http://www.akamai.com/`, 2004.

[4] K. C. Almeroth and M. H. Ammar. On the use of multicast delivery to provide a scalable and interactive Video-on-Demand service. *Journal of Selected Areas in Communications*, 14(6):1110–1122, 1996.

[5] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 131–145, Banff, Canada, Oct 2001.

[6] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roseli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, Feb 1996.

[7] S. Annapureddy, C. Gkantsidis, and P. Rodriguez. Providing video-on-demand using peer-to-peer networks. In *Internet Protocol TeleVision (IPTV) Workshop, WWW '06*, Edinburgh, Scotland, May 2006.

[8] J. G. Apostolopoulos, W.-T. Tan, and S. J. Wee. Video streaming: Concepts, Algorithms, and Systems. `http://www.hpl.hp.com/techreports/2002/HPL-2002-260.pdf`, Sep 2002.

[9] L. Barroso, J. Dean, and U. Hoelzle. Web search for a planet: The google cluster architecture. `http://research.google.com/archive/googlecluster.html`.

[10] M. Bellare, R. Canetti, and H. Krawczyk. Keyed hash functions and message authentication. In *Advances in Cryptology—CRYPTO '96*, Santa Barbara, CA, Aug 1996.

[11] A. R. Bharambe, C. Herley, and V. N. Padmanabhan. Analyzing and improving a bittorrent network's performance mechanisms. In *Proceedings of IEEE INFOCOM 2006*, Barcelona, Spain, Apr 2006.

[12] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, Jun 1995.

[13] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct 2003.

[14] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *ACM SOSP'03*, Lake Bolton, New York, USA, Oct 2003.

[15] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Operating Systems Design and Implementation (OSDI)*, San Diego, Oct 2000.

[16] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. A distributed real time MPEG video audio player. In *NOSSDAV*, 1995.

[17] P. A. Chou, Y. Wu, and K. Jain. Practical network coding. In *Allerton Conference on Communication, Control, and Computing*, Oct 2003.

[18] Y.-H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Measurement and Modeling of Computer Systems*, pages 1–12, 2000.

[19] B. Cohen. BitTorrent. `http://www.bittorrent.com`.

[20] B. Cohen. Incentives build robustness in bittorrent, 2003.

[21] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct 2001.

[22] C. Dana, D. Li, D. Harrison, and C.-N. Chuah. BASS: BitTorrent assisted streaming system for video-on-demand. In *International Workshop on Multimedia Signal Processing (MMSP)*. IEEE Press, 2005.

[23] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors: Load sharing among workstation clusters. *Journal on Future Generations of Computer Systems*, 12:53–65, 1996.

[24] End system multicast. `http://esm.cs.cmu.edu/`, 2005.

[25] Feidian. `http://tv.net9.org/`.

[26] M. J. Freedman, E. Freudenthal, and D. Maziéres. Democratizing content publication with Coral. In *Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar 2004.

[27] C. Gkantsidis, J. Miller, and P. Rodriguez. Anatomy of a p2p content distribution system with network coding. In *IPTPS*, 2006.

[28] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *IEEE Infocom*, 2005.

[29] Gnutella. `http://gnutella.wego.com/`.

[30] Google. Google maps. `http://maps.google.com/`.

[31] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–210, Dec. 1989.

[32] B. Gronvall, A. Westerlund, and S. Pink. The design of a multicast-based distributed file system. In *Operating Systems Design and Implementation (OSDI)*, pages 251–264, 1999.

[33] J. H. Hartman and J. K. Ousterhout. The zebra striped network file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, Dec 1993.

[34] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava. Promise: Peer-to-peer media streaming using collectcast. In *Multimedia*. ACM Press, 2003.

[35] T. Ho, M. Mdard, M. Effros, and D. Karger. On randomized network coding. In *41st Allerton Annual Conference on Communication, Control and Computing*, Oct 2003.

[36] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb 1988.

[37] A. Hu. Video-on-demand broadcasting protocols: A comprehensive study. In *IEEE Infocom*, pages 508–571. IEEE Press, Apr 2001.

[38] K. A. Hua and S. Sheu. Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems. In *ACM SIGCOMM*, pages 89–100. ACM Press, 1997.

[39] Y. hua Chu, A. Ganjam, T. E. Ng, S. G. Rao, K. Sripanidkulchai, J. Zhan, and H. Zhang. Early experience with an internet broadcast system based on overlay multicast. In *USENIX Annual Technical Conference*. USENIX, 2004.

[40] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Operating Systems Design and Implementation (OSDI)*, pages 197–212, 2000.

[41] V. Kompella, J. Pasquale, and G. Polyzos. Multicasting for multimedia applications. In *IEEE Infocom'92*, volume 3, pages 2078–2085. IEEE Press, May 1992.

[42] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 282–297, Bolton Landing, NY, USA, Oct 2003.

[43] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). *Lecture Notes in Computer Science*, 2139, 2001.

[44] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *ASPLOS*, Cambridge, MA, Nov 2000.

[45] X. Li, S. Pauly, and M. Ammar. Video multicast over the internet. *IEEE Network*, 1999.

[46] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. eplication in the Harp file system. *Operating Systems Review*, 25(5):226–238, Oct 1991.

[47] P. Lyman and H. R. Varian. Broadcast media. `http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/broadcast.htm`.

[48] N. Magharei, A. Rasti, D. Stutzbach, and R. Rejaie. Peer-to-peer receiver-driven mesh-based streaming. In *ACM SigComm (poster session)*. ACM Press, 2005.

[49] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, Mar 2002.

[50] D. Mazières. A toolkit for user-level file systems. In *USENIX*, Boston, MA, Jun 2001.

[51] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, Dec 1999.

[52] M. Medard, R. Koetter, and P. A. Chou. Network coding: A new network design paradigm. In *IEEE International Symposium on Information Theory*, Adelaide, Sep 2005.

[53] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.

[54] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec 2002.

[55] D. Newman. UCI KDD archive. `http://kdd.ics.uci.edu/`.

[56] W. B. Norton. Video internet: The next wave of massive disruption to the U.S. peering ecosystem (v0.91). `http://www.pbs.org/cringely/pulpit/media/InternetVideo0.91.pdf`.

[57] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, Feb 2005.

[58] A. Parker. P2P in 2005. `http://www.cachelogic.com`, 2005.

[59] J. C. Pasquale, G. C. Polyzos, and G. Xylomenos. The multimedia multicasting problem. *ACM Multimedia Systems*, 6:43–59, 1998.

[60] PlanetLab. `http://www.planet-lab.org/`, 2004.

[61] PPLive. `http://www.pplive.com/`.

[62] M. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[63] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, San Diego, CA, Aug 2001.

[64] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the oceanstore prototype. In *FAST*, Berkeley, CA, Mar 2003.

[65] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, Nov 2001.

[66] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct 2001.

[67] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *IEEE/ACM Trans. on Networking*, 2002.

[68] Swivel. Cost per megabyte of hard drive space. `http://www.swivel.com/data_columns/spreadsheet/1924958`.

[69] C. Thekkath, T. Mann, and E. Lee. Frangipani: A scalable distributed file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, Saint Malo, France, Oct 1997.

[70] D. A. Tran, K. A. Hua, and T. Do. Zigzag: An efficient peer-to-peer scheme for media streaming. In *IEEE Infocom*. IEEE Press, 2003.

[71] United States Census Bureau. Census data access tools. `http://www.census.gov/main/www/access.html`.

[72] S. Viswanathan and T. Imiehnski. Pyramid broadcasting for video on demand service. In *IEEE Multimedia Computing and Networking Conference*. IEEE Press, 1995.

[73] A. Vlavianos, M. Iliofotou, and M. Faloutsos. BiToS: Enhancing BitTorrent for supporting streaming applications. In *IEEE Global Internet*, 2006.

[74] A. Westerlund and J. Danielsson. Arla—a free AFS client. In *1998 USENIX, Freenix track*, New Orleans, LA, Jun 1998.

[75] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for

distributed systems and networks. In *Operating Systems Design and Implementation (OSDI)*, pages 255–270, Boston, MA, Dec 2002. USENIX Association.

[76] D. Wu, Y. T. Hou, W. Zhu, Y.-Q. Zhang, and J. M. Peha. Streaming video over the internet: Approaches and directions. *IEEE Tran. on circuits and systems for video technology*, 11(3):282–300, Mar 2001.

[77] D. Xu, M. Hefeeda, S. Hambrusch, and B. Bhargava. On peer-to-peer media streaming. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*. IEEE Press, 2002.

[78] M. Zhang, J.-G. Luo, L. Zhao, and S.-Q. Yang. A peer-to-peer network for live media streaming using a push-pull approach. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 287–290, New York, NY, USA, 2005. ACM.

[79] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum. CoolStreaming/DONet: A data-driven overlay network for peer-to-peer live media streaming. In *IEEE Infocom*. IEEE Press, 2005.

[80] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J. Selected Areas in Communications*, 2003.