UNIVERSITY OF CALIFORNIA, SAN DIEGO

The Composite Endpoint Protocol (CEP): High-Performance Partial Content Distribution

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

Doctor of Philosopy

in

Computer Science

by

Eric Haynes Weigle

2007

The Dissertation of Eric Haynes Weigle is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2007

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACKNOWLEDGEMENTS

I would like to thank my advisor Andrew A. Chien, the administrators Troy Chuang, Adam Brust, Patricia Bladh, Jenine Combs, and the other past and present members of the CSAG group for their support and feedback over the past several years.

Material from Chapters 2, 4, 5 and 7, in part, appeared in “The Composite Endpoint Protocol (CEP): Scalable Endpoints for Terabit Flows,” Eric Weigle and Andrew A. Chien, Proceedings of IEEE Conference on Cluster Computing and the Grid (CCGRID), 2005. The dissertation author was the primary investigator and author of this paper [120].

Material from Chapters 3, 4, 6, and 7, in part, appeared in “Partial Content Distribution on High Performance Networks,” Eric Weigle and Andrew A. Chien, Proceedings of the IEEE International Symposium on High-Performance Distributed Computing (HPDC2007), June 2007. The dissertation author was the primary investigator and author of this paper [121].

Material from Chapter 8, in part, appeared in “Peer-to-Peer Error Recovery for Hybrid Satellite-Terrestrial Networks,” Eric Weigle, Matti Hiltunen, Rick Schlichting, Vinay A. Vaishampayan, and Andrew A. Chien, Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing (P2P2006), September 2006. The dissertation author was the primary investigator and author of this paper [122]. Many thanks to the researchers at AT&T Labs for their ideas, feedback, and efforts on my behalf over the last two years.

This work was supported in part by the National Science Foundation under awards NSF Cooperative Agreement ANI-0225642 (OptIPuter), NSF CCR-0331645 (VGrADS), NSF ACI-0305390, and NSF Research Infrastructure Grant EIA-0303622. Support from the UCSD Center for Networked Systems, BigBangwidth, and Fujitsu is also gratefully acknowledged.

VITA

2000	Bachelor of Science in Computer Science and Mathematics, University of Wisconsin, Madison
2000-2004	Technical Staff Member; Visiting Scientist, Los Alamos National Laboratory
2003-2007	Graduate Student Researcher, University of California, San Diego
2005	Master of Science in Computer Science, University of California, San Diego
2005-2007	Summer Intern in Management; Contractor, AT&T Labs- Research
2007	Doctor of Philosophy in Computer Science, University of California, San Diego

ABSTRACT OF THE DISSERTATION

The Composite Endpoint Protocol (CEP): High-Performance Partial Content Distribution

by

Eric Haynes Weigle

Doctor of Philosophy in Computer Science

University of California, San Diego, 2007

Professor Andrew A. Chien, Chair

This dissertation introduces the Composite Endpoint Protocol (CEP) which solves two related problems: large-scale high performance transfers, and partial content distribution. Achieving high performance in large-scale networks, with speeds above 1Gbps and latency up to 200ms, is difficult; individual machines can not fully exploit overall system capacity, and existing protocols (e.g. TCP) have well-known problems. Similarly, while whole-file content distribution is well studied, when individual clients each desire *different* parts of a file new techniques are required. The core algorithms and abstractions needed to exploit large scale networks or provide sub-file distribution semantics do not exist.

The underlying problem is fundamental: *transfer scheduling*. Given a set of heterogeneous nodes which *have* data and nodes which *need* some subset of that data, perform transfers to best satisfy all nodes' demands. No strong semantics are implied here; subsets of this data may be replicated, missing, not fall on block/word boundaries, etc. The solution is a *transfer scheduler* which implicitly or explicitly specifies *which* nodes transfer *what* data and *when*.

CEP solves the transfer scheduling problem using minimal centralization for metadata/scheduling and infrastructure for fully distributed data transmission. Hybrid centralized/distributed algorithms and heuristics dynamically generate the most desirable transfers as system state evolves. In this way, CEP enables both large-scale high performance transfers and provides rich partial content distribution semantics. The dissertation includes the following contributions:

1. An *efficient* mechanism for multiple heterogeneous nodes/processes (a composite endpoint) to take part in a single logical connection, where core algorithms run in $O(n\ log\ n)$ for the common case;

2. *Simple*, *flexible* interfaces for describing data layouts and composite endpoint communication, backed by a *general* mathematical abstraction;

3. Multiple transfer scheduling algorithms which produce *high performance* (over 10 Gbps), *high resolution*, and when possible provably *optimal* output, with detailed analysis of each;

4. A *scalable* and *robust* composite endpoint architecture which supports tens of thousands of participants and transparently survives server failures.

We describe the theoretical and real-world underpinnings of this problem, including in-depth analysis of the algorithms involved, discuss two implementations of the Composite Endpoint Protocol, as provide an empirical evaluation showing the benefits of CEP under a variety of conditions: over $10\times$ faster than Apache, BitTorrent, DHTs, or uniform striping techniques.

Chapter 1: Introduction

This dissertation introduces techniques enabling (1) *large-scale, high performance transfers*, with (2) *partial content distribution semantics*. Large scale means transfers involving up to tens of thousands of participants. High performance means close to underlying hardware limitations, providing tens of gigabits of bandwidth with millisecond latency in LAN environments. Partial content distribution means that each participant may supply or demand arbitrary parts of a logical data set.

In other words, the Composite Endpoint Protocol (CEP) provides a rich model allowing thousands of machines to specify their individual desires, which CEP then efficiently satisfies. Providing any one of these features is itself difficult problem. Providing all of them together is even more difficult; no existing approach does so.

First, "large scale" entails coordinating large numbers of machines in a distributed system, where machines may have heterogeneous hardware, network connectivity, be in different administrative domains, etc. Some may even fail during the transfer. We must determine *capacity* of individual transfer participants and *detect failures*. Existing approaches have problems exploiting heterogeneity, tolerating failures, or with scalability.

Second, "partial content distribution" entails allowing each individual machine to supply or demand any arbitrary set of bytes; not necessarily falling on block/word boundaries, nor necessarily in a contiguous range. Arbitrary subsets of data may be replicated on multiple machines, and there may be no machines serving some data. Machines may simultaneously serve some data and want other data. We must determine who *has* what data, who *wants* what data. Existing approaches do not capture these rich semantics; data is instead assumed to be in fixed-size blocks and all machines are assumed to want the same data. They do not provide the infrastructure necessary to collect this information.

Third, “high performance” in such a large, complex system entails a similarly large, complex optimization problem. We call this the *transfer scheduling problem*. Given information on system state– who has/wants which data and capacities, as above– the system must determine how to satisfy all participants. That means determining a *transfer schedule*: *who* communicates, *what* data they send, and at what *rate*, such that everyone get exactly what they want as quickly as possible. Existing approaches do not attempt to optimize globally. Instead they focus on individual transfer participants and provide minimal global functionality.

Transfer scheduling is a fundamental distributed systems problem encountered in a variety of situations. Any large scale data transfer, be it physics, biology, or geology data sets, ISO images for Linux distributions, off-site repositories for system backups, or media/content distribution networks (CDNs), etc., is solving an instance of the transfer scheduling problem. Peer-to-peer file sharing is a distributed instance of the problem, typically optimizing for robustness. Distributed file systems and distributed memory implicitly solve instances of the transfer scheduling problem, providing block or whole-file semantics.

A more concrete example is cluster-to-cluster file transfer, which initially motivated this work. Here, individual cluster nodes’ capacity (e.g. 1Gbps Ethernet links) is small relative to total system capacity. To utilize high bandwidth (e.g. 10Gbps) links we must exploit multiple nodes. This means determining which nodes communicate, what data they send, and at what rates: a straightforward instance of the transfer scheduling problem. Since total system performance is more important than the work individual nodes perform, there is flexibility to optimize by, e.g., placing load on more powerful, lightly loaded, or well-connected nodes. Section 2.2 discusses these examples in more depth.

While prior work focused on special cases, CEP solves the most general form of the transfer scheduling problem. We capture a more complex set of transfer constraints, providing richer partial-content transfer semantics. We work on a larger range of systems, including transfers on heterogeneous nodes and transfers with many participants. We provide higher performance in many environments, achieving higher bandwidth, lower latency, and more robust transfers.

Specifically, we claim that CEP, the Composite Endpoint Protocol, provides an *efficient* mechanism for multiple heterogeneous nodes (a composite endpoint) to take part in a single logical connection; has *simple*, *flexible* interfaces for describing data layouts and communication constraints, backed by a *general* mathematical abstraction; provides multiple transfer scheduling algorithms which produce *high performance*, *high resolution*, and (when possible) provably optimal output; and has a *scalable* and *robust* architecture which supports large numbers of participants and tolerates failures.

We achieve these features using graph-structured transfer scheduling algorithms, heuristics, and partially centralized metadata management infrastructure. Transfer scheduling is an optimization optimization problem which can be attacked with linear programming, greedy algorithms, implicit scheduling, or other techniques; together these form the core of our approach. Similarly, efficient detection, representation, and management of metadata is necessary to enable transfers with complicated requirements.

The rest of this chapter discusses high-level motivating factors for this work, our new contributions, and gives the actual thesis statement. We conclude with an outline of the dissertation.

1.1 Motivation

CEP targets high performance and rich transfer semantics. Motivating the former are historical trends in hardware and software- these have have led to a world where capacity is available but underutilized. User demands and advances in distributed systems motivate the latter. This section discusses these trends and relevant background.

1.1.1 Exploiting Hardware Trends

Over the last several decades, computer technology has changed dramatically. Today we are entering a world where such incredible amounts of core bandwidth are potentially available that end node capacity is a bottleneck. This is even true given the oft-cited Moore's Law [79], which as commonly phrased states that computational power doubles every 18 months. A similar law exists for network capacity [28], but network speeds are growing even faster than computational speeds [105].

Networking technology is shifting from electric to optical signals, which provide two main benefits. First, optical fiber can carry high speed signals much farther without repeaters. Second, multiple wavelengths of light can be utilized on a single fiber with almost no interference. Thus both signal fidelity and density are very high. In practical terms, 6.4Tbps/fiber was demonstrated by NEC in the year 2000 using 160 channels at 40Gbps each [33]; and more recently over 1000 wavelengths per fiber has been demonstrated [58]. That leads to a potential capacity of over 40 Terabits per second, *per fiber*. This kind of capability has driven large companies to snap up spare ("dark") fiber [51].

The problem in this environment is that end nodes run at much slower speeds– typically less than 1Gbps– a fraction of available bandwidth. To exploit available capacity, we must use nodes in parallel. Beowulf clusters [16, 102] are attractive for this purpose: built from commodity components, they provide low cost, scalable infrastructure. They are commonly used for physics simulations [24], biology/bioinformatics [12, 80], data mining, and even supercomputing [115]– all tasks that may have large scale transfer requirements.

A final hardware problem is heterogeneity. Even in "homogeneous" clusters, nodes have heterogeneous features due to failures, transient load, wiring limitations, or configuration issues. Given rapid change in technology, nodes purchased even a few months apart may differ in CPU speeds, memory, etc. The problem of heterogeneity is exacerbated when including peers on the Internet- nodes with DSL or cable access are slower by orders of magnitude. While hardware technology and physical infrastructure exists, transfer scheduling software to effectively utilize complex heterogeneous environments does not.

1.1.2 Providing Rich Software Semantics

Systems with multiple nodes have a variety of communication options- the combination of which nodes communicate, what they send, how fast they send provides a large space in which to make decisions. While prior work makes a variety of simplifying assumptions to manage this complexity, we show it is possible to capture arbitrary user constraints and act upon them efficiently. This allows us to provide a unified framework encompassing a variety of use models with a single scheduling and optimization engine. We support communication patterns that are impossible to do efficiently in block-based systems; for example handling combinations of small, oddly-sized pieces and bulk data in a single transfer.

Put another way, we capture and solve a superset of the problems prior work can even represent. This includes problems such as striped file transfer, where peers each send disjoint portions of a file, distributed editing, where individual peers work on small pieces of a file, or data scatter/gather, where some peers have the entire file while others need only portions of it, or vice versa. Our unified framework also means that improvements to the underlying algorithms or software improve performance across all environments. Our simple interfaces (see Section 5.2) enable users to exploit these features without concern for the underlying model.

Another important example is traditional whole-file content distribution– "get identical copies of a data set to every machine." There is a large body of work focusing on such replication [3,29,65,81]. We support this and also let users specify arbitrary data demand/-constraints, not necessarily a whole file or block-aligned pieces of one. This captures a wider variety of situations. It provides new challenges, such as byte-range-based metadata management, and optimization opportunities, such as constraint structure we can exploit.

The distinguishing feature between use models is the level of *sharing*. For traditional content distribution, all peers want the *same* data: they share demand. For partial content distribution, peers may each desire *different* data: lower sharing of demand. Low-sharing environments have limited opportunities to exploit peer-to-peer transfers. To achieve good performance it is critical to efficiently (1) discover data supply/demand constraints, and (2) allocate capacity satisfying peers' differing constraints.

1.1.3 Limitations of Current Approaches

Given the differences between traditional whole-file and partial content distribution, traditional techniques do not provide a good solution. In lower sharing environments peers at best download useless extra data; the smaller the fraction of data desired, the worse the overhead. At worst, they fail entirely: nodes are unable to retrieve desired data (see Section 7.10.3). Traditional distribution approaches are mostly orthogonal to this work.

Peer-to-peer networks provide the most common example of contemporary distributed systems where multiple peers cooperatively transfer data. Examples include BitTorrent [29], KaZaA [59], LionShare [74], or the Logistical Filesystem [10]; BitTorrent is the most popular. Peers in such systems can download *blocks* from multiple peers, eventually reconstructing a *whole file*. This differs from the byte range and partial file semantics we offer. Their metadata management features can not support partial content distribution– information on rare data does not "trickle through" the network, leading to partitions and inaccessible data. We show this experimentally in Section 7.10.3 (page 137).

BitTorrent [29], the most well-known and popular tool for large peer-to-peer downloads, is worth discussing further. BitTorrent supports only whole-file traditional content distribution. It uses a centralized "tracker" node to maintain global information on (1) participating peers and (2) a rough measure of their progress. When a new peer wishes to download a file, it asks the tracker for a list of existing peers who have or are downloading that file. Peers trade information on their progress pairwise, and each peer uses heuristics to weight block upload/download from others in parallel. BitTorrent does not support downloading only portions of a file. It is also slower and has higher latency than CEP. We discuss this further in Section 6.2.2 and the second half of Chapter 7.

Work on Distributed Hash Tables (DHTs) and overlay networks is also a popular research area. There is a large body of literature and implementations such as Pastry [23] or Chord [108] are available. These have been used as building blocks for peer-to-peer metadata management or file transfer mechanisms. But again, these systems support only block-based transfers and whole-file semantics. We discuss use of and problems with DHTs in more detail in later sections; e.g. 5.3.3 (page 90) and 7.5.3 (page 118).

Multicast trees/meshes [64,65], erasure coding [18,19,76], and network coding [73] techniques are also popular. They similarly support only block-based whole-file transfers. The whole-file assumption (in multicast trees, children totally share demand) and inter-block dependencies (in erasure codes, multi-block encoding), mean these can not be used in partial-sharing environments. We discuss this further in Section 4.4 (page 64).

Our final example comes from the high performance computing community, where data striping is commonly used to improve performance. Tools such as GridFTP [4] or RFT [92] and messaging systems such as MPI or PVM [6, 78, 109] allow peers to collectively transfer data. These systems require uniform/homogeneous nodes, are limited to simple data constraints (i.e. striping) and have limited fault tolerance. CEP supports a more general environment. We show this experimentally in Section 7.6.1 (page 120).

In summary, recent trends in hardware performance have created an opportunity for very high performance distributed transfers. Current block-based whole-file transfer approaches can neither provide the rich semantic features desired nor offer high performance in widely varied environments. Naive work-arounds applying existing technology– such as creating a separate CDN for each subset of desired data– perform poorly and have unacceptable overhead. This and other related work is covered in more detail in Chapter 9.

1.2 Thesis Statement and Contributions

Here we provide the thesis statement and a brief explanation. Chapter 2 discusses the problem in detail, as well as clearly defining the solution requirements.

1.2.1 Thesis Statement

Fully utilizing high speed links for large, complex transfers requires metadata management infrastructure and simultaneous transfers between multiple nodes. Composite endpoints using hybrid centralized/decentralized transfer scheduling, via graph-structured algorithms and feedback heuristics, provide a general, high-performance and robust approach.

Subsidiary theses required to substantiate this thesis include:

- ***Generality***: produce desirable results in a wide variety of environments and user constraints. This also requires:
- *high resolution*: nodes can transfer any arbitrary set of bytes, not necessarily in blocks, not necessarily a whole file, and
- *simplicity/flexibility*: good interfaces exist for describing user and system constraints.
- ***High performance***: converge to bandwidth and latency near hardware/data limitations. This also requires:
- *efficiency*: computation time amortized over all nodes commonly $O(n\ log\ n)$, worst-case $O(n^2)$, with query response in $O(1)$; and
- *scalability*: system works for transfers involving tens of thousands of nodes and 10Gbps+ links.
- ***Robustness***: failures which do not totally eliminate desired data are tolerated.

1.2.2 Dissertation Contributions

CEP, the Composite Endpoint Protocol, provides all these features. It composes up to tens of thousands of senders and receivers by way of a set of simple, flexible user APIs. We provide rigorous analysis of the problem and algorithms involved, as well as experiments to show functionality in a variety of environments and optimality under certain conditions. Our implementation is efficient, high speed, and tolerates server failures. It allows nodes of any speed with any subset of data to participate in the transfer. This dissertation provides the following contributions to the field:

- Rigorous mathematical definition of the transfer scheduling problem, also known as the the partial content distribution problem.

- In-depth analysis of the transfer scheduling problem, including complexity and various sub-problems, and development of a useful graph-structured canonical form.

- A set of new algorithms which (1) efficiently solve the transfer scheduling problem– commonly $O(n\ log\ n)$, worst-case $O(n^2)$, with query response in $O(1)$ and (2) produce high performance transfer schedules– typically equivalent to an optimal linear programming solution– that exploit heterogeneous node and network performance.

- A set of simple, flexible interfaces for integration with various types of applications, and which allow expression of arbitrary data layout and node constraints.

- Multiple implementations of CEP using different underlying network stacks and protocols, which have been used in real world situations.

- Side-by-side comparisons with other approaches showing the features and performance of CEP, improving bandwidth/latency by an order of magnitude as compared to BitTorrent: over 8$\times$ higher bandwidth for low-sharing configurations, 4$\times$ higher bandwidth for high-sharing configurations, and $\frac{1}{10}^{th}$ the latency for small transfers.

- Empirical evaluation showing the above features in a variety of environments: (1) functionality– composition of hundreds of real-world nodes and tens of thousands in simulation, (2) high performance– achieving over 30Gbps in the local area, 10Gbps in the WAN, and 1Tbps in simulation, and (3) fault tolerance– surviving failure of $\frac{1}{2}$ the servers with only a 2% decrease in performance.

1.3 Dissertation Outline

This chapter covered the motivating factors for the dissertation, the thesis statement, and our claims. This section gives an overview of the remainder of this document.

Chapter 2 discusses the problem in more detail. We walk through several increasingly complex use cases, outline the desired solutions, and more rigorously define the terms and sub-problems discussed at a high level in this chapter.

Chapter 3 provides analysis of the transfer scheduling problem. We cover reductions and simplifications to the general transfer scheduling problem which are worthwhile in real-world environments, conversion to the canonical graph form of the problem, and discuss which forms of the problem can be solved efficiently and which are NP-complete.

Chapter 4 discusses the main scheduling algorithms we develop in this work. While the prior chapter provided basic analysis and background material relevant to all environments, this chapter covers specific algorithms for generating transfer schedules. We cover three primary algorithms, extensions to them, and discuss their various performance characteristics. Together with Chapter 3, this addresses our claims of high resolution, efficiency, scalability, and high performance from a theoretical perspective.

Chapter 5 discusses the system design. It covers the “glue” which combines the scheduling algorithms, network communication, and other pieces into a cohesive whole. This includes the state machine for control of the protocol, several core algorithms not directly related to scheduling, and the various application programmer interfaces (APIs) used to interact with the system. This chapter addresses our claims of simplicity/flexibility and robustness.

Chapter 6 discusses the two main implementations of CEP and their variants. We include information on the network infrastructure, Unix sockets [106] and Globus XIO [5] stacks, and relevant engineering issues encountered in development.

Chapter 7 is our core empirical evaluation. In various real world, emulated, and simulated environments, we determine the performance of the system under the criteria discussed above. It covers the test configurations, and provides results supporting our claims of high performance, efficiency, scalability, and robustness from an experimental perspective.

Chapter 8 departs slightly from the rest of this document to look at a specific content distribution problem– traditional content distribution on hybrid satellite/terrestrial networks. We show that a simple transfer scheduling mechanism can provide excellent performance in this special case. This chapter speaks mainly to the generality of the transfer scheduling approach, but also shows efficient, scalable, robust and fair performance in this special-case environment.

Finally, Chapter 10 summarizes the claims and evidence in this dissertation, as well as providing some concluding remarks and discussion of future work.

Chapter 2: The Transfer Scheduling Problem

This chapter discusses the central problem of the dissertation: transfer scheduling. We show that the problem is interesting, difficult, and has valable open research questions. We provide an overview of the issues in many-to-many communication, make these issues concrete with several use cases, and then show how to effectively represent problem constraints. We conclude with a more rigorous definition of the underlying computational problem, which we solve in Chapters 3 and 4.

We will use the following terminology: a *node* refers to a physical machine, while a *peer* refers to the machine and any associated software. We typically have only one peer per node, so the two are roughly interchangeable. A *client* is a peer that wants data, while a *server* is a peer that provides data. A client peer may simultaneously be a server, or vice versa. A *scheduler* is a peer or set of peers which determines a transfer schedule for the global transfer. Typically the scheduler is also a *metadata server*, which collects and serves information on data location, host capacity, and system performance.

2.1 Overview

Transfer scheduling solves a generalization of the content distribution problem– what we call the partial content distribution problem. Given a logical set of data and some number of distributed peers, where each peer which may have and want different subsets of that data, how to best transfer data such that each peer gets what they desire?

This is *not* just whole file transfer to all participating peers. In contrast to traditional content distribution, each client may want *different* subsets of the data. Similarly, this is *not* just one-to-many multicast. Multiple servers may have replicas of different parts of the data. Finally, this is *not* just many-to-one download, as in GTP [126]. There can be many clients/servers acting in parallel. Peers may be both server and client concurrently.

The partial content distribution problem includes all of these earlier problems as special cases. Given the large class of problems this covers, we focus on the subset without known solutions: where peers require less than whole-file transfers and there are nontrivial mappings between clients and servers. However, the techniques we develop in this dissertation are sufficiently general so as to be useful for *all* distribution problems.
A good solution would include several features. We have five competing goals:

1. **High Performance** (i.e. high capacity): high aggregate system bandwidth and low response latency.

2. **High Efficiency** (i.e. low cost): do not waste system resources, exploit inexpensive hardware, be efficient.

3. **Simplicity** (i.e. low complexity): provide a straightforward API, support legacy code, minimize system “knobs.”

4. **Robustness** (i.e. tolerate crashes): survive faults, failures, errors, and inaccurate metadata.

5. **Generality** (i.e. be comprehensive): work on a variety of hardware/software platforms, with different input constraints and goals.

The first goal, performance, generally resolves to high bandwidth. While typically only *aggregate* bandwidth matters, we also try to keep latency low and ensure fairness when possible. High performance is difficult when available nodes may be relatively slow, heterogeneous, or have limited access to the data. There may many nodes, meaning that we must have good scalability properties– our target is on the order of 10,000 peers per logical transfer. We project that the largest “common” compute clusters will be around that size in the coming years; even today’s largest supercomputers only use a few thousand Input/Output (IO) nodes.

The second goal, efficiency, is minimizing resource expenditure. Many approaches can achieve high bandwidth but are very inefficient- e.g. flooding or whole-file transfers. We must minimize overhead from metadata transfer, computation time for scheduling algorithms, and ensure peers do not receive extraneous data.

The third goal, simplicity, is how to make a powerful system available without painful configuration or manual tuning. This resolves to providing the right Application Programmer Interface (API) and suitable tools for the target use model, and an internal transfer mechanism that transparently manages low-level details.

The fourth goal, robustness, is how to tolerate various failures. Aggregating devices causes capacity to grow linearly but failure probability to grow exponentially. In a distributed transfer, node or link failure should not cause the entire transfer to fail if another data replica is available. This is a particularly bad problem for high performance systems, which tend to use bleeding-edge hardware. As anecdotal evidence, it took over 30 runs of the LINPACK benchmark [67] for a successful result during the burn-in period for Los Alamos National Laboratory's ASCI Q supercomputer.

The final goal, generality, is simply that solutions should function in all cases. There is a unifying thread between the different use models– namely the idea of transfer scheduling– and we can exploit this to produce a more comprehensive approach than those currently taken.

2.2 Use Models

This section makes the issues described above concrete through illustrative examples. We go through several specific use cases and show how the problems encountered are interesting and not adequately solved by existing work. Previously, we defined the problem negatively– by explaining what it was not– while this section defines what it is.

Our first example, high speed cluster-to-cluster communication was the initial motivation for this work. Next we look at remote visualization; where data constraints may change and latency is more important. Finally we look at content distribution, and the problems traditional approaches have when applied in partial-sharing environments.

2.2.1 Cluster-to-Cluster Bulk Transfer

Consider a transfer between two homogeneous clusters of nodes. Each node has access to a local shared file system, and we wish to transfer a file from one cluster's file system to the other. The naive approach is a one-to-one transfer between a single server node and a single client cluster node. This will be limited by the capacity of a single node and have problems in the wide area due to TCP limitations (see Section 7.2.1).

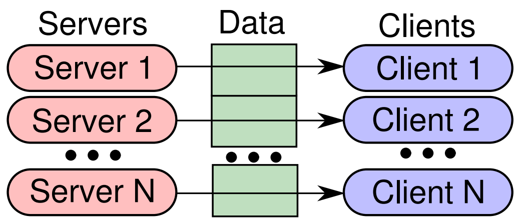


Figure 2.1: A Simple Striped Transfer

The next obvious approach is a striped transfer from several servers to several clients as in Figure 2.1. Each client/server pair transfers a disjoint piece of the file. This is useful to avoid TCP's problems in the wide area, to distribute data for analysis on a cluster of nodes, or to logically collect otherwise disparate transfers for management purposes.

In an ideal world, this would be enough; but we wish to solve a more realistic problem. Nodes may not be homogeneous; equally sharing work would limit performance to the speed of the slowest node. Clients may want arbitrary sections of the file; e.g. tiled simulations require some overlapping data at the edges. Servers may provide arbitrary sections of the file, due to caching, load balancing, etc. Finally, nodes may host both a client and server simultaneously, e.g. during a distributed computation. Figure 2.2 illustrates some of these features.

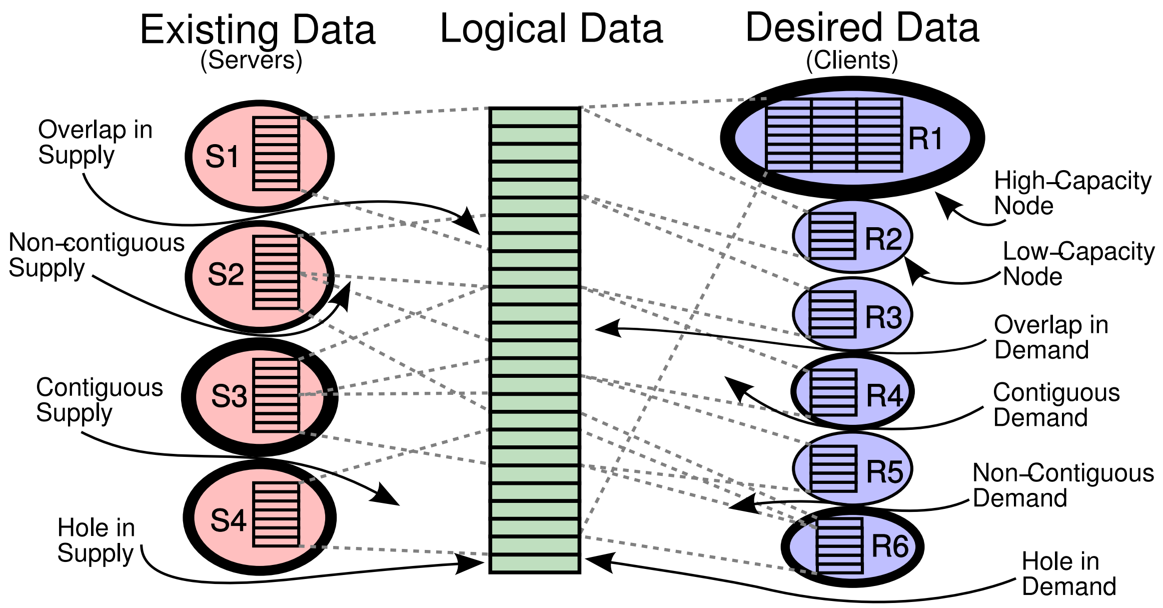


Figure 2.2: Sample Problem

Nodes S_1 to S_4 (the Servers) have access to overlapping subsets of a logical data collection, e.g. a file or database. Nodes R_1 to R_6 (the Receivers) want subsets of that data. Nodes have different capabilities, indicated by the thickness of their oval. The data logically exists in a linear name space.[1] Note the overlap in supply and demand for given ranges, and ranges which no node supplies or demands. These features distinguish our problem from the related $M \times N$ problem (a.k.a. the N-to-M problem, the M-by-N problem, and so forth) defined by Sussman [71] and others [36]: ours explicitly allows replication, holes, and failures, dynamic behavior, and nodes may be both client and server.

In the general case, clients must *locate* servers providing desired data, and *select* the best one(s) from which to download their data. The replica location and selection tasks provide new challenges and opportunities for optimization: peer-to-peer transfers, locality detection, load balancing, etc.

[1]This was a parallel creation of a linearization mechanism, discussed in depth in [125] by Sussman.

2.2.2 Remote Visualization

This use model considers visualizing a large data set, such as those found in research on physics, geology [24], climate change, or medical imagery [80]. Consider data currently on a storage cluster's distributed file system, to be analyzed on a second cluster, and the analysis rendered for display (perhaps on a third cluster). The user wants to interact with the visualization. While the prior use model was for latency-tolerant bulk data transfer, here response time and dynamic behavior are important.

The networking portion of this application is complicated. Frame rendering time varies depending on complexity and data location, so each rendering node will have a different amount of data belonging in differing logical locations in the video stream. Presentation on a tiled display provides no 1-to-1 mapping between display and rendering nodes.

Similarly, we may have source and destination replication at the file system or application level as the user steers back and forth through the data. There will be holes in supply and demand for the same reason, and it may be difficult to map a complex data set into a linear range without empty sections.

We may have heterogeneity. Nodes may have imperfect load balancing for rendering tasks or different performance depending on which distributed data is physically on their local disk. We may also have dynamic behavior. Transfer requirements will change as the user interacts with the system.

Work flow tools such as Kepler [7] are increasingly being used to capture the separate pieces in such a process. Some of these operations entail movement of massive amounts of data, and that is where we come in to the picture. For example, consider the simple work flow graph in Figure 2.3; our work (CEP) fits naturally into this structure.

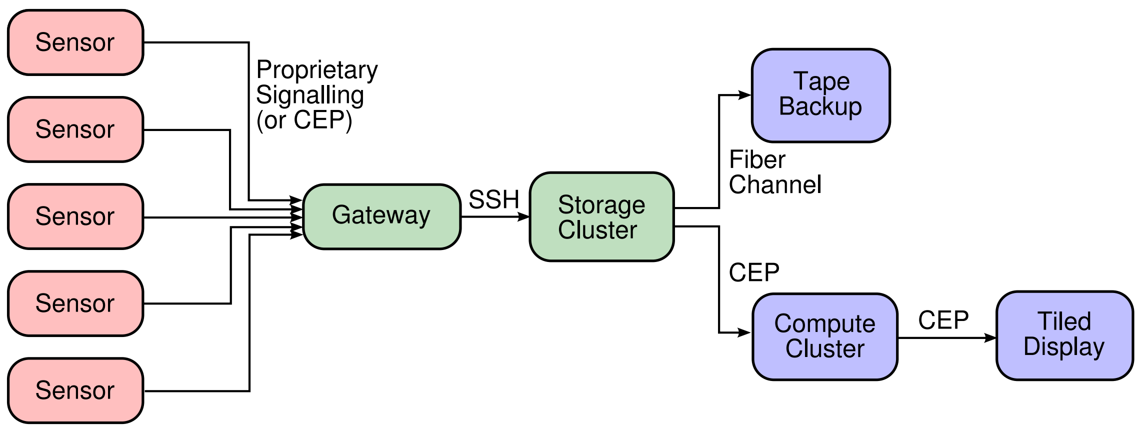


Figure 2.3: Sample Work Flow

2.2.3 Content Distribution

The distinction between traditional and partial content distribution is that the latter allows different users to request or provide different subsets of the data. For example, consider a small Internet TV station. Some users want all the shows, some only a favorite show, some want to skip commercials, some want only the commercials (e.g. the Superbowl), some want adult content included, some a version safe for children, and so forth. All users may want data at different bit rates or frame sizes.

As discussed in Chapter 1, traditional approaches using multicast, caching, peer-to-peer transfers, or erasure coding work poorly for partial content distribution. Enabling this use model would requires the server predict user interest, create and store multiple versions of the data, and support hundreds of transfers. Even then each transfer would be independent, unable to exploit data in others. We show this empirically in Chapter 7.

This brings us back to the concept of demand overlap, or demand sharing. Traditional content distribution is total sharing: all nodes want the same data. Partial content distribution sees this a spectrum of possibilities from no sharing (nodes want disjoint data), to low-sharing (nodes' data has minimal overlap), to high or total sharing. Section 2.3.1 provides an explicit definition of a sharing metric.

While this work focuses on high-speed wired networks, content distribution problems arise in nearly every network imaginable. As special-case extension, Chapter 8 considers traditional content distribution on a network whose nodes have satellite capabilities as well as physical Internet connections. Due to the magnitude of changes required to best support this environment, it is discussed separately from the main flow of this dissertation.

2.3 Problem Specification

This section covers the next obvious question: how to efficiently capture all the user's constraints, and in such a way that processing them is feasible? We do so by distinguishing between the physical and logical constraints. Physical constraints are the structure and characteristics of the network. Logical constraints are constraints on the data. Each set is input as a simple list of known information, as described below.

2.3.1 Logical Constraints: Data

We make no assumptions regarding data location, either the input (current) or output (desired) structure. Either may have arbitrary overlaps (replicated data areas), holes (data for which no sender/receiver exists) and be of any size. We assume no special data encoding (e.g. erasure coding).

We capture these constraints as two sets of contiguous byte ranges for each node. That is, each node provides a list of ⟨start byte, end byte⟩ tuples which is the data they require (are a client for) and provide (are a server for). This representation is inexpensive for most environments. The only inefficient common case is strided access over small data. This can be handled with a simple extension, ⟨ start byte, block size, stride length ⟩ tuples, but this is left to future work. Alternatively the user can remap their data.

Given the lack of assumptions on data layout, we can capture a wide variety of situations at high resolution: two of our goals. Performance will obviously vary depending on the constraints. The most important predictor of transfer behavior and appropriate technique is the level of sharing. We will talk about the correspondence more in Section 2.5 discussing the solution space, but for now we quantify the idea with the following metric:

$$Sharing = avg_{\forall i}(avg_{\forall j}((|\{d_i\} \cap \{d_j\}|)/|\{d_i\}|))$$

For each node n_i and reachable peer n_j, the overlapping proportion of data n_j has or wants that n_i wants, aggregated over all nodes. $\{b_i\}$ and $\{b_j\}$ are the set of data that nodes n_i and n_j have, and $||$ takes the size of the set. Effectively this is a measure of set similarity.

Put another way: For each node, calculate the average level of data shared with its peers; then average that across all nodes to get the sharing measure. When all nodes want the same data, this is 1. When all nodes want different data, it is 0. When half want one set of data and half another, it is 1/2. For the special case of block input, this metric can be calculated particularly efficiently by ORing block bitmaps.

2.3.2 Physical Constraints: Nodes and Networks

We make somewhat stronger assumptions about the networks used for CEP transfers: that they are IP-based and well connected. That is, an IP packet from any node can reach any other; firewall/NAT traversal is not supported. We assume all devices can run TCP and a minimal amount of user-level code. Weak or proprietary devices, such as sensors or microscopes, can participate through a gateway.

We assume nothing about the number or location of peers. We assume nothing about pairwise network properties (bandwidth, latency). We assume nothing about pairwise node properties, i.e. homogeneity. We will make stronger assumptions to perform specific experiments. Our motivating environment, for example, has high core bandwidth (10+ Gbps) and hundreds of endpoint nodes with Gigabit Ethernet.

Similar to the logical constraints, we capture network links as part of a tuple space: $\langle$ source node, destination node, bandwidth, latency$\rangle$. Unsurprisingly these tuple spaces can be transformed efficiently into a graph structure, as specified in our thesis statement. We do not currently use metadata such as a link's loss rate or other dynamic behavior as part of our scheduling algorithms. Input data should represent the current net goodput and be updated as behavior changes.

2.4 Rigorous Definition of Transfer Scheduling Problem

This section gives an abstract specification of the problem and our desired solution. First we discuss capturing an instance of the problem, then what a valid solution entails, and finally the objective function to maximize- the goal. The next chapter shows how to simplify the problem without loss of generality.

A **problem instance** is a complete specification of constraints as described above; A set of nodes, with the data each node has and needs, and a set of links, with the bandwidth and delay of each link. We use the terms *range* for an arbitrary set of integers specified as disjoint *segment*s $[b_i ... e_i]$ ($b$egin to end, inclusive). These ranges are the bytes that a given node stores or wants, in no particular order. Segments may overlap. These terms were chosen to bring to mind line segments and the range of a function.

```
Set of $Nodes$ { $Node_0$ ... $Node_n$ }
    $Node_i$ := {
        Range of data $required$ := { $\langle b_0, e_0\rangle$ ... $\langle b_r, e_r\rangle$ }
        Range of data $provided$ := { $\langle b_0, e_0\rangle$ ... $\langle b_p, e_p\rangle$ }
    }
Set of $Links$ { $Link_0$ ... $Link_m$ }
    $Link_i$ := $\langle Source, Destination, Bandwidth, Latency\rangle$
```

Listing 2.1: Problem Instance

An element of the **solution space** is a transmission mapping: a set of 6-tuples. This is just a list of which nodes communicate, what they send, and when they start and stop sending. Transfer rate is constant over that period.

$$\langle\ source\,,\ destination\,,\ \texttt{byte}\ begin\,,\ \texttt{byte}\ end\,,\ start\,,\ stop\ \rangle$$

To be a valid solution, the set must satisfy several **constraints**. Each tuple must be well-formed: receivers must get the data they have requested, servers may only provide data they have, and link capacities must not be violated. For clarity, we treat ranges as a simple sets – enumerating every byte in every segment in the range. Set notation simplifies the expression. See Listing 2.2. This specification assumes a single physical path between pairs of nodes and that routes do not change while the transfer is in progress.

```
# Constraint tuples must be well-formed
$\forall c \in T : c_{source} \in Nodes$ and $c_{destination} \in Nodes$ and
    $c_{start} \geq 0$ and $c_{stop} \geq 0$ and $c_{begin} \geq 0$ and $c_{end} \geq 0$

# Senders can send only data they have;
# originally or from another transfer
$\forall c \in T,\ \forall x \in \{c_{begin}, ..., c_{end}\} :\ x \in c_{source_{provided}}$ or
    $\exists c' \in T \mid c'_{beg} \leq x \leq c'_{end}$ and $c'_{dest} = c$ and
        $c'_{stop} + latency(route(c'_{source}, c'_{dest})) < c_{start}$

# Receivers can only receive data they want
$\forall c \in T,\ \forall x \in \{c_{begin}, ..., c_{end}\} :\ x \in c_{dest_{required}}$

# Receivers must get their data
$\forall N_i \in$ Nodes, $\forall x \in N_{i_{required}}$
    $\exists c \in T \mid c_{destination} = N_i$ and $c_{byte\_begin} \leq x \leq c_{byte\_end}$

# Can not send faster than link capacity.
$\forall$ times $t \in \{min_{\forall c \in T}(c_{start}) ... max_{\forall c \in T}(c_{stop})\}$
    $\forall l \in Links,\ \forall c \in T$
        $\Sigma(l_{stop} - l_{start})/(c_{end} - c_{begin} + 1) \leq l_{capacity}$ where
            $l \in route(c_{source}, c_{destination})$ and $l_{start} \leq t \leq l_{stop}$
```

Listing 2.2: Linear Program Problem Constraints

This specification explicitly allows peer-to-peer sharing in the second sender data constraint clause, which states that they may send bytes in a segment already received from another node. $latency(route())$ refers to the network delay sending from the source to the destination. It explicitly denies optimistic receiver operations– no downloading extra data.

The primary **objective** here is speed: minimize the time at which the last node finishes receiving data. We are interested in the total completion or termination time, not that of any individual node. Thus, we minimize the completion time of the *last* node.

Our evaluation uses other metrics to show system characteristics. With equivalent aggregate termination times, we wish to maximize fairness, streaming performance, or system utilization (depending on the enviroment). For fairness we use Jain's measure [55]. Streaming bandwidth measures the rate at which nodes can start utilizing data they receive: the largest run of data with no holes. Table 2.1 shows these objectives.

Table 2.1: CEP Primary and Secondary Objectives

Primary Objectives (equivalent)	
Termination time:	minimize $\forall c \in T : c_{stop}$+$latency(route(c_{source}, c_{dest}))$;
Bandwidth:	maximize $(\sum_{c \in T} data(c))/time$
Secondary Objectives (May have trade-offs)	
Fairness:	maximize $(\sum x_i)^2/(n \cdot \sum x_i^2)$
Link utilization:	maximize $capacity/bandwidth$
Streaming bandwidth:	maximize $(highest\ contiguous\ byte)/time$

2.5 Solution Space

While the prior section discussed the solution space from a theoretical point of view, this section gives a high-level overview of different ways one might solve the problem. Stepping back for a moment, we show how current research fits together.

All transfer schedulers fall into two rough categories; *explicit* schedulers plan an full transfer and subsequently implement it, while *implicit* schedulers use node/block selection heuristics, forwarding semantics, or special data encoding to get receivers their data. Most current approaches fall into the latter category; CEP is a hybrid mechanism.

Along the same lines, metadata transfer and scheduling can occur *locally* or *globally*.[2] This is a measure of coordination, but need not mean centraliziation; decentralized approaches often communicate heavily to maintain invariants such as DHT structure. Increased coordination enables partial-file sharing; without it nodes cannot efficiently locate a source for a desired piece of data.

Peer-to-peer systems typically have only local metadata and optimize locally with an implicit scheduler; global behavior is an emergent property derived from local actions. This approach makes sense for dynamic, heterogeneous, "black box" networks: long-term plans in such environments are inappropriate. In contrast, high performance and research networks [20, 31, 69, 117] tend to be less heterogeneous, less dynamic (some even include bandwidth reservation features) and their structure is often known. CEP can exploit these features to improve performance via explicit planning and global information.

Figure 2.4 gives a graphical view of some current approaches in terms of coordination and performance. The level of coordination is how much metadata is shared among peers: their local/global-ness. Most approaches here act as implicit schedulers; e.g. network coding can be applied to create explicit schedules but that is not its intended use.

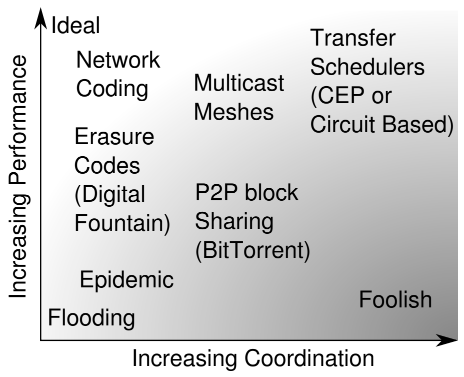

Ideal
Network Coding
Multicast Meshes
Transfer Schedulers (CEP or Circuit Based)
Erasure Codes (Digital Fountain)
P2P block Sharing (BitTorrent)
Epidemic
Flooding
Foolish
Increasing Performance
Increasing Coordination

Figure 2.4: Coordination & Performance

[2]While metadata propogation and scheduling are logically separate, it only makes sense that better-informed peers make the decisions. This is not a democracy.

2.6 Summary and Conclusion

This chapter has shown that the general transfer scheduling (or, equivalently, the partial content distribution) problem is interesting, difficult, and unsolved. We have given concrete examples, a rigorous problem definition, and a map of the potential solution space. This lays the groundwork for the analysis of Chapter 3 and algorithms developed in Chapter 4, which solve the problem from a theoretical perspective. Chapter 5 and 6 then take those algorithms to produce a real-world implementation with all the desired features and performance characteristics.

Acknowledgements: Material from this chapter, in part, appeared in “The Composite Endpoint Protocol (CEP): Scalable Endpoints for Terabit Flows,” Eric Weigle and Andrew A. Chien, Proceedings of IEEE Conference on Cluster Computing and the Grid (CCGRID), 2005. The dissertation author was the primary investigator and author of this paper.

Chapter 3: Analysis of Transfer Scheduling

This chapter focuses on analysis of the transfer scheduling problem from a theoretical perspective. It provides a more rigorous foundation for the core algorithms proposed in Chapter 4. Together, these chapters target our goals of efficient processing; similarly the quality of these algorithms are what produce a high-performance transfer. Subsequent chapters address design of systems using these algorithms and their implementation.

Here we first discuss the implications of the weakly structured input described in Chapter 2. We assumed no specific structure on input data constraints and nodes had no particular capabilities. As for the network, we assumed only that it was well-connected. Now we walk through several preprocessing steps showing how such input can be processed efficiently- in time commonly $O(n\ log\ n)$. These simplify the problem and add useful structure without loss of generality. This leads to construction of the desired canonical form: a segment graph structure, which supports queries in constant time. The transfer scheduling algorithms in Chapter 4 take this as input.

We conclude the chapter with a discussion of the run-time complexity of algorithms using such data and an example illustrating typical problem analysis/solution.

3.1 Input Specification

Our input specification allows applications to efficiently define transfer requirements with arbitrary constraints on data and network structure. In particular, applications specify byte ranges instead of fixed-sized blocks for transfer. This captures a more general class of problem, but to exploit such structure more complex algorithms are required. This section discusses the tradeoffs between ranges and blocks and how to perform efficient operations on ranges. This chapter uses such operations to process the unstructured input into more useful forms.

3.1.1 Data Ranges and Data Blocks

Essentially all current approaches use block-based instead of range-based mechanisms. They split data into equally sized chunks, typically some power of two, for scheduling and transfer; this simplifies their algorithms. Range-based mechanisms effectively turn a special case for block mechanisms– partial block transfers– into the common case. Conversely, blocks can be seen as a special-case optimization for range-based systems.

Ideally one system could provide the best features of both approaches: general but able to use special-case optimizations. To do so we need a way to determine when and how to apply block scheduling techniques in the context of a range-based system, or vice versa. This can be accomplished by a few simple heuristics. “When” is whenever there are large overlapping ranges or range endpoints naturally terminate at regularly-sized boundaries. “How” is just to select the correct block size.

First, we look for large overlapping ranges. These are high sharing areas; when they exist, using block-based algorithms is desirable. Detection of such overlap can be done by building a sorted list of $\langle$ byte range, count of instances $\rangle$ tuples. This requires one iteration over all byte ranges. Running this after the segmentation algorithm from Section 3.3 allows us to detect ranges which have significant overlap but are not identical. As a side benefit the relevant metadata will be fresh, local, and warm in the cache.

Next, we simply look for ranges in the list which have sufficiently high sharing and are sufficiently large. These two thresholds can be determined experimentally. Results from BitTorrent [29] have shown that sharing on the order of 20-30 peers is required for the algorithms to be most effective, with 30-50 preferred [30]. The overhead of these algorithms requires that the ranges be at least hundreds of megabytes in size.

Then we determine the best block size. For large overlapping ranges we can choose any block size that is small enough to transfer quickly but large enough to minimize costs of metadata management. Again, this is determined experimentally; see Section 7.8.

Finally, even if we found no large ranges, it may still be useful to use block based mechanism given a natural block structure to the data. Say, e.g., 90% of the ranges fall on a 16KB boundary. Then blocks of 16KB would be a rational choice. Detecting this can be done quickly (constant time to any fixed degree of certainty) by random sampling and modular arithmetic using the tuples list.

Together these allow us to determine when to apply block scheduling techniques, and what parameters to use with them. Unfortunately, this will not improve performance if we already have good range-based techniques. Worse, the main benefit of block-based transfers is simplicity, but applying these techniques does exactly the opposite– adding more complexity to a range-based system.

In practice we need only half this approach: determine a good block size as above, and set it as a maximum transfer size. The performance effects of using blocks resolves primarily to (1) dynamic per-block replica selection and (2) data replication growing quickly as blocks propagate through the network. A maximum transfer size set to the correct block size achieves these benefits.

3.1.2 Efficient Range Matching

Range matching is the first step to transfer scheduling: before being able to optimize, one must be able to find *some* server for the desired data. This means finding ranges that overlap a search range. At its core, any scheduling algorithm must be able to do this efficiently.

Our problem statement assumed nothing about the structure of input ranges, e.g. they need not be disjoint or in sorted order. If we need stronger semantics, we must build any structure required. One useful tool is the “Interval Tree” data structure described in Section 14.3 of [32].

By augmenting nodes in a red-black tree with a “max value” field, an interval tree supports range insertion, deletion, and search for an overlapping range in time $O(log\ n)$. If we wish to find *all* overlapping ranges it takes time $O(min(n, k\ log\ n))$, where k is the number of overlapping ranges output.

Unfortunately, the worst-case search performance is $O(n)$, which is unacceptable. This occurs when a request matches every node in a tree– i.e. in a high-sharing environment where many different peers serve the same byte range. Finding the best peer to serve a request becomes computationally more difficult when there are more options.

We would like a stronger bound: logarithmic time range matching. By using the techniques described in the following sections, we can achieve this: first, sort and make ranges disjoint (time $O(n\ log\ n)$). Then insert them into an interval tree (time $O(n\ log\ n)$) where each node is augmented with a priority queue/heap. If the range already exists in the tree, push the peer onto the priority queue (time $O(log\ n)$). The priority queue is keyed by e.g. available peepeerpacity. In this way, peer lookup takes only logarithmic time: look up the node in $O(log\ n)$ time, then pop off the queue in $O(log\ n)$ time.

There are two problems with this approach. First, the “best” result depends upon the client (querying) node- which this approach does not take into account. Second, peers are placed in multiple priority queues. Changing data in one queue corrupts the heap structure in others. In practice we pay a higher up-front cost (worst case $O(n^2)$) to convert to a graph-structured canonical form (Section 3.3), which then allows a lower cost query ($O(1)$).

3.2 Preprocessing: Simplifications and Reductions

This section discusses a series of transformations to the transfer scheduling problem that move it towards the canonical form. All are made without loss of generality unless otherwise stated. These transformations let us avoid strong assumptions on input: whatever it may be, we can quickly transform it into the form desired.

3.2.1 Disjoint, Sorted Ranges

Converting arbitrary input ranges to a set of disjoint ranges sorted in increasing order is straightforward. We only need to sort the list and combine overlapping ranges. That is, we convert lists like {[3,5], [7,10], [2,4]} to just {[2,5], [7,10]}. This conversion takes time $O(n\ log\ n)$ using the following algorithm.

```
Given an arbitrary range in an appropriate encoding:
    Sort in increasing order by begin value,
        then by end value # to break ties.
    Iterate through each range $S_i$:
        If ($S_{i+1}(begin) \leq S_i(end)$): # There is overlap
            Remove both ranges.
            Insert combined range [$S_i(begin), max(S_i(end), S_{i+1}(end))$]
            Reconsider range $S_i$ # May need to merge again
```

Listing 3.1: Creating Disjoint, Sorted Ranges

First we sort using any of the well-known $O(n\ log\ n)$ sorting algorithms. Then we walk through the list, an $O(n)$ operation, checking to see if adjacent entries need to be merged. We know that ranges which overlap or touch must be adjacent due to the sort. Together these are at most an $O(n\ log\ n)$ operation.

This naive algorithm can be improved given knowledge of the structure of the data. In particular, Section 3.1.2 shows how red-black trees can be used for efficient range matching. Therefore another approach is to simply insert all values into such a red-black tree, and check for overlap during the insert procedure. If there is overlap simply remove the existing node and insert a merged value instead.

Finally, an important subset of the problems we encounter have predefined structure, such as blocks. Section 3.1.1 showed how this can be detected with high probability in constant time. In such cases, we can use a hash-based bucket sorting algorithm to perform this conversion in linear time.

3.2.2 One-Segment Ranges

The original problem description allows peers to have an arbitrary range of bytes, not necessarily contiguous, but this does not actually increase the generality of the problem. If instead peers were treated as having only a single contiguous data range it simplifies the problem statement greatly. Consider the following transformation algorithm:

```
For all nodes # all senders and receivers
    If the node has more than one segment in their range:
        # Sum the total bytes in the entire range:
        Let $\sigma$ = $\sum_{\langle b_i,e_i\rangle\in R}(e_i - b_i + 1)$
        For each segment $\langle b_i, e_i\rangle$:
            Add a node:
                Speed $r \cdot \frac{e_i-b_i}{\sigma}$
                Edge to only that segment
        Remove the original node
```

Listing 3.2: One-Segment-Per-Peer Transformation

A solution to this new problem is a solution to the original problem; satisfying each "sub-receiver" means the original receiver gets its data. We added no new data to the range, so it receives no extra data. Dividing the node's speed among "sub-receivers" means we can not exceed the original speed. Nothing else changed; thus a solution to this new problem is a solution to the original.

As all data must be received before a node is considered "done," the best way to split the network capability of a node is proportional to the size of data transferred. If we were to split in a non-proportional way, some segment(s) would complete sooner but others would complete later. In other words, the total completion time for any non-proportional allocation is at least as large as for the proportional one, so the proportional one provides the optimal solution for the simplified problem. Thus an optimal solution here is an optimal solution for the original problem.

This transformation assumes independence in the transfer of each segment, which is not quite true. Multiple transfers between a pair of peers need not go over separate connections; re-using a single socket improves performance by avoiding TCP handshaking and slow start behavior. In the same way, this creates hidden performance and failure correlations among these newly-created nodes (e.g. a bottleneck link or competing load).

Luckily, we can both use this simplification and maintain the correct structure. By representing this using weighted *edges* in a graph rather than new *nodes*, we capture the right semantics. This is shown in Section 3.3.

3.2.3 Transfers Starting at Time Zero

The last transformation normalizes over time. The original problem description allows for a solution where transfers start and complete at any time. However, this gains us nothing in generality. Network speeds are almost infinitely divisible, and it is almost always better to immediately request data rather than waiting [48]. In other words, implicit space-sharing of a link is just as good as explicit time-sharing the link.

Therefore, we remove the start/stop times as part of the criteria for a solution. In its place we assume that a node transmits/receives data at rates proportional to the size of data sent to/from each peer. Put another way, a node divides available bandwidth among flows such that their expected termination times are all equal. When otherwise limited, it simply sends as fast as possible; the excess bandwidth may be allocated to other flows. This does not change total aggregate completion time– which is always limited by the slowest node. Rate mismatches are quickly detected in practice and a scheduler can address it by shifting load between replicas.

A solution to this new problem will be a solution to the original problem. This transformation does not change the set of data transferred, only when it is done. Therefore we can not violate any of the original data or network constraints.

An optimal solution to this problem will be at least as good as an optimal solution to the original problem. Our new constraint ensures that each node either (a) fully utilizes its link for the entire transmission, or (b) one of its peers is a bottleneck for some data.

In case (a), we have 100% link utilization so there is no way any solution to the original problem could do better. Similarly in case (b), we are transferring as fast as possible to/from the bottleneck node. The only possible reason why it is not handling data at the speed we wish it to is when that node handles a larger amount of data proportional to its speed. In other words, there is no way to improve the overall completion time because that peer node is in case (a), perhaps due to problems in the network.

Counterintuitively, local reallocation of “excess” bandwidth to other flows when some peer is bottlenecked will not necessarily improve overall performance. This is because that “excess” is due to another peer not using its full share of bandwidth– i.e. going slower than expected. Local reallocation of flow will not improve that peer’s performance. It is *other* nodes from which that peer can fetch data that need to do the reallocation. We locally experience their bottleneck, but they need to schedule around it. This second-order dependency is one reason the transfer scheduling problem is complex.

Again, there are some potential problems with this transformation in practice. First, it relies on adequate congestion control at the network level. Using stock TCP as a carrier, this is only true in the long term, without link errors or round-trip-time differences across competing flows. Second, it assumes the peer OS can avoid thrashing. Starting all flows concurrently may lead to high load and poor locality of reference. Fortunately, our experiments in Chapter 7 show no noticeable problems degradation from such problems.

3.3 Conversion to Canonical Form

This section describes the segment graph, our canonical data format, and input for the algorithms in Chapter 4. A segment graph is a set of vertices and directed edges. Vertices represent segments or nodes, with any associated metadata (e.g. an filename). Edges link *from* segments to nodes that *need* that data (clients), or *to* segments from nodes that *have* that data (servers). This captures the input data constraints; network constraints can be captured in the same way, with edges mirroring physical topology and capacity.

This structure is sufficiently general to allow many different types of processing. The simplest transfer scheduler consists of receivers picking a random linked sender for each segment they need; as a first approximation this how schemes like BitTorrent work.

To construct this graph we turn once again to an augmented binary search tree whose entries represent segments. We maintain a separate list of peers. Segments are doubly linked to senders providing them and to receivers requiring them.

The structure is filled with a sorted list of known segments as follows: for each peer, add their requested or supplied segments individually to the tree. If the segment totally overlaps a pre-existing segment, link the node to the existing segment. If it does not overlap, add the segment and then link. If it partially overlaps, split the segments to create total a region of total overlap and a disjoint region, then recur on the pieces. Listing 3.3 (page 35) captures this process, where `SegmentRanges` is the main function.

When `SegmentRanges` is complete, each node has two sets of links: one for the ‘have’ relationship and one for the ‘need’ relationship. Peers link to segments they have or need, segments link back to nodes that have or need them. All segments are disjoint.

The input and output structures are equivalent: walking a node’s links in the output and applying the merge algorithm from Section 3.2.1 reproduces exactly the original input. Any solution over this structure can likewise be translated to the prior format in time $O(nk\ log\ k)$ where n is the number of nodes and k is the number of edges per node.

```
SegmentRanges(Senders, Receivers):
    For each node $n$ in Senders $\cup$ Receivers:
        Let $b, e$ be $n$'s beginning and ending bytes.
        AddRange($b, e, n$)
    Walk the segment structure; for each edge:
        Backlink node to segment and tag sender/receiver

Split(Range $r$, p):
    Create a new range $s$:
        $s_{begin}$=$p$
        $s_{end}$=$r_{end}$
        $s_{nodes} \leftarrow r_{nodes}$
        $r_{end}$=$p-1$.
    Insert $s$ into range structure.

AddRange($b, e, n$):
    Find $b$ in the range structure.
    If $b$ falls outside any segment:
        Let $s$ be the next segment to start.
            If no segment is next, let $s_{begin} = \infty$.
        Add $b$ as the start of a new segment, $r$.
        $r_{nodes} \leftarrow n$
        If $e < s_{begin}$: $r_{end}$ = $e$
        Else: $r_{end} = s_{begin}$; AddRange($r_{end}$, $e$, $n$)
    If $b$ is the start of some segment $r$
        If $e < r_{end}$: Split($r$, e$-1$); AddRange($b, e, n$)
        Else If $e$=$r_{end}$: $r_{nodes} \leftarrow (r_{nodes} \cup n)$; return;
        Else: AddRange($b$, $r_{end}$, $n$); AddRange($r_{end}$, $e$, $n$)
    If $b$ falls inside some segment $r$:
        Split($r$, $b$)
        AddRange($b, e, n$)
```

Listing 3.3: Segment Graph Conversion Pseudocode

The problem is that the `SegmentRanges` algorithm has worst-case running time of $O(n^2)$ where n is the number of nodes (or equivalently the number of segments, given the one-segment-per-node transformation). Each time we add a segment, we induce at most two new splits, one for the begin byte and one for the end byte, amortized over the entire run of the algorithm. That is, given n nodes with a begin and an end byte, we have at most $2n$ points on a line, which creates at most $2n - 1$ segments.

The creation of each segment requires finding the begin/end among pre-existing segments; these lookups take time $O(log\ n)$. Each of these segments will link to at most $2n$ sender/receiver nodes. This term dominates, so the total linking time will be $O(n)$.

Together, this gives a total time for the creation of these segments and mapping to their respective sender/receivers of $O(n^2)$. This upper bound is tight, as shown by the following worst-case example; the simple input specification creates many graph edges.

Consider n nodes in two classes. The first class, nodes $i = 1...\frac{n}{2}$ have range $[3i, 3i+1]$. The second class, nodes $(\frac{n}{2}+1)...n$ have range $[0, 2n]$. All nodes in the first class create a disjoint segment all their own, that is, $\frac{n}{2}$ segments. All nodes in the second class force the creation of a link for all nodes in the first class and all spaces between them. That is, at least n links per range.

Thus, we are forced to create $\frac{n}{2} \cdot n$ links, giving a lower bound of $\Omega(n^2)$. Enumerating them in this algorithm will take time $\Theta(n^2)$. Figure 3.1 illustrates this for $n = 8$; note the rats-nest of links the second class of nodes must build.

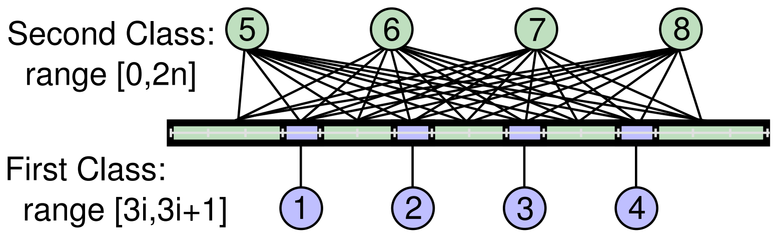


Figure 3.1: Worst-Case Input to Segment Ranges

That said, the worst-case time is certainly not the average-case time. Most real-world uses of this algorithm run in time $O(n)$. Such cases include total data replication (one segment, n links), data striping across nodes (n segments, one link each), or any fixed combination of the two (n/k segments, k links each).

Similarly, this worst case only occurs with unboundedly large segments. Given an upper bound on the length of a linked segment, the algorithm runs in linear time. Given a lower bound on the minimal segment split, we also get linear time (both with large constant factor). In the limit, the latter case is equivalent to a block-based scheme.

Finally, should the cost of creating this structure be too high for large problem sizes, an useful related structure can be created in time $O(n\ log\ n)$. By extending the graph with virtual internal nodes, forming a mesh, we eliminate the need for so many edges. In the example above, we need one virtual node linking to all segments, and all the second class nodes link to it. This reduces the number of edges by a factor of n. The tradeoff is that processing on the resulting structure becomes more complicated and expensive. We explore these ideas further in our discussion of network flow algorithms in the next chapter.

In terms of optimality, building this structure has neither added nor removed constraints from the problem. Thus an optimal solution here will be an optimal solution to the original statement.

3.4 Problem Complexity

To this point we have given several relatively inexpensive polynomial-time algorithms to permute the problem into more useful forms. None have actually attempted to produce an optimal solution to the transfer scheduling problem. The question at this point is: how efficient can such solution algorithms be? It turns out that the canonical specification from Section 3.3 is sufficiently general that we can reduce from an NP complete problem.

In the real world, however, most input will not contain the special structures required to make the problem NP complete. A large class of transfer scheduling problems can be solved optimally in polynomial time and many can be solved in linear time. So while the worst case may be very bad, the average case (as defined by our use models from Chapter 2) is not nearly so difficult. This of course depends entirely on the network and data constraints given by the user application.

The rest of this section discusses the reduction from an NP complete problem, approximation algorithms, and the cost of distributed scheduling. Chapter 7 returns to this topic empirically, evaluating the runtime of different algorithms on common input.

3.4.1 NP-Completeness

Transfer scheduling can be seen as a generalization of the “file transfer problem” described by E.G. Coffman, Jr. In [57], Coffman shows the assumptions under which the file transfer problem is NP-complete. As transfer scheduling problems can encapsulate such file transfer problems; it follows that transfer scheduling is also NP-complete.

To see this directly, consider a reduction from the edge coloring problem [53]. Given a graph $G = \{V, E\}$ to be colored, we construct a constraint graph as follows:

First construct the network constraints: a star-topology network with a central ‘switch’ node and $|V|$ nodes connected to the switch with equal-capacity links. Then construct data constraints corresponding to the edges in the original graph: i.e. if there is an edge between node n_i and node n_j in G, add a 1-byte data dependency between n_i and n_j. Ensure all segments are disjoint by incrementing a counter for each dependency.

Consider an optimal solution to this transfer scheduling problem. It determines when each data dependency is sent and all data is sent in minimal time. The time at which a data dependency is sent is the color of the corresponding edge. It is a proper coloring because nodes can not send two bytes simultaneously. It is a minimal coloring because the transfer could not complete earlier, i.e. using fewer colors.

Thus any algorithm which can solve all transfer scheduling problems can be used to solve the edge coloring problem, and through it the rest of the problems in NP. Even restricting the structure of the constraint graph to a bipartite graph– as when senders are just senders and receivers are just receivers, or a tree– as with "traditional" content distribution– the problem is still NP-complete. See Table II in [57] and their discussion of "Arbitrary Ports" for further information.

Finally, the full transfer scheduling problem is even less straightforward, though not necessarily more computationally complex. This example, though NP complete, effectively ignores network structure. In other cases, one must account for that structure.

3.4.2 Approximation and Asymptotic Bounds

The above analysis specifically utilized one-byte data constraints because achieving the NP-completeness result required that transfers be indivisible and serial. In practice, flows are larger and can be subdivided and parallelized in nearly arbitrary ways. That was the point of Section 3.2.3; starting transfers at time zero makes them all parallel. This section walks through an example which qualitatively illustrates the differences between the NP-complete cases and more easily solvable ones.

Consider two source nodes of speed s which each have the same k segments, each segment may have a different size, and one receiver of speed $2s$ wants all segments. A solution here is the transfer schedule which minimizes the total transfer time. If transfer of a segment were indivisible, this is precisely equivalent to the minimum multiprocessor scheduling problem– which is known to be NP-complete (SS8 in Garey and Johnson [46]).

However, since segments are divisible, we can simply request half of each segment from each node in parallel. This will provide a solution optimal to within a factor of 1.5: odd-sized transfers cannot be split equally. For larger transfers, the "odd" overhead can be amortized over the length of the transfer. We can get within any fixed $\epsilon > 0$ of an optimal solution as the transfer length increases.

In general, most problems we expect to find in the real world have this property: for all practical purposes they can be solved nearly optimally in polynomial time. Such problems include those with large divisible data and no network limitations (or at most a few bottlenecks), or networks with high sharing. In the first case, the ability to split large transfers into arbitrarily sized, independently scheduled portions is enough to approach an optimal solution. In the second case, data can merely be retrieved from the most local, least loaded node, regardless of wider-area network structure.

In contrast, other types of problems are fundamentally more complex. This is not simply the negation of the class above– that is necessary but not sufficient to make the problem "hard." To pose difficulty, the problem must include nontrivial bottlenecked networks and either small, indivisible data or data with low sharing. In the first case, cannot spread load by subdividing data transfers. In the second case, we are limited to a given set of source/destination pairs. In both cases, nontrivial networks provide a large set of possible transfer routes implying that an exhaustive search may be unavoidable. Lastly, time-dependent network or data constraints can also be difficult to handle, even if known ahead of time, as it dramatically increases the set of possible schedules.

In such "hard" cases, the polynomial-time algorithms we discuss in the next chapter will still provide *some* schedule. However, depending on the algorithm, the initial output may be far from optimal. Feedback and heuristics are used to correct for this over time. These constraints are rare in practice- our target environments fall in the class of polynomial time-solvable problems. Similarly, most applications have either high sharing properties (such as CDNs or peer-to-peer file-sharing) or simple block-based data constraints. In both cases, we can efficiently solve them to produce high quality schedules.

3.4.3 Centralized and Decentralized Scheduling

Prior discussion has implicitly assumed that the scheduling data exists in a single location for an algorithm to process. In a distributed system that may not be the case. Then either an extra data-retrieval term must be included in our analysis, or we must divide the processing cost over multiple nodes.

First consider a centralized node scheduling using distributed data. In this case, the data access cost rises from $O(1)$ to $O(log\ n)$ where n is the number of nodes in the system. This assumes a DHT such as Chord [108] were storing the information. The naive approach of serially pre-fetching all data takes time $O(n\ log\ n)$ assuming there is a roughly constant amount of data per node and bounded maximum network latency. While this may be an expensive operation, it does not affect the order of the run time for these algorithms.

Second, consider decentralized scheduling over distributed data. In this case, assuming appropriate clustering of data in the DHT, many operations can be performed solely on local data. Then the extra $O(log\ n)$ term is effectively eliminated giving performance comparable to a centralized scheduler. Unfortunately maintaining the same transfer semantics in a distributed environment is difficult; Sections 5.3.2-5.3.3 describe the issues involved in distributing scheduling in more depth.

In the end, centralization versus distribution is a design choice; both have their drawbacks. Centralized systems have to capture global state and may not scale, while decentralized systems have higher overhead and difficulty optimizing globally. We focus on hybrid algorithms with both a centralized part and distributed part, based on our assumptions about the target environment (Chapter 2). The algorithms in Chapter 4 seek a balance.

3.5 Sample Analysis

While Figure 2.2 showed an example problem using intuitive pictures, this section shows one more rigorously. This example ignores network constraints, which are already in a graph structure and hence need no preprocessing. “Speed” here is the NIC speed of the host, which provides a simple upper bound on its transfer rate. Data units are irrelevant here; one may think of ranges as being over gigabyte blocks.

Input (Fig. 3.2, left) is specified by the nodes themselves using the specification defined in Chapter 2. Applying the transformations given in this chapter produces the canonical segment graph (Fig 3.2, right). Note that we can immediately detect when the problem has no solution. In this case receiver 1 can not be satisfied: no peer provides segment 8-9. This is a simple $O(n)$ time test that our implementations roll into the conversion algorithm directly.

Solution: So a solution exists, we add an edge from node S2 to segment 8-9. Figure 3.3 shows an optimal graph solution (right) and two equivalent explicit schedules (left). Blocks 1-3 and 14-16 are the bottleneck here; they determine the overall termination time. Thus, there is flexibility in setting the transfer speed of the remaining nodes without changing the overall termination time. Excess bandwidth there can be allocated to other flows.

In practice, desired rates can not be enforced while using TCP. They are at best roughly followed as short flows complete and larger flows grow to fill remaining capacity. Underutilized network links should be filled by the remaining flows.

Node	Speed	Data
S1	100	1-6
S2	1000	4-7,10-13
S3	100	11-16
R1	2000	1-16
R2	100	1-3
R3	100	4-6
R4	100	11-13
R5	100	14-16

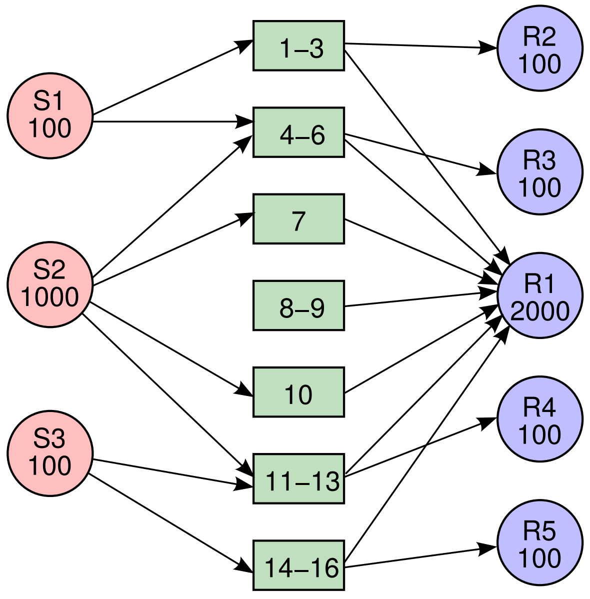


Figure 3.2: Explicit and Canonical Graph **Input** to Transfer Scheduler

Transfer Nodes	Data Range	Solutions	
		Rate	Rate
S1→R1	1-3	50	50
S1→R2	1-3	50	50
S2→R3	4-6	69	50
S2→R1	4-6	259	50
S2→R1	7	86	17
S2→R1	8-9	172	33
S2→R1	10	86	17
S2→R1	11-13	259	50
S2→R4	11-13	69	50
S3→R1	14-16	50	50
S3→R5	14-16	50	50

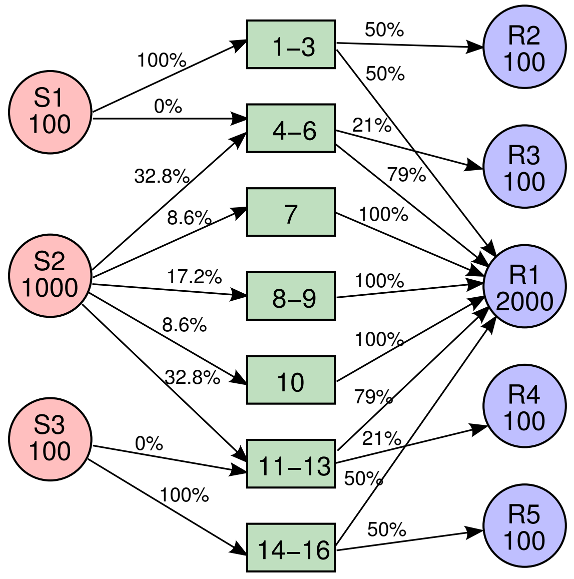


Figure 3.3: Explicit and Canonical Graph **Output** from Transfer Scheduler

3.6 Summary and Conclusion

This chapter has shown how one can efficiently, commonly $O(n \ log \ n)$ time, convert from an arbitrary transfer scheduling problem into a structured specification. This specification allows fast insertion, deletion, and search of ranges for the purposes of computing transfer schedules. In the process, we have shown in a more rigorous fashion the complexity of the problem and sub-classes which can and cannot be solved efficiently.

Returning to our claims, this chapter has shown that a graph-structured representation provides a straightforward, efficient, general problem encapsulation. Chapter 4 shows how to produce transfer schedules from this representation, enabling high-performance distributed data transfer.

Acknowledgements: Material from this chapter, in part, appeared in “Partial Content Distribution on High Performance Networks,” Eric Weigle and Andrew A. Chien, Proceedings of the IEEE International Symposium on High-Performance Distributed Computing (HPDC), 2007. The dissertation author was the primary investigator and author of this paper.

Chapter 4: Scheduling Algorithms

This chapter discusses several scheduling algorithms, building off the tools introduced in Chapter 3. These algorithms form the core of a transfer scheduler; using their output we support high performance distributed data transfer.

Our approach focuses on optimizing for data constraints. This works under the assumption that either core network bandwidth is not a bottleneck or that there is no useful information on network structure. In either case, a reasonable plan is to optimize for data constraints and use heuristics to dynamically correct for network behavior.

All algorithms require information on data constraints, which peers want/provide which data, and network constraints, the bandwidth and latency of links. At this point we assume it is provided in the canonical segment-graph form. Typically, the more information the better the output. Chapter 5 addresses the issues surrounding metadata collection.

This chapter first covers algorithms based on the ideas of network flow, a natural approach for transfer scheduling; then linear programming, to provide a known optimal baseline; finally greedy hill-climbing, a technique to improve algorithm runtime. Then we discuss a few powerful optimizations and how theoretical and practical results differ in common environments. We conclude with a high-level comparison of all techniques.

4.1 Network Flow Algorithm

As a network transmission problem, a natural approach is to analyze the problem using Flow Networks [1, 32, 38, 42]. This section first shows how to convert the graph-structured form of the problem developed in Chapter 2 into a flow network. Then we show how to use that to calculate a transfer schedule. Finally we discuss the quality of the solutions produced by this approach– unfortunately they are not optimal for general graphs.

4.1.1 Conversion to a Flow Network

Converting the canonical segment-graph form of the problem into a flow network is done using the following mapping. First, **nodes**:

- The senders S_i become the sources, and the receivers R_j become the sinks.
- The data segments become nodes in the center of the flow network.
- We must create a new super-source and super-sink, but they map from nothing.

Then we convert the **edges**:

- Each sender is fed from the super-source by a link with their original speed.
- Each receiver can sink to the super-sink by a link with their original speed.
- Infinite capacity between each sender S_i and all data nodes which S_i provides.
- Infinite capacity between data nodes and receivers R_j which R_j requires.

For example, Figure 4.1 gives one graph and the conversion to a flow network. Note that this conversion ignores network constraints other than the upper-bound NIC speed for each peer. These can either be handled by heuristics during the transfer, or by using a second flow network in parallel. That is, create two flow networks: one for the data constraints and another for the network constraints. Solve them together; on each iteration attempting to push additional flow through the data constraint graph, check and update spare capacity on the network constraint graph.

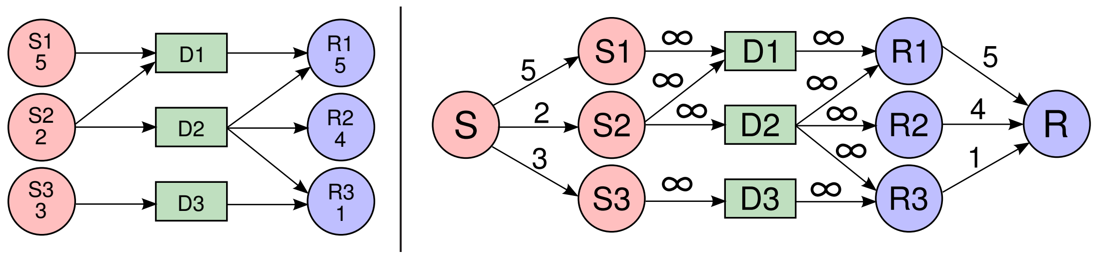


Figure 4.1: Original Problem Specification and Conversion to a Flow Network

Observe that some nodes such as S_3 are superfluous, and can be removed via an optimization step. In general we can remove any node with only a single associated segment, replacing it with a direct link with appropriate capacity. For sparse graphs with little sharing this optimization may remove nearly all the nodes.

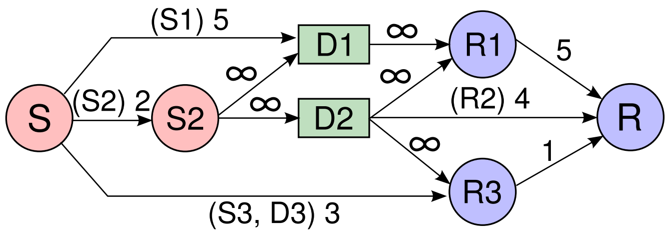


Figure 4.2: Optimized Flow Network Graph

Figure 4.2 shows the results of this (worst-case time $O(m)$) optimization. We indicate the nodes removed in parentheses; this metadata allows us to convert back to a transfer schedule after solving on this graph. This optimization reduces the run-time of the solution algorithms described in the next section.

Lastly, note that the scale for bandwidth values here and in other algorithms does not matter. Bandwidth input is effectively unitless; equally scaling input produces equivalently scaled output. Only the ratio between values is significant.

4.1.2 Solution Using Network Flow Algorithms

There are many fast algorithms for solving network flow problems, which is what originally motivated this approach. We can apply any of them to find the maximal instantaneous transmission rate for the network; for example the original Ford-Fulkerson algorithm, which runs in time $O(m|f|)$, or the Edmonds-Karp algorithm, which runs in time $O(nm^2)$. Here n is the number of vertices, m is the number of edges, and $|f|$ is the size of the maximum flow in the graph.

The main difference between transfer scheduling and network flow is the abstraction of time. Network flow problems ignore time, attempting to maximize instantaneous flow– put another way, they assume that the problem is solved once and the solution runs forever. Transfer scheduling does not ignore time– in fact we explicitly try to minimize global termination time. Therefore, straightforward application of network flow algorithms will fail. A max-flow solution may allocate all flow through high-capacity nodes, starving others. This violates the problem's data constraints. Here we address it by iteration (Section 4.2.1 gives another approach): solve the initial instance and as peers complete recur using the new constraints. Rather than simply starting over between iterations, we re-use prior state to improve the runtime of the algorithm. See Listing 4.1 for pseudocode.

Runtime depends on the max flow algorithm selected and the input problem– its constraints. The Edmonds-Karp algorithm is good for sparse graphs, which represent transfers with low sharing. Regardless of the algorithm selected and graph input, this approach terminates in worst-case time strongly polynomial in the number of data constraints. The iteration's inner loop requires one $O(log\, n)$ heap operation, then a few constant-time graph operations. The final optimization step in the worst case takes the same time as a full run of the max-flow algorithm. Together this requires running a strongly polynomial time algorithm up to n times, giving another strongly polynomial time algorithm.

```
Calculate Max flow.
For each link to a receiver:
   Calculate when that link becomes idle.
   This is the time the receiver has that segment;
      Just the size of the segment / current rate.
   Insert that (time, node) into a heap.
While (receivers exist):
   Remove the minimal time from the heap.
   Delete that link.
   If the node has no ingress, delete that receiver.
   Push-back that flow
   Creates at most one adjunct path, re-optimize.
      As we re-optimize, update the elements in the heap.
```

Listing 4.1: Iterative Max-Flow Algorithm

The point of the receiver heap is to improve the iteration's common case from polynomial to linear time. Most nodes are already optimized from the prior iteration, and via the heap we know exactly where spare capacity exists– paths to the node that just finished. Commonly reallocating that flow takes one pass over the edges via e.g. depth-first search. This gives time linear in the number of edges m. Worst-case behavior occurs in dense graphs, when nearly all nodes have dependencies to the node that just finished; then flow must be recalculated for all nodes, and multiple passes over all edges.

4.1.3 Network Flow Optimality and Approximation

Network flow works works well for two classes of problems. First is on networks with relatively homogeneous performance; meaning the flow network will tend to weight all edges rather than one 'fast' edge. This will in turn satisfy all nodes and all will terminate about the same time, which is the goal. Shahrokhi and Matula derive a strongly polynomial algorithm solving this case [104].

The second class is one with disjoint data segments– a sparse graph. Here the choice of flow paths is strictly constrained to certain edges. This satisfies the nodes and tends to perform well. With no replication there is only one path sender to receiver, so the flow network algorithms will produce an optimal solution.

Unfortunately this algorithm is not guaranteed to provide an optimal solution on all graphs, due to the differing semantics between the goals of instantaneous flow maximization and aggregate transfer termination. Typically multi-commodity flow formulations suffer from the same problem. The maximum concurrent flow problem [104] is the only exception, capturing the right termination semantics, but current solution techniques work only on networks with homogeneous links and peers.

Another problem with network flow is that implementing these solutions requires explicit multipath routing; this is impossible over IP networks without extra infrastructure.

Still another problem is error due to integer arithmetic: transfers are nearly arbitrarily divisible, but these algorithms use integer values for flow rates. This aggregate error by definition will be smaller than (number of edges)$\times$(scaled unit value). That is, if '1' in the graph represents 1Mbps and there are 11 edges, total error may be as much as 11Mbps due to the implicit rounding. The error can be decreased below any arbitrary ϵ by scaling the edges prior to applying the max-flow algorithm, but the values must fit inside a hardware integer (32 or 64 bits) or arithmetic itself becomes expensive.

We mentioned that this algorithm produces optimal schedules for sparse graphs; those without replication. With replication, say by a factor of k, the iterated flow network solution will approximate the optimal by a factor of no worse than $k \cdot\ (1 + \epsilon)$, where ϵ is as above. This is because each iteration will maximize the flow; completing the transfer to at least one receiver's replica. In the worst case, the bottleneck flow will not begin until the final iteration (at which point its flow will be maximized, otherwise it would not be the last iteration) and run to completion. Since it is the bottleneck, it will take the longest time to run, and the prior k iterations will have taken less (or equal) time. Thus the total termination time for a solution using this algorithm is no more than $k \cdot\ (1 + \epsilon)$ times the termination time for an optimal schedule.

That is not a very useful bound, since it is not tight for $k > 2$. For networks with arbitrary replication, adequate capacity, but a single bottleneck link, we can show a more useful bound: $2 \cdot (1 + \epsilon)$. This is via the following observations:

First, to achieve an optimum schedule the bottleneck link must be fully utilized at all times– but if faster links also are ingress to a receiver, max-flow using them may starve that bottleneck. Then maximum termination time is then the "starvation time" plus the bottleneck transfer time. This gives a ratio to an optimal solution of (starvation time + bottleneck time) / (bottleneck time) or equivalently (starvation time)/(bottleneck time) + 1.

Starvation time can be no more than bottleneck time; otherwise the other link would be the system's bottleneck. Thus the ratio is at most a factor of 2. This is in addition to the integer error factor described above.

Figure 4.3 shows one example where performance approaches this bound. Many network flow algorithms would simply select the path using source 1 and data block A, maximizing the instantaneous flow at 10 units. The iterated algorithm would then complete block A at time 10, and finish block B after another 10, for a total completion at time 20.

For comparison, one optimal solution takes 9 units of flow through block A and 1 unit of flow through block B until time 10 (completing block B), and then completes the remaining 10 bytes of block A at time 11. The ratio here is thus 20/11 or about 1.8 times worse than optimal. This is near both worst-case bound shown earlier.

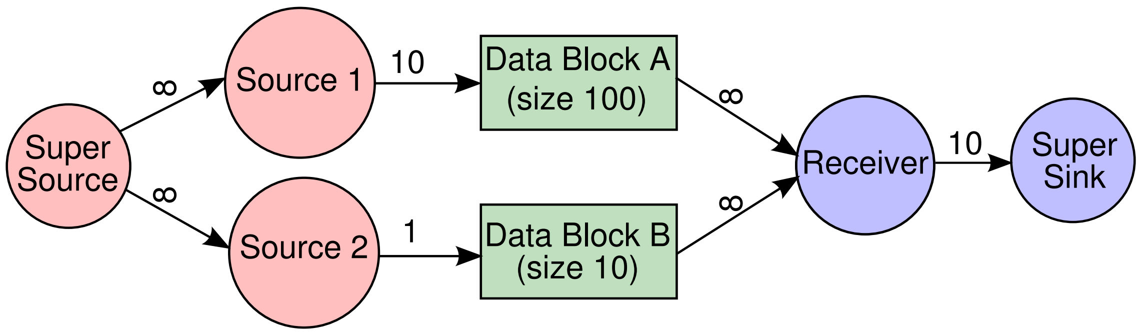


Figure 4.3: Example of Poor Performance Using Iterated Network Flow

4.2 Linear Programming Algorithm

Linear programming (LP) methods have been used to efficiently and optimally solve min/max problems for decades. Transfer scheduling can be seen as global maximization problem, so it makes sense to try and use LP techniques. While we know this will produce an optimal solution given the right input equations, the algorithms involved are more computationally expensive than other approaches.

4.2.1 Conversion to Linear Equations

To create the desired linear equations, we again work from the canonical segment graph. In this case, we think of the problem as one of weighting edges: figuring out the proportion of a sender's or receiver's capacity spent transferring each data segment. These weights are a rate-based transfer schedule. Note that while this solution targets maximizing flow, this flow is subject to the constraints missing from the network flow problem. Confusingly, max-flow here has the right semantics and is optimal, while max-flow there had the wrong semantics and need not be optimal.

As we know, a simple flow network has a problem with node starvation. A non-iterative approach to this is replacing shared links with unshared, proportional links whose capacity sums to the original. This is the same argument given for the transformation in Section 3.2.2, and an optimal solution for this problem will give us an optimal solution in general. Compare the two graphs in Figure 4.4 (the first of which is replicated from Figure 4.1). For the sake of this exposition, assume the sizes of D_1, D_2, D_3 are the same.

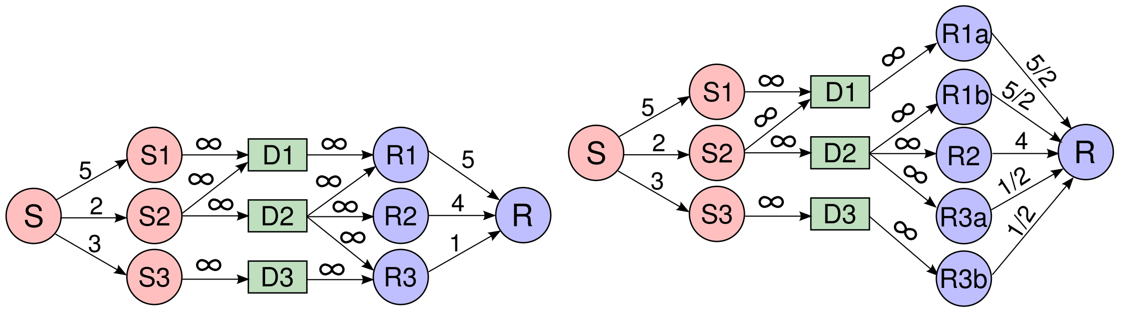


Figure 4.4: Proportional Flow Network Graph

An interesting observation at this point is that, ignoring network constraints, network flow here would produce an optimal result. Starvation cannot occur as there is always a path through each receiver to push more flow. Unfortunately, accounting for network constraints, network flow alone still cannot guarantee non-starvation.

The next step is to apply the single-infinite-link optimization discussed earlier to simplify the graph. Then, since our transmissions must be proportional to minimize the total completion time, the problem resolves to getting the right amount of flow through data nodes D_i. We have also relabeled the remaining infinite-speed links. This is shown in Figure 4.5, which looks somewhat familiar to our original graph.

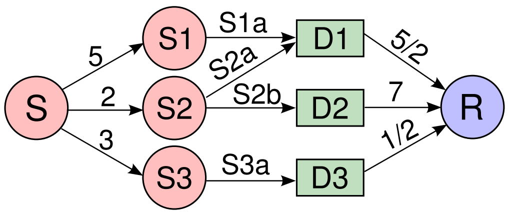


Figure 4.5: Simplified Flow Network

Ideally peers here are receiver limited so each D_i gets sufficient input flow to saturate the output link to R. If not, we need to get flow to each proportional to their fraction of total system demand. This minimizes the total completion time. As before, no non-proportional weighting can finish earlier: it may cause some flows to do so, but either it makes no difference to other flows or it makes them finish later.

At this point our constraints (the node speeds, data accessibility, and termination requirements) are clear, and Table 4.1 shows the equations. TD and TS stand for Total Demand and Supply, PD and PS stand for Proportional Demand and Supply, SD stands for Supply to D, and S_*,D_* are values to/from each S or D node as labelled in the graph.

Table 4.1: Linear Equations Derived from Sample Problem

Proportional Demand	Supply to D's:	Proportional Supply	Link Constraints
$TD = D_1 + D_2 + D_3$	$SD_1 = S_1 S_{1a} +$	$TS = \sum_1^3 SD_i$	$S_{1a} \leq 1$
$PD_1 = D_1/TD$	$S_2 S_{2a}$	$SD_1/TS \leq PD_1$	$S_{2a} + S_{2b} \leq 1$
$PD_2 = D_2/TD$	$SD_2 = S_2 S_{2b}$	$SD_2/TS \leq PD_2$	$S_{3a} \leq 1$
$PD_3 = D_3/TD$	$SD_3 = S_3 S_{3a}$	$SD_3/TS \leq PD_2$	

Given these equations and the values calculated for Figure 4.5 we can use well known LP techniques such as the Simplex method or ellipsoid algorithms [86] to determine the weights S_{1a}, S_{2a}, S_{2b} and S_{3a}. These provide our rate schedule.

4.2.2 Solving Generalized Linear Inequalities

This section generalizes the example from the prior section, showing how to derive equations in standard form for linear programming. Seen another way, we are roughly converting the mathematical constraints from Section 2.4 to a more explicit functional representation. The notation is, at best, rather painful. First we initialize terms using the graph, as discussed above. Here is a list of those terms (all constants after the initial calculation except $S_{i,j}$). Let:

- $|Send_i|$ be the speed of sender i.
- $|Recv_i|$ be the speed of receiver i.
- $Data_i$ be some data segment i, $|Data_i|$ its size.
- $Needs_i$ be the total amount of data $Recv_i$ needs ($\sum_{i\ needs\ j} |Data_j|$)
- $PropDemand_{i,j}$ be the proportional demand that i creates for $Data_j$, $((|Recv_i|/Needs_i) \cdot |Data_j|)$
- $Demand_j$ be the total demand for a data segment ($\sum_i PropDemand_{i,j}$)
- $TotalDemand$ be the total demand in the network ($\sum_j Demand_j$)
- $PropDemand_i$ be the proportional demand for a segment ($Demand_i/TotalDemand$)
- $S_{i,j}$ be the link from a sender S_i to a data segment Dj.
  These are the variables we solve for!

Calculating the input values takes one pass over the receivers and their edges, so takes time no more than $O(n + m)$ arithmetic operations. Listing 4.2 gives pseudocode for this pass. Then, given those values, we set up inequalities based on the graph as before. Table 4.2 summarizes the resulting constraint equations. Finally, put into a more suggestive form we get the equations shown in Listing 4.3. The values of K, L, M obviously depend on the size of the graph being converted.

Table 4.2: Summary of Linear Programming Constraint Equations

Constraint Type	**Conversion To Equations**
Supply to data segment nodes	$Supply_j = \sum_{Send_i \text{ links to } Data_j} \lvert Send_i \rvert S_{i,j}$
Proportional supply	$Supply_j \leq PropDemand_j \cdot TotalSupply$
Non-starvation	$Supply_j > 0$
Link bandwidth conservation	$\sum S_{i,j} \leq 1$

```
$TotalDemand = 0$
# Set the demands on data segment:
For each receiver $Recv_i$, speed $|Recv_i|$
    $Needs_i = 0$;
    For each data segment $Data_j$ they need
        $Needs_i += |Data_j|$
    For each data segment $Data_j$ they need
        $Demand_j += |Recv_i|/Needs_i$
        $TotalDemand += |Recv_i|/Needs_i$
For each data segment $Data_i$
    $PropDemand_i = Demand_i/TotalDemand$
```

Listing 4.2: Determining Linear Equation Coefficients

```
Maximize $TotalSupply = \sum_i Supply_i$
Subject to
    $\forall j : Supply_j = |Send_{i_1}|S_{i_1,j} + |Send_{i_2}|S_{i_2,j} + ... + |Send_{i_K}|S_{i_K,j} > 0$
    $\forall j : Supply_1 + Supply_2 + Supply_3 + ...$
        $+(1 - PropDemand_j^{-1})Supply_j + ... + Supply_L \geq 0$
    $\forall i : S_{i,0} + S_{i,1} + ... + S_{i,M} \leq 1$
    $\forall i : S_{i,0} + S_{i,1} + ... + S_{i,M} >= 0$
```

Listing 4.3: Final Linear Program

At this point it should be clear that this system of equations can be solved straightforwardly with LP techniques. A further optimization is to run a secondary optimization on the residual graph after allocating this bandwidth, to increase total transfer speed. Although this will not affect the total finish time of the run, it allows some flows to terminate early.

Important cases are when we are either totally receiver limited or totally sender limited. Then the problem has a simple structure and we get optimal solutions either via the flow network or via this mechanism. The complicated cases, when finding the optimum is most difficult, are when some large number of nodes are sender limited and others are receiver limited. “Large” means on the order of hundreds or thousands, which is well within the range of LPs we can solve with today’s technology. Finally, in the static case, an LP solution gives an optimal result up to one-byte rounding errors.

4.2.3 Conversion of Rate-Based and Explicit Schedules

We have used rate based and explicit transfer schedules interchangeably. The only difference is how transfer interleaving is done. Explicit schedules send as fast as possible to one peer, then another, and so on. Rate-based schedules do it implicitly, specifying rates to multiple peers concurrently; the interleaving is done by the transport protocol/OS. This section shows how to convert between the two in time no worse than $O(mn)$ where m is the number of edges in the graph and n is the number of nodes.

Explicit Transfer Schedule → Rate Based Schedule: Conversion in this direction is easy: we need only divide out the explicit data transfers to get their rates. As rates are now spread out evenly over the entire transfer period, at no time can their aggregate be larger than the explicit schedule: meaning no speed constraint violations occur. The same amount of data is sent, and to the same peers: meaning that no data constraint violations occur. The termination time is the same, meaning it is equivalently optimal. Thus, the rate based solution produced is equivalent to the original schedule.

Rate Based Schedule → Explicit Transfer Schedule: Conversion in this direction is a bit trickier, and tends to be the conversion done in practice to translate solutions to the transfer graph into actual transmissions. This algorithm requires buffering in network, either in the local network stack, device memory, or in router queues. By design it exceeds designated speed constraints for small time periods– just not on average.

```
Let $stop\_time$ be the global termination time for
   the explicit schedule.
For each sender (whose max rate is $max\_rate$)
   Let $total$=0
   For each receiver (desiring $data$ bytes from sender)
      Let $total$ = $total$ + $data$
   For each receiver
      Allocate each receiver rate $\frac{data}{total} \cdot stop\_time$
```

Listing 4.4: Explicit to Rate-based Schedule Conversion

```
Select an interval time $t$ # e.g. 1 second.
For each interval [0,$t$], [$t,2t$], ...
   For each Sender
      For each Receiver
         Sender sends $t \cdot rate$ bytes as fast as possible.
      Sender sleeps until end of current interval.
```

Listing 4.5: Rate-based to Explicit Schedule Conversion

Selection of an appropriate interval is key. In the limit as the interval approaches zero, it becomes exactly the rate based scheme. Similarly, at the end of each interval period, the data sent coincides with that of the rate based scheme. Interval length trades between minimizing overhead and exceeding buffer capacity. 1-10 seconds tends to be a good compromise among overhead, bursty traffic, and buffer overflows. In the worst case we can look at all sender/receiver pairs, and set the interval time to be t=min(buffer size / rate). This guarantees no buffer overflows.

Note that algorithms which iterate over senders and receivers trying to create a ‘cleaner’ schedule with full-speed transfers until each segment is complete tend not to work very well. This is because each receiver must in the worst case account for constraints from all others, leading to a complicated algorithm.

4.3 Greedy Weight Based Algorithm

This algorithm and its derivatives form the core of our implementation. It is similar to the LP algorithm in attempting to proportionally allocate bandwidth among peers based on their requirements. It differs in that it makes primarily local decisions– thus its output is not necessarily optimal but it runs extremely quickly. This is appropriate for latency-sensitive transfers, dynamic transfers that are rescheduled frequently, or those where some constraints are unknown. This section discusses the basic greedy algorithm, how to optimize for high-sharing environments, and how parts of the algorithm can distributed.

4.3.1 Basic Greedy Algorithm for Low Sharing

The basic greedy algorithm works by balancing supply and demand, then dividing out capacity proportional to transfer size. The idea is simple: servers allocate more of their bandwidth to highly desired segments than undesired segments. How much more? Exactly the ratio between the demand for the two segments.

At a high level, the algorithm first calculates the demand for each data segment in the system. Then it allocates senders' capacity proportional to the demand for each data segment they provide. Finally receivers allocate incoming bandwidth proportional to the individual segment's size and subdivide their requests amongst the senders proportional to the sender's weighted speed. The pseudocode in Listing 4.6 gives more detail. Input is again the canonical segment graph form.

The run time of this algorithm is $O(m)$ where m is the number of edges in the input graph. Step 1 look at all receiver links ($O(m)$), Step 2 at all sender links ($O(m)$), Step 3 at both sender and receiver links ($O(m)$). Iterating over nodes is insignificant; there are always more links than nodes. Thus our total run time is at most $O(m)$. In low-sharing or total-sharing configurations, $m \approx n$ and this is linear in the number of peers. For intermediate-sharing configurations it may be as bad as $O(n^2)$.

```
# 1. Calculate proportional demand:
   For each receiver:
      Sum $total\ size$ = total data required =
         size of all linked segments.
      Label each link with:
         $(receiver\ speed * (segment\ size)/(total\ size))$.
   For each segment, label with total demand:
      Sum up receiver demands for that segment
         # (marked on links).

# 2. Calculate proportional supply:
   For each sender:
      Sum demand for all segments sender provides
         # (marked on segments).
      For each linked segment,
         provide to that segment bandwidth:
            $(sender\ speed) * (segment\ demand/total\ demand)$

# 3. Output rate schedule:
   For each sender:
      For each segment they provide:
         # equally allocate rate to receivers.
         Send to each receiver at rate:
            $min((total\ segment\ rate)/(\#\ of\ receivers), receiver\ demand)$

# 4. Execute Transfer
   Convert to explicit schedule
   Each node locally optimizes for speed/locality
   Recalculate as necessary
```

Listing 4.6: CEP Basic Greedy Algorithm

This mechanism is optimal when the transfer is strictly sender or receiver limited. Then bottlenecks are all on one side of the graph and proportionally allocating the available bandwidth among them is the best approach. In general this works well in cases with high server replication as that flexibility allows a good balance between supply and demand. It also works well with low replication on either side, as it efficiently finds the limited data supply/demand mapping. It works poorly when there is low server replication but high client replication, as it does not account for second-order scheduling effects: clients providing the downloaded data to others. We address this in the next section.

As before, we focus on data constraints. Step 4 handles complexity in network constraints via simple heuristics. Feedback on experienced transfer speed biases receivers toward appropriate senders over time. If peers enter or leave the system, recalculation can be done efficiently on intermediate values cached after each step: updating aggregate data is done with a constant number of operations, and updating the rage schedule only affects the nodes with which a given peer has dependencies.

When used in a steady-state– rescheduling as peers come and go– one feature of this algorithm is that flow priority is inversely proportional to flow age. New flows have the most remaining to transfer, become global bottlenecks, and are allocated high capacity. In this way they can, e.g. quickly build up a buffer of frames for video playback. Flows near completion are effectively lower priority. This tends to hold servers with nearly-complete replicas in the system for the sake of the global transfer rate.

4.3.2 Optimizing the Greedy Algorithm for High Sharing

The basic algorithm was designed for low-sharing environments; where receivers demand different data segments. With no sharing, a pre-calculated optimal solution is optimal through the entire transfer. But when peers share demand, transfer progress changes data constraints in such a way that the optimal solution may also change.

To address this, we extend the algorithm with a “High Sharing” optimization similar to the two-phase technique used in Bullet [65] or the Super Seed mode in some BitTorrent applications. Our algorithm extends these to a more general case: it supports *arbitrary* sharing and data constraints. Listing 4.7 lists the new algorithm.

```
# Determine proportional demand
For each segment $s$:
    demand[$s$] = $\Sigma_r$(down_rate[$r$][$s$]$\cdot|s|$)
total_demand = $\Sigma_s$demand
capacity = $\Sigma_r$up_rate[$r$]
For each receiver $r$:
    allocation = (up_rate[$r$]/capacity)$*$total_demand
    # Supply proportional to demand
    For each segment $s$:
        If demand[$s$] $\leq$allocation and
         demand[$s$]$>$0 and down_rate[$r$][$s$]$>$0:
            seed[$r$][$s$] = True
            allocation $-$= demand[$s$]
            demand[$s$]0
```

Listing 4.7: CEP High-Sharing Optimization

At a high level, it works by breaking the transfer into an initial ‘seeding’ phase where data is distributed across the network, and a ‘feeding’ phase where peers download data directly. Both phases apply the original CEP algorithm to schedule the transfer, but in the seeding phase the constraints are specially constructed to prepare for an efficient feeding phase. The goal, as before, is to distribute data such that demand (load) on a given node is proportional to its supply (capacity).

This differs from existing approaches which randomly distribute data across the network assuming segments are uniform size (blocks) and load per segment will be equal. We deterministically distribute data exactly how it is needed. Ideally we could estimate the best location for a segment, i.e. place it on the fastest node closest to all clients, for a particularly efficient feeding phase. This is left to future work.

The input for the algorithm is `down_rate[`r`][`b`]`: the speed at which receiver r wants to download segment s, estimated as their total capacity divided amongst the segments they want, and `up_rate[`r`]`: the total upload capacity of receiver r. The output is `seed[receiver][segment]` which is `True` if `receiver` should get `segment` in the seeding phase. This result is in passed as the *output* constraints for the seeding phase and the *input* constraints for the feeding phase. Thinking of this as acting on constraint graphs, we effectively divide it such that peers get fair “slices.” This distribution works well provided network bottlenecks are known– otherwise, the capacity estimates may place inappropriate load on poorly-connected nodes.

4.3.3 Distributing the Greedy Algorithm

This section shows how to distribute some decision-making in the basic greedy algorithm. It still relies on features of a centralized metadata server; most importantly global load balancing tolerant of heterogeneous peers/links and range matching. Without these features, data lookup overhead increases and performance tends to be limited by the capacity of the slowest node. These characteristics are evident in our evaluation (see Section 7.6.1).

The distinction between the basic centralized algorithms and hybrid algorithms is the intelligence of the endpoints. The original scheme uses dumb clients; slaves to the master centralized scheduler. They report all metadata to the scheduler, request a schedule, and implement the response. The server provides the minimum information required: one replica per data range. If failures occur, rescheduling must be done.

The hybrid approach differs in the scheduler’s response: it additionally provides alternate replicas for data the client requires. This allows the end-points to perform independent replica selection based upon their current performance. The logical changes are as shown in Listing 4.8.

```
# Scheduler:
    Schedule as in basic algorithm
    Provide response as with basic algorithm and
    Provide $k$ extra servers selected as follows:
        Calculate residual node, link capacity.
        Of nodes with available capacity,
            sort based on location.
        Return closest $k$ nodes serving data with
            $||overlap|| > min\_overlap$.

# Peers:
    Initially follow schedule as specified by server.
    Take no action violating upload rate guidelines.
    May do anything else to optimize download rates.
```

Listing 4.8: Logical Changes to Distribute Greedy Processing

By enforcing upload rates on peers, global load balancing is enforced. That is, when peers meet desired upload rates then download rates should be what the centralized scheduler expected, and hence the global performance will be as expected. If peers can not meet upload rates, it must be due to a previously unknown bottleneck somewhere in the system. In this case, the extra replicas can be exploited to improve performance. As these replicas have good locality, and LANs tend to have abundant cross-sectional bandwidth, such local traffic tends not to affect the performance of other transfers. Thus this generally can be done without concern for global system performance.

Since this approach does not wholly remove the centralized portion of the system, what good is it? First, it reduces scheduler load: we can increase the period between metadata updates and rescheduling steps without decreasing performance, as peers can optimize locally at the edge. Second, performance improves when scheduler information is inaccurate: whether out-of-date or missing due to dynamic network behavior or unknown bottlenecks. Finally, peers can make progress under failure: it provides automatic fail over at the edge when nodes/links go down without needing to contact the scheduler. The quantitative effect of these features is shown in the Section 7.7.

4.4 Erasure Coding Algorithms

Research in erasure and network coding systems [19, 76] addresses similar issues to this work, so it is natural to explore them as part of a transfer scheduling system. Erasure coding combines blocks to create redundant data, which then provides additional distribution flexibility. Network coding is an extension whereby erasure coding is performed on-the-fly within a network. This can dramatically improve performance: linear codes can achieve the maximum transfer rate for whole-file multicast [73].

Unfortunately, current schemes provide such benefits only for whole-file transmissions. Second, networks exist where it is impossible to achieve the network capacity by network coding [37]; unsurprisingly, their main example is analogous to partial file sharing. Third, max-flow alone does not guarantee globally minimal termination time; other constraints are necessary, as we discussed in Section 4.1.

This section first shows the problems using erasure coding for transfer scheduling, then how these can be avoided to best exploit the performance benefits.

4.4.1 The Problem with Sharing: Dependencies

The nature of erasure coded data is that it creates dependencies between blocks. This allows data to be recovered via redundancy, but it can cause high latency and overhead. Obviously, we wish to avoid these problems.

Optimal erasure codes spread redundancy such that regardless of which specific blocks are lost, provided enough blocks are received in total, the original data is recoverable. They are typically nonsystematic codes– where only encoded blocks are sent rather than the original data stream. The effect is that very little of the original data can be recovered until nearly all of the blocks are retrieved. Even with systematic erasure codes, under loss the same effects can be observed. In the worst case, the first block is lost and the whole file must be retrieved before redundancy allows it to be recovered.

How might erasure codes best be combined with transfer scheduling? Consider a simple input file containing 8 blocks, which has been spread over two servers for load balancing. Figure 4.6 shows this as a simple transfer constraint graph.

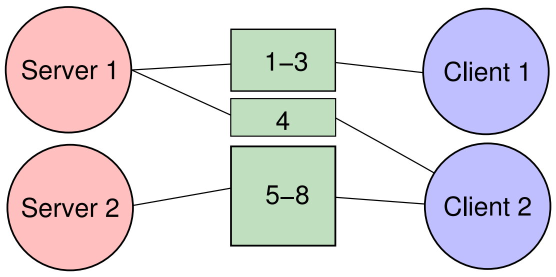


Figure 4.6: Transfer Constraint Graph

Naively applying a nonsystematic erasure code over the input data before the load balancing step would produce a graph like that shown in the left of Figure 4.7. In this graph, servers may may have either raw blocks or encoded blocks. The $\oplus$ symbol refers to a erasure coding combining blocks (via, e.g., XORing) while $\bigcirc$ represents just the block itself. In most cases we expect clients will want to download the 'raw' unencoded blocks, but the graph shows how clients may desire encoded blocks.

The problem with such an approach is the existence of "cross-constraint dependencies." That is, dependencies tying a desired block to an undesired block for a given receiver. In such a case, an edge in the constraint graph crosses a receiver data constraint boundary. To resolve the blocks that the receiver requires, it must download excess data.

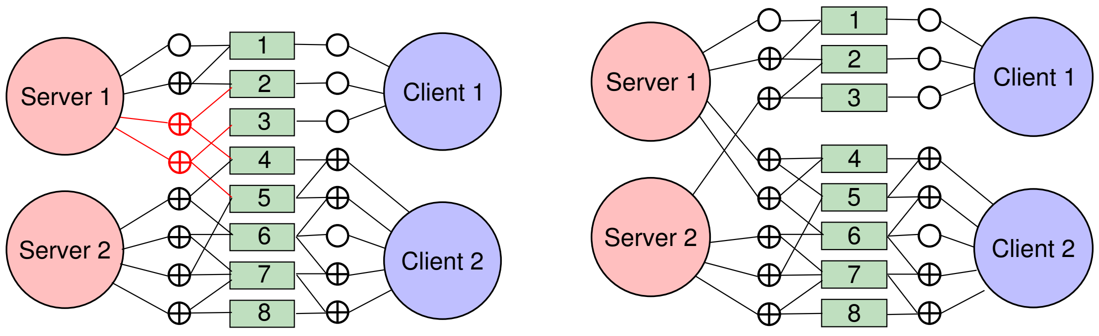


Figure 4.7: Problem With/Without Cross-Constraint Erasure Coding

Two cross-constraint dependencies are highlighted here: $2 \oplus 4$ and $3 \oplus 5$. These force each client to download nearly the entire file, even though they only want a subset of that data. For example, client 1 needs block 1, which it can download directly. It also needs block 2, which it can get from $1 \oplus 2$. Block 3 is only available via $3 \oplus 5$, but we now need 5. That entails downloading block $5 \oplus 7$ and finding block 7. This chain continues until it downloads a minimum of 7 blocks: it needs all but $7 \oplus 8$ to decode block 3: $(3 \oplus 5) \oplus (5 \oplus 7) \oplus (7 \oplus 6) \oplus (6 \oplus 4) \oplus (4 \oplus 2) \oplus (2 \oplus 1) \oplus (1) = 3$. For whole-file sharing this behavior adds only latency. With low sharing it adds unacceptable overhead: nearly the whole file must be downloaded, regardless of client interest.

Applying erasure coding *after* load balancing does not help without knowledge of client demands: a client wishing one block per server could still be forced to download the entire file. The trick is avoiding cross-constraint dependencies. This can be done *if* client sharing is known: (1) split the file into chunks such that if a client wants any part of a chunk, it wants all of it (2) erasure code only over those chunks. Step 1 can be done using the segmentation algorithm from Section 3.3.

In the above example, simply switching the highlighted edges removes the cross-constraint dependency chain. This gives us to the second graph, the right of Figure 4.7. This encoding is load balanced, erasure coded, and has no cross-constraint dependencies.

In summary, the problem with erasure coding for partial content distribution is that it combine bits from disparate locations in a file. With low sharing these are probably not bits a given peer wants– and it have to download excess blocks to decode its data. This overwhelms any performance gains from the coding.

4.4.2 Exploiting Erasure Coding

Comparison between Figure 4.7 and the basic transfer constraint graphs shown earlier reveals that erasure codes add yet another level of generality to the distribution problem: peers may provide/desire not just blocks, but combinations or functions of blocks.

This idea can minimize download latency for erasure coded data in implicitly scheduled systems: receivers determine what would allow them to decode the maximal number of blocks (via, e.g., a depth-first search through the dependency tree), and add a dependency for that combination. This would reduce the effects of nonsystematic code use.

In terms of algorithms, little needs to be changed to handle encodings without cross-constraint dependencies. We simply perform initial scheduling as before, weighting flows as appropriate at the level of byte ranges. Then we account for erasure coding semantics.

Servers must allocate supply proportional to demand for *decoded* blocks. This is complicated if, e.g. a server has block $1 \oplus 3$ and $2 \oplus 3$ but wants to allocate bandwidth equally to block 1, 2, and 3. This is impossible with the current encoded blocks; block 3 gets twice the bandwidth. The server should create block $1 \oplus 2$ and offer equal weight ($\frac{1}{3}$) to each of these encoded blocks.

Clients simply look at which blocks are in a byte range and bias block requests proportional to allocated flow to servers. This is a simplification of the rate-based to explicit scheduling algorithm from Section 4.2.3. We can use well-known techniques such as lottery scheduling to randomly but proportionally request blocks from each server.

Put another way, *within* a given segment, CEP allows bytes to be downloaded in any order at any rate– so long as the aggregate transfer conforms to the schedule. Within a segment, we by definition have a high degree of sharing (total). Therefore all the techniques for block transfer, erasure coding, etc. from other work can be applied– if the segment is sufficiently large to warrant the overhead.

To handle encodings with cross-constraint dependencies, we first perform a modified depth-first search for the blocks we need to download to be able to decode those we actually want. We select the one with the minimal expected time to finish (sum(block size$*$expected rate from that server) over all blocks). Next we mark those blocks as 're-quired'; this removes the cross-constraint dependency. Then we can proceed as above.

Unfortunately, while we can address erasure coding's issues, there is little reason to use them with transfer schedulers. Proponents of erasure and network coding cite lack of management overhead, enhanced system reliability, irrelevance of network structure, and high bandwidth as benefits of the technique. Transfer scheduling already maintains metadata on data and network structure and provides high bandwidth. Similarly, we can tolerate the same class of failures. So in the end, this provides little benefit. The techniques are complimentary: erasure coding techniques are appropriate for latency tolerant, high-sharing transfers; CEP is appropriate for lower sharing, less latency tolerant transfers.

4.5 Algorithm Performance Characteristics

Most of this chapter focused on abstract development of scheduling algorithms. Unfortunately, theory and practice are rarely the same. This section discusses a few caveats for real-world use of these approaches and the heuristics and knobs required to make the system work. We conclude with a summary and comparison between the different approaches.

4.5.1 Theory versus Practice

The discussion of conversion between rate-based and explicit schedules touched upon the problem of unknown network constraints. This is the main difference between theory and practice. Other important aspects include the primarily compute-once nature of the transfer schedulers, the types of networks for which they were designed, and the underlying protocols used: all affect real-world performance. We will return to these ideas in Chapter 6 when we discuss our implementation.

We have focused on pre-calculation and then execution of a transfer schedule. In practice, this is only realistic for three types of networks: (1) relatively static networks such as pre-reserved lambda networks; (2) those with quality of service guarantees; (3) those with known long term performance patterns and very large transfers. These are exactly the high-performance networks described in our target use models.

For more arbitrary networks (e.g. the commodity Internet) static scheduling performs poorly. Performance depends more on dynamic behavior and potentially unknown network constraints than known data constraints. While this is not the focus of this work, our experimental evaluation shows that focusing on data constraints with simple update heuristics is still effective.

Another caveat is that the dominant network protocol on the Internet is TCP. Without traffic shaping, manual tuning, or large amounts of buffering, its congestion control mechanisms induce problems in the wide-area: TCP's AIMD mechanism produces a "sawtooth" that at best achieves only 3/4 of the bandwidth available. This means our algorithms may miscalculate by 25% or more from the ideal, when using TCP as the underlying transport. Attempting to run rate-based transfers over TCP is also problematic for the same reason. TCP's flow and congestion control mechanisms cause the packet sending rate to have little relation to the rate at which the application 'sends' data (passes it to TCP). We discuss this further when talking about our implementation in Section 6.2.1

The critical difference between theory in reality is that CEP must actually implement a calculated transfer schedule; this may or may not be possible, depending on the validity of our input data and assumptions.

4.5.2 Scheduling with Inaccurate Metadata

In an ideal world, the metadata server is an oracle which instantaneously knows the status of everything in the system. In practice, metadata may be out of date, misestimated, or have other problems. Here we classify the types of potential errors, while Section 7.7.2 evaluates the performance effects for transfer scheduling.

There are three types of potential error: **server** error, when it claims it has data it doesn't, or it doesn't have data it does; **client** error, when it claims it needs data it doesn't, or it doesn't need data it does; and **network** error, when we overestimate, underestimate, or have no estimate of capacity.

Well-behaving peers never claim excess supply or demand- such errors only occur with malicious peers. Server lies will be seen as failures on the client side and can be handled as in Section 7.7.1. Clients claiming excess need will reserve excess server bandwidth, which may give them unfair performance but will not otherwise affect the system.

Similarly, clients under claiming need should not happen. Doing so only hurts the client. Servers under claiming data provided may occur in two situations. First, at the start of a transfer, before knowledge of system state is collected; this is unavoidable. Second, during the transfer, clients download data but may not immediately notify the scheduler that they can serve that data– but the information is superfluous. The scheduler can calculate data locations at any point based upon initial conditions if network capacity is known.

Thus everything resolves to capacity estimation. Lack of an estimate is equivalent to an overestimate, as we have only the NIC speed as an upper bound. Overestimates will typically overload the given node(s), while underestimates will shift load elsewhere. This of course depends upon the data constraints. Inflexible data constraints make capacity estimations irrelevant– clients must download data from the specified server.

4.5.3 Heuristics and Knobs

Heuristics for replica selection are the most important thing separating theoretical and actual performance. With insufficient information about network structure or peers to make intelligent central decisions, peers must make an educated guess and adapt as transfers progress. In the limit, with no central scheduler, such heuristics create an implicit scheduler. This section describes our heuristics and the various “knobs” on the system.

The first heuristic is the way peers select among replicas from which to download their data. We take the most simplistic view possible: download data from the fastest peer. So this resolves to peer capacity estimation. Each peer has an upper bound rate which is used initially. After some data has been downloaded, we use actual experienced rates. We ignore other options such as locality or using the network weather service [83].

This single-server selection only applies when downloading small amounts of data. For larger amounts of data (greater than one "chunk," see below), peers automatically download disjoint subsets in parallel. This is again a very simple approach. Other work takes a much more complicated view of data selection, as it is the core of their scheduling mechanism– e.g. BitTorrent's thresholded random-rarest-first block selection.

Similarly, we make no special effort to address the "straggler" problem, where one slow node determines the aggregate termination time. The scheduling mechanisms above already optimize for such nodes so no extra peer techniques are required.

Table 4.3 covers the main parameters common across different scheduling algorithms, and their default values. The "fixed values" are set as part of the implementation and can not be changed without altering the source code.

Table 4.3: System Parameters and Values

Tunable Parameters		**Fixed Values**	
Parameter Name	**Default Value**	**Setting Name**	**Set Value**
Maximum chunk size	16 MB	Global reschedule interval	Never
Metadata update interval	5 seconds	Popularity threshold	100%
Server write queue	16-64 MB	Optimistic schedule delay	0
Maximum concurrency	256		
Transport protocol	TCP		

The "chunk size" is the largest amount of data sent in a single communication between two peers. The larger the size, the more efficient the transfer. The smaller the size, the more responsive the protocol (lower latency). 16MB is tuned for 100Mbps networks or better; at least one chunk transfer should complete in the metadata update interval.

That interval is how frequently metadata updates are sent to the central scheduler. This is a balance between load and data fidelity. A 5-second timeout is sufficient to handle tens of thousands of peers (see Section 7.5.3).

The write queue is how much data a server tries to keep in memory pending a write. 1-4 large chunks is enough to amortize fixed disk read and socket write overhead, and ensure kernel TCP buffers remain full.

We set a maximum number of concurrent transfers to avoid running out of file handles. Attempts to connect or transfer beyond this are simply killed. Similarly, the default protocol is TCP, mainly for simplicity.

The reschedule interval is when the scheduler would perform a global checkpoint, recalculate all transfers, and push new metadata to all peers. We do not support this due to its high cost. We instead incrementally schedule individual peers as new requests arrive.

The popularity threshold is the point at which the “high sharing” optimization engages; 100% means all peers must be demanding the same data. Other values are evaluated in Section 7.10. Currently the user must manually engage this optimization.

The optimistic scheduling delay is how long the scheduler should wait to perform scheduling in hopes additional clients will register information. We want to avoid scheduling, immediately getting new data, and having to continually reschedule. Ideally we schedule just after the pulse of registrations from a synchronized start has passed. By default, scheduling is initiated explicitly by the application code.

While this may seem like a lot of tuning is possible, performance is not very sensitive to these values. A large range is acceptable. We explore the effects of these parameters throughout Chapter 7.

4.5.4 Comparison of Algorithm Features

Table 4.4 (page 73) gives good and bad aspects of each algorithm at a high level. It includes popular algorithms from related work. These include GridFTP [4], a user-level file transfer application supporting striped N-to-N file transfers between clustered nodes above a shared file system, BitTorrent [29], the well-known hybrid centralized/decentralized peer-to-peer file transfer protocol, and Bullet [65], a mesh-based whole-file content distribution network using special data encoding and request striping. Our implementation (Section 6.2.1) focuses on the greedy weight based algorithm for the obvious reasons: its speed, simplicity, and good performance in practice.

The differences between the various algorithms are due to optimization for different criteria; some for network performance, others for fault tolerance, others for security. Each also assumes a slightly different environment as far as node connectivity, homogeneity, and trust are concerned. The algorithms presented in this dissertation optimize for global aggregate bandwidth, assuming a connected network, heterogeneous nodes and speeds, and shared trust among nodes. The most critical difference is that they do not assume full replication and high sharing among all nodes in the transfer.

Our algorithms were designed so that a user will generally achieve best performance by specifying the least restrictive constraints. That is, the more flexibility allowed in the problem specification, the more room for optimization by the transfer schedulers. However, if desired, the user can indirectly control the amount of work assigned to a node by scaling its speed or the amount of data which it holds/requires. For example, if a transfer is specified with no overlaps (e.g. a scatter/gather operation), the scheduler has no freedom of choice and will do exactly what the user specified.

Table 4.4: Summary of Algorithms

Algorithm	**Good Characteristics**	**Bad Characteristics**
Network Flow	Relatively fast, often optimal, dynamic features	Very poor performance on edge cases.
Linear Programming	Known Optimal results.	High overhead, slow algorithm, static scheduler.
Greedy & Hybrid Greedy	Simple, very fast, dynamic features, good performance.	Not optimal in general case.
Erasure Codes	Simple, optimal in some cases	High CPU use, high latency, bulk transfer.
File Transfer (e.g. GridFTP)	Simple, very fast, integrated security.	No scheduling. No heterogeneity, full or no replication.
Peer-to-peer (e.g. BitTorrent)	Very scalable, fairly fast and fault tolerant.	Full replication only, little security, not optimal.
Multicast (e.g. Bullet)	Very scalable, good performance.	Full replication only, fault tolerance.

4.6 Summary and Conclusion

This chapter has introduced several new algorithms for transfer scheduling that efficiently produce near-optimal schedules for our target environment. We have developed a useful, known optimal baseline: the LP algorithm. We also have our main approach, the much faster Greedy algorithm and its optimizations.

All algorithms exploit multiple nodes and simultaneous transfers to avoid single-node bottlenecks. All use hybrid centralized/decentralized scheduling to exploit global characteristics while tolerating limited knowledge. This leads us nicely toward the coming chapters where we build a full system around these algorithms and evaluate them under a variety of conditions.

At this point, we have a fairly useful result. We have shown that given transfer constraints we can translate to a conical form and approximate an optimal transfer schedule in worst-case time $O(n^2)$; commonly in time $O(n\ log\ n)$. This is no worse an order than simple comparison-based sort. This is true even though the general form of the transfer scheduling problem is NP complete.

Acknowledgements: Material from this chapter, in part, appeared in “The Composite Endpoint Protocol (CEP): Scalable Endpoints for Terabit Flows,” Eric Weigle and Andrew A. Chien, Proceedings of IEEE Conference on Cluster Computing and the Grid (CCGRID), 2005. Other material from this chapter, in part, appeared in “Partial Content Distribution on High Performance Networks,” Eric Weigle and Andrew A. Chien, Proceedings of the IEEE International Symposium on High-Performance Distributed Computing (HPDC), 2007. The dissertation author was the primary investigator and author of both these papers.

Chapter 5: System Design

This chapter addresses the design of the Composite Endpoint Protocol, focusing on the external interfaces and internal structure. Relevant implementation details are covered in the next chapter, and previous chapters have discussed the algorithms involved.

The programmer's API and command-line programs exporting it represent the entire user interface. Simple, versatile, and efficient interfaces target our goal of simplicity and ease of use. Similarly, a logical breakdown of system internals allows for efficient implementation, targeting the ultimate goal of high performance data transfer.

5.1 Overview

At a high level the life cycle of a CEP transfer is a simple loop:

```
1  User provides initial constraints
2  Loop:
3      Generate a transfer schedule
4      Start transferring data
5      Wait for modified constraints (user or system)
6      If all peers complete, stop; Else continue
```

Line 1 and 5, constraint specification, are the only part of this process involving the user. Constraints are input via one of the the CEP application-programmer interfaces (APIs). Unsurprisingly, the lowest-level interface mirrors the input specification given in the Analysis chapter, Section 3.1. In a dynamic implementation, the user may explicitly change the constraints as the transfer progresses (line 5) in addition to the implicit constraint changes as downloads complete.

Line 2 and 6 are the abstract flow of control. The system is based off a simple state machine. We discuss this in detail later in this chapter; see Section 5.3. Line 3, generating a transfer schedule, was discussed in the last chapter.

Line 4, transferring data, is implementation specific. The message protocol is part of the state machine discussed in Section 5.3.1. Implementation details are in Chapter 6.

This chapter starts with details on the various APIs (line 1,5) we have developed to feed the transfer scheduler (line 3). Then we show how they naturally lead to the protocol's flow of control (line 2,6). This chapter begins to address the issues of complexity and fault tolerance raised in our problem statement.

5.2 System Interfaces/APIs

The use model determines the best interface. Our first use case was a single *logical* transfer comprised of many individual smaller transfers, e.g. "Move this huge database from this cluster to that cluster." This form of composite communication motivates the global maximization and termination semantics the transfer scheduling algorithms provide.

The alternative is a peer-to-peer model where different users specify sub-transfers in a global transfer namespace, e.g. "Fetch the second hour of video, skipping these commercials." This form of communication motivates the local optimizations allowed through hybrid scheduling.

In either case, the system must acquire the same data: a list of senders, receivers, the data each sender provides, and the data each receiver desires. This is exactly the data required for input to the transfer scheduling algorithms. For a single user, information is all specified at once. For a peer-to-peer system, each user specifies their piece.

All this information need not be specified explicitly– a user may describe their problem at various levels of detail. At the lowest level, they explicitly provide everything. At higher levels, they provide a general goal and the system determines the specifics. We map onto pre-existing interfaces for file transfer and Unix sockets, generating some constraints automatically. This is a natural way to simplify the interface; easing user understanding and optimizing for common requirements.

The most useful way to think of all these APIs is as an interface to a specialized database, one maintaining transfer constraints in the canonical graph structure. Calls that update information maintain the consistency of this structure.

To simplify the implementation, mixing use of separate APIs within a single program is not supported. Different processes participating in a transfer can use any API: at the protocol level everything interoperates.

Lastly, we introduce a new term in this chapter: “DBS.” This stands for Distributed Byte Stream, which is what we call an instantiation of a transfer schedule. The term extends the idea that TCP is a byte stream protocol, while CEP provides a global, logical byte stream in a distributed system. The idea is that this provides a new namespace: internally segments are named by start and end bytes in the DBS.

The next three sections cover the three main APIs accessible to the user, starting with the most powerful. After that, we give two useful extensions to the APIs making them more useful for particular types of transfers. Finally we give an example program and summarize the features of each interface.

5.2.1 Low-Level API

The low level API allows users to specify individual constraints, which directly modify the canonical segment graph structure. This is intended only for programmers’ use. We expect users will want to use a front-end to the system, such as the command-line interface described in the next section. The additional flexibility offered at this level is most useful when transfers include memory components or particularly complex mappings.

This API consists of a small number of operations. We present these as C function calls associated with a CEP handle, but all APIs have corresponding $C++$ class interfaces. Implementations simply look up the handle and set fields in a structure as required.

`int cep_open(uint64_t id)`: This creates and returns a CEP handle. The `id` is a transfer identifier required to distinguish between different CEP transfers occurring simultaneously. It is analogous to the port number in TCP or UDP and is used globally by all peers participating in the transfer. Commonly, this will be a random number or the hash of a filename/Uniform Resource Identifier (URI). By default the scheduler is assumed to be on the local machine; if not, the user must make a call to explicitly change the scheduler.

`bool register_info(int handle, dbsInfo *dbs)`: This adds info on a "dbs"– a segment and associated metadata– to the database/scheduler. Alternatively, it can be seen as mapping local data into the global DBS namespace. The dbsInfo fields include:

- `peerID`: some network identifier for the peer. For the TCP-based implementation it is a character string name like "foo.ucsd.edu" or "192.168.10.12" and a 16-bit unsigned port.
- `speed`: a 32-bit integer representing the estimated max speed of the peer, as used in the transfer scheduling calculations. This can be in any units so long as all peers are consistent; we typically use kilobits per second.
- `filename`: a character string representing an actual file or memory location. URI syntax is also allowed for compatibility with Globus XIO (see Section 6.3).
- `byte_offset`: a 64-bit integer representing the offset within `filename` at which our segment begins.
- `byte_begin`, `byte_end`: 64-bit integers representing the location in the global namespace where the data is mapped.
- `server`: boolean, true if the peer has this data, false if the peer wants this data.

We logically require two pairs of numbers to map local file/memory to global DBS data: byte ranges in the both namespaces. Since the two ranges are by definition the same size, we need specify only three numbers. We opted for a global range and a local offset; alternatively we could have specified local/global offsets and a single length.

`bool unregister_info(int handle, dbsInfo *dbs)`: removes the entry from the database, mirroring the register call. The semantics differ slightly, however. If the entry does not completely cover an existing entry, it removes just the overlapping segment. This might be necessary, for example, when a transfer has partially completed and we wish to reschedule based on the current data layout. Both this and the register function are overloaded with explicit versions taking the dbs fields as parameters directly.

`void schedule(int handle)`: invokes the current transfer scheduling algorithm. This function may be called at any time to generate or regenerate the transfer schedule based on current database information. The ability to reschedule complicates the implementation and is too computationally expensive for some algorithms.

`bool transfer(int handle, bool block)`: Begin or continue the actual data transfer. Must be called after `schedule()`. If `block` is *true* the call will not return until the transfer has completed (returning true) or failed (returning false). Nonblocking calls return immediately. Minor functions not discussed here are used to check the progress of nonblocking transfers.

Together these calls provide all the information to construct the transfer graphs required for our scheduling algorithms, as well as control transfers in progress.

5.2.2 File Transfer API

The file transfer API provides the abstraction of bulk file transport. In this special case, namely whole-file content distribution, we can provide a less intricate constraint specification. This API was designed for integration with the Distributed Virtual Computer (DVC) work [110]. As before, we give C-style declarations.

`int cep_handle_create( cep_handle_t *handle, string sched, uint16_t port, uint64_t id)`: creates a CEP handle. Analogous to calling the `cep_open()` procedure in the low-level API, but specifies the master scheduler explicitly.

`int cep_register_write(cep_handle_t handle, string peer, uint16_t port, string filename)`: states that the given peer serves the given file. That file data provides the entire global DBS. The required range values are calculated from file system information and entered in the database. In this use model all servers have a copy of the entire file, perhaps via a shared file system. `cep_register_read` is the parallel call for clients, who will each get their own copy of the entire file. Peers can also call `cep_register` which allows limited access to the lower-level API.

`cep_schedule` and **`cep_transfer`**: create the transfer schedule and actually perform the transfer. They have the same semantics as the low level API.

We also provide a command-line tool similar to Globus GridFTP or RFT [4, 92]. It allows users to start a CEP file transfer by passing a scheduler node, filename, and flow identifier. On the server side, a default flow ID can be automatically generated by hashing the file. Clients then use that ID to look up the transfer. This tool in conjunction with the weakly-constrained extension (Section 5.2.4) is useful for performance evaluation.

5.2.3 Sockets-like API

The sockets-like API eases converting legacy point-to-point code to use CEP. It has a similar set of calls to standard Unix sockets; semantics differ slightly since it supports many-to-many communication rather than just point-to-point.

A sockets-like API is attractive in that it can be implemented as ‘shim’ code intercepting socket calls. This allows both legacy and new code to coexist with the same function calls in the same program. One mechanism for doing this is library interposition; the `LD_PRELOAD` feature in Unix-like systems. When not used in this fashion, all calls are prefixed with `socket_` to avoid name conflicts with the standard functions.

`socket`, **`listen`**, **`accept`**, **`send`**, **`recv`**, **`write`**, and **`read`** are implemented with nearly the same semantics as standard sockets. The only difference regards blocking: a write or send may block indefinitely, as multiple clients may be reading the data. Closing the socket forces a blocked write to terminate. `write`'s return value is the total number of bytes written to *all* clients, which may be larger than the buffer size.

`bind` and **`connect`** have the same semantics as standard BIND(2) and CONNECT(2) but we accept either a real or a virtual address. Virtual addresses may refer to a group of peers. Other software, such as DVC [110], provides the infrastructure to map such virtual addresses to a list of physical addresses. We use this list to determine the connection's participants.

`getsockopt` and **`setsockopt`** get or set socket options. For CEP (level IPPROTO_CEP) the options are shown in the following table. The first two are the most important. Most users will need these calls to complete the local-global data mapping required by the CEP scheduler.

`CEP_IDENTIFIER`	Get/set stream identifier
`CEP_POSITION`	Get/set offset in a global namespace (i.e. `seek()`)
`SO_LINGER`	Linger on close if data is present
`SO_ERROR`	Get error on the socket (get only)

`close` is similar to the standard CLOSE(2); it completes the transfer and closes the socket. It has different semantics based on current socket options. If the linger socket option is *not* set (the default), it maps to `shutdown(handle, CEP_FORCED)`. Otherwise it maps to `shutdown(handle, CEP_UNFORCED)`. Forcing a shutdown immediately kills all flows in progress on the local peer regardless of their completion state. Not forcing a shutdown prohibits new connections but waits for peers to finish pending reads before destroying the flow handle.

Lastly, we provide a Globus XIO-like interface parallel to this sockets interface. For more information see Section 6.3.1; low-level details are omitted to avoid repetition.

5.2.4 Weakly Constrained Transfer Extension

This section describes an extension to the prior interfaces for weakly constrained transfers. That is, we can specify a *set* of of peers to send or receive ranges of data. This adds no generality to the sender side– it is syntactic sugar equivalent to duplicating the constraint for all senders. The benefit is weakening receiver side download semantics. Instead of all receivers downloading their own copy of the data, only *one* copy is downloaded. Furthermore, we can exploit this flexibility to download that copy efficiently.

This is useful when peers at the receiver side of a logical transfer share a file system. Any peer can download any part of the file, so long as one copy of the data is saved in aggregate. Naively using the earlier data specification would download too many copies of the data. The only way to avoid it with the earlier APIs is to manually construct a client/server mapping for the transfer. By avoiding this we simplify the user's life, and allow for dynamic load reconfiguration given changing peer load or network performance.

To use this extension one simply (a) creates a virtual peer structure, which can refer to one or more machines, (b) fills it with the cluster nodes involved, and (c) passes it as the registration functions in Section 5.2.1 or 5.2.2. Virtual peers are a feature of the DVC [110] software, but we have re-implemented that small subset of the API to remove the DVC/Globus dependency.

Internally, this flexible specification is translated to an explicit set of constraints simply by dividing the range up and allocating segments proportional to the capacity of each node. The scheduler and transfer runs as before. When rescheduling, we must first re-calculate the constraints before recalculating with the scheduling algorithm: subtract completed segments from the original range and divide as above.

The performance benefits of this approach are discussed in Section 7.6.2.

5.2.5 Block Level Interface Extension

The section describes a second extension: a block-based interface augmenting the low-level API. This simplifies integration with existing block-based applications, allowing them to take advantage of CEP scheduling without major code modifications. The motivation is that most systems use blocks, so we need to support that interface. Legacy code and application integration and is important; that was the same motivation for our sockets-like API.

Luckily, as blocks are merely a special case for ranges, we can do so with minimal changes. The interface we provide is just a wrapper for the low-level API with the same set of functions. The only difference is that ranges are specified in units of blocks, and we add a function to set the block size.

Internally, when a block range is specified we multiply by the block size to get a segment. Then we can aggregate with existing segments using the algorithm in Section 3.2.1. This minimizes the size of the constraint graph. Then we can apply our transfer scheduling algorithms as before.

Like the file transfer API, we provide a command-line tool implementing this interface for testing and performance evaluation. We have also integrated this interface with a BitTorrent client application, described in Section 6.2.2. Section 7.8 studies the performance effects of this approach.

5.2.6 API Summary and Examples

For comparison purposes, Table 5.1 (page 85) gives a listing of the main interfaces discussed in this chapter. We have omitted call parameters to save space. The block-based and XIO interfaces are not listed as they duplicate the low-level and sockets API, respectively. One can see how each API provides essentially the same information, but in different ways. Each targets a different use model.

We considered but did not implement an extension providing a shared memory/ *mmap()* API. The idea was abandoned due to implementation complexity and concern that users would expect different semantics and performance characteristics than those actually provided by the protocol.

Given these interfaces, we return to our example from Section 3.5 (see Page 43). Implementing this with the low-level API gives a program such that shown in Listing 5.1 (page 86). This is actually a valid program that will compile.

While this program statically specifies the transfer to be performed, this information would rarely be specified so explicitly in practice. It can easily be read from a configuration file, calculated by a distributed file system, or based on other application structure. Distributed file systems know the location and replication of distributed objects and want to move them quickly, so mesh naturally with this interface.

Table 5.1: Application Programmer Interfaces

API Name		
	Call	**Purpose**
Low-Level API		
	`cep_open()`	Set transfer ID
	`register_info()`	Provide database info
	`unregister_info()`	Remove database info
	`satisfiable()`	True if a schedule can be created
	`schedule()`	Create transfer schedule
	`transfer()`	Perform transfer
File Transfer API		
	`cep_handle_create()`	Set scheduler, transfer ID
	`cep_handle_destroy()`	Kill flows, state
	`cep_register_write()`	Provide database info: server
	`cep_register_read()`	Provide database info: client
	`cep_register()`	Provide database info: low-level
	`cep_get_info()`	Get transfer status information
	`cep_schedule()`	Create transfer schedule
	`cep_transfer()`	Perform transfer
Sockets API		
	`socket_socket()`	Initialize state
	`socket_listen()`	Set master to localhost
	`socket_accept()`	Accept metadata transfer
	`socket_send()`	Register write, implicit schedule, and do transfer
	`socket_write()`	Same purpose as send()
	`socket_recv()`	Register read, implicit schedule, and do transfer
	`socket_read()`	Same purpose as read()
	`socket_bind()`	Implicitly provide name info
	`socket_connect()`	Implicitly provide group info
	`socket_getsockopt()`	Get database information
	`socket_setsockopt()`	Set transfer ID, database location, termination constraints
	`socket_close()`	Terminate local or global transfer
Weakly Constrained API		
	`create_collective()`	Create a “collective” == a cluster == a virtual peer
	`add_range()`	Add a peer/range to the collective
	`remove_range()`	Remove a peer/range from the collective

```
#include <stdio.h>
#include "cep.h"

int main(void) {
   int fd, idx;
   dbsInfo *dbs= {
      // peer:port, speed, name, byte offset, byte begin/end
      {"S1.some.com:5555",  100, "file:///data/big",0,  1,  6},
      {"S2.some.com:5555", 1000, "file:///data/big",0,  4,  7},
      {"S2.some.com:5555", 1000, "file:///data/big",0, 10, 13},
      {"S3.some.com:5555",  100, "file:///data/big",0, 11, 16},
      {"R1.ucsd.edu:5555", 2000, "file:///tmp/copy",0,  1, 16},
      {"R2.ucsd.edu:5555",  100, "file:///tmp/set1",0,  1,  3},
      {"R3.ucsd.edu:5555",  100, "file:///tmp/set2",0,  4,  6},
      {"R4.ucsd.edu:5555",  100, "file:///tmp/set3",0, 11, 13},
      {"R5.ucsd.edu:5555",  100, "file:///tmp/set4",0, 14, 16},
      {NULL, 0,  NULL, 0, 0, 0}
   }

   // a 64-bit XOR-folded MD5 hash of "My Data File"
   fd = cep_open(0xe79d8fd79b44cf61);
   for (idx=0; dbs[idx].peer!=NULL; idx++) {
      register_info(fd, dbs[idx]);
   }

   schedule(fd);

   if (transfer(fd,true)) {
      printf("Transfer succeeded\n");
      return (0);
   } else {
      printf("Transfer failed\n");
      return (-1);
   }
}
```

Listing 5.1: Example CEP Program

5.3 System Internals

CEP actions are driven by the current transfer constraints, which are updated by receipt of messages or local timeouts. We use a simple state machine approach, where events (changes to the transfer constraint database) correspond to changing states. This section first discusses the CEP state machine, then how we manage metadata, and why we chose not to use the DHT abstraction as part of the core CEP design.

5.3.1 Control Flow and Messaging

Internally, all implementations are driven by changes to the local database of transfer information. Figure 5.1 gives the state machine for these events. These events are usually triggered by external messages, the most important of which are listed in Table 5.2.

We discuss the format of these messages and other low-level details in the implementation chapter. They are passed over a reliable transport mechanism such as TCP. This assumption greatly simplifies the protocol design, allowing us to ignore many types of errors from the underlying network.

Table 5.2: Primary CEP Messages

Control Messages		
	Spawn Client	Ask a remote daemon to act as a client.
	Spawn Server	Ask a remote daemon to act as a server.
	Quit	Ask a remote daemon to quit.
Metadata Messages		
	Get	Request a transfer schedule for one segment.
	Put	Provide one segment worth of a transfer schedule.
	Take	Remove a peer/segment from a remote database.
Data Transfer Messages		
	Read	Request a range of bytes.
	Write	Provide a range of bytes.

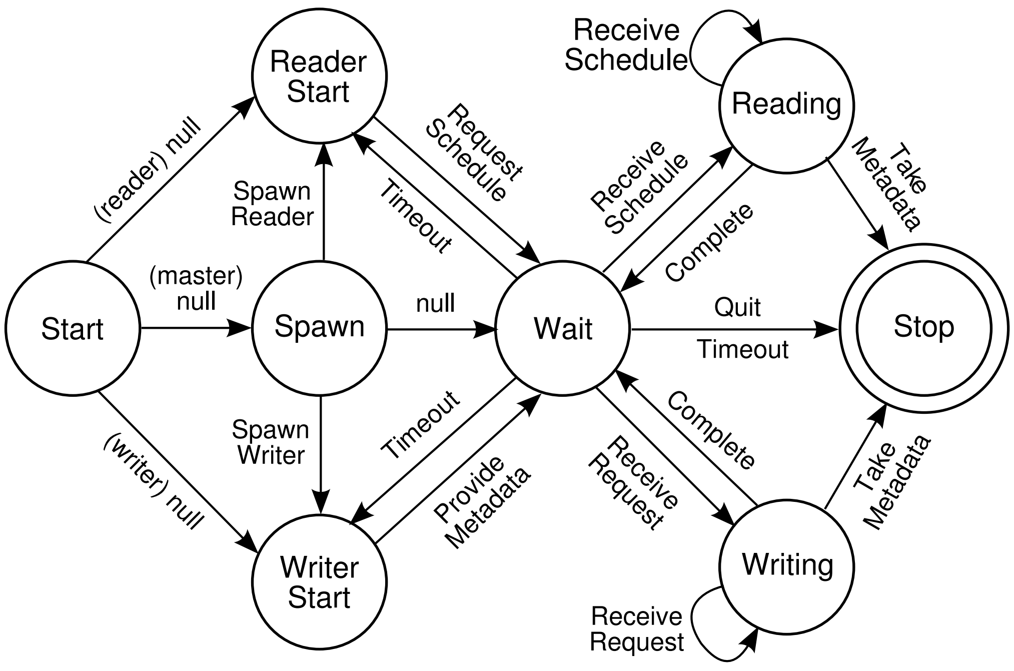


Figure 5.1: CEP System Internal State Machine

5.3.2 Centralized Metadata Management

Having a single centralized metadata server offers several benefits. It provides a single point of control, a single global view on the network for scheduling, a canonical location for peers to find the metadata they require, it is simple, eases implementation, and high-end hardware can be employed to optimize performance. The downsides are that it is a single point of failure, and may limit scalability. We address the alternatives in the next two subsections. Even here, only metadata is centralized; data is always distributed and processed in parallel.

Figure 5.2 shows a simplified view of the structure in each node of the system. The user API provides and receives information to/from the transfer scheduling mechanisms, and also indirectly controls the internals of the system. The transfer scheduler in turn decides the actual network transfers which occur. The system management mechanisms get or set information and work with the transfer scheduler to actually control the transfer(s).

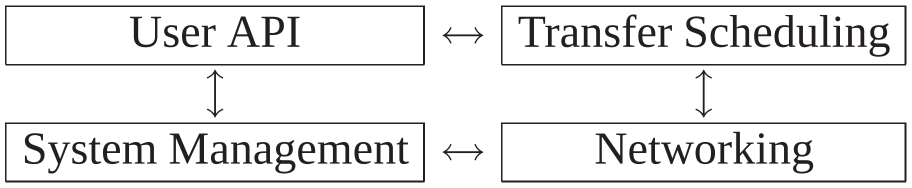


Figure 5.2: System Structure

This use model has a centralized “master” process which takes user input, calculates a transfer schedule, and sends it to remote “slave” processes to actually implement. The master process maintains the constraint database and the slaves are mindless daemons. Only one CEP process is required per physical node. Each process may take part in any number of concurrent logical CEP transfers, and within each transfer there may be multiple network flows. Figure 5.3 shows this structure, highlighting the difference between master-to-daemon control flow and daemon-to-daemon data flow.

The user first creates an application and links with a shared CEP library. After starting daemons on all peers participating in the CEP transfer[1], they run their code. It adds information via the APIs described above, which is sent to the master– the local CEP library, unless otherwise specified. The master schedules the transfer and sends commands to the daemons, which implement the actual data transfer. The daemons are simple applications using the same CEP library; they only set up access control and wait for commands.

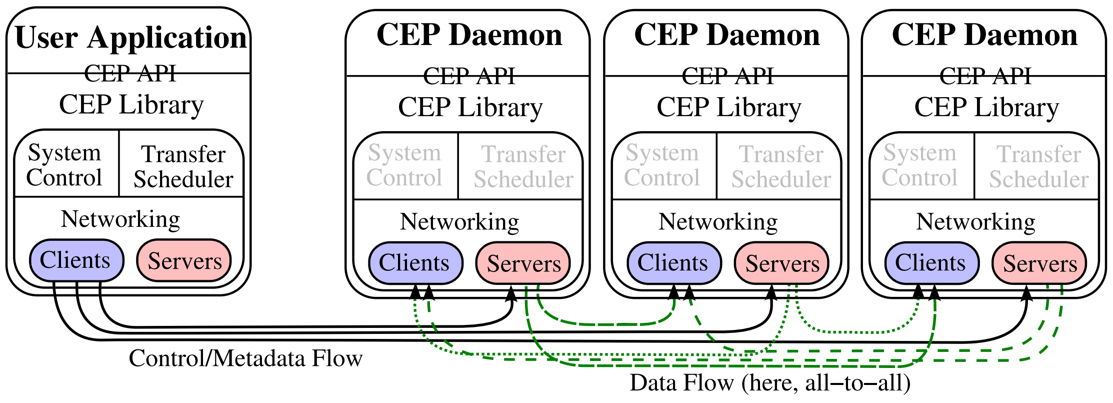


Figure 5.3: Overview of Global Structure: Push Model

[1]A tool such as `inetd` can be used to launch the CEP daemon automatically.

5.3.3 Distributed Metadata Management and the DHT Abstraction

Our core techniques are based on the idea of distributing transfer work across multiple nodes to improve performance. Centralized scheduling therefore requires some justification; especially given the alternative distributed hash table approach. This section discusses the difficulties in transfer scheduling using that abstraction.

A naive straw-man approach is to place data into the DHT structure directly, as a value keyed by start/end byte. That is, hash the start/end byte to determine where the data should be stored, then send it to that node. This is a bad idea.

First, it creates at least one unnecessary transfer to copy source data to one or more other nodes. They may be far away, have limited bandwidth, limited disk space, or high load– the DHT abstraction hides this from the user. In general, our nodes need not be homogeneous, but DHTs treat them as such[2], causing performance problems.

Second, popular data induces high server load. The best way to do load balancing in a DHT is still under active research. It can be partially addressed by caching (as in Coral [45]), but caching does not help one-time transfers, those with low sharing, or those with mutable data. It may add latency to locate data, and the cache object abstraction meshes poorly with byte-range-based transfers.

A third problem is that we are not exploiting known problem constraints given to CEP: data may already have high replication and exist near to potential clients. Forcing the DHT abstraction throws away that valuable structure.

A fourth problem is central to the abstraction provided by DHTs: they provide exact match key lookups. In contrast, we support “fuzzy” matching based on overlapping byte ranges. There are only three ways to support such matching in the DHT abstraction. First is registering block information for all subranges that might be queried. Second is partial centralization, via e.g. partitioning the range-space over peers and having each peer

[2]Some peer-to-peer systems support limited heterogeneity via “supernodes,” but this single-level hierarchy does not solve the problem.

manage every range which overlaps its subset. Then peers can do a limited amount of fuzzy matching on requests. Third is simply treating the range as a set of blocks which can be individually exact-matched, abandoning the idea of ranges.

The first approach is too expensive, enumerating $(s(s+1))/2$ keys for a range of size s. By creating a multi-resolution map this can be improved to $O(logs)$ keys; but the lowest-resolution maps induce high load on the nodes managing them. In either case, determining the best replica has little relation to finding the range in a DHT; although finding *some* match is a prerequisite. The second approach has problems with load balancing as most data tends to fall in a small subset of the total 2^{64} name space. The third approach fundamentally changes the semantics of the problem, as discussed in the introduction.

We can partially solve these issues with another layer of abstraction. Instead of storing data directly in the DHT, we store block mappings. Peers publish block IDs as keys, and peer addresses as values. Now peers look up data, find an exact match on the block ID, and contact the peer to download the block. This avoids the initial copy, but not other problems. We need to know *which* value is returned when duplicate keys are registered; multiple peers may all say “I have this block.” The behavior is typically undefined: DHTs offer no strong consistency guarantees for multiple concurrent servers. Optimizing load means getting the *best* value, not just *some* value.

This leads to still another layer of abstraction. Peers look up the block ID in the DHT and find a “manager” peer. They contact that peer, which maintains a list of servers for the block. It replies with the best replica(s) based on locality, load, or another metric. This is essentially what file systems built on DHTs do to provide consistency. Unfortunately, it still does not completely address the problem; in the worst case, with a single large/popular block, the manager peer has to maintain as much state as a centralized server. Peers also have insufficient information to perform global load balancing or efficiently detect locality.

In summary, using a DHT to distribute transfer scheduling metadata is fraught with problems: poor network locality, poor toleration of heterogeneous nodes, issues with load sharing/hot spots, insufficient consistency/mutability semantics, and exact-match only lookup. To address these problems, one must construct extra infrastructure augmenting the DHT. Unfortunately such infrastructure tends to subvert the benefits of using a DHT in the first place. Finally, even if all these issues were solved, how to globally optimize data transfer in such a structure is an open question.

The ultimate conclusion is that a DHT is the wrong abstraction for transfer scheduling; or equivalently that transfer scheduling is the wrong technique for DHTs. Scheduling and centralization go together, decentralization and replica selection heuristics go together. We focus on hybrid centralized/decentralized approaches. Section 7.5.3 provides some benchmarking data showing performance of the CEP metadata server and alternatives such as a DHT metadata server.

5.4 Summary and Conclusion

This chapter introduced the interfaces to CEP and internal structure of the system. Domain-specific interfaces with simple, versatile APIs provide users a straightforward way to capture their constraints and efficiently perform many-to-many network transfers. Likewise, a simple centralized scheduler design permits efficient implementations, which we discuss in the next chapter. Later, Chapter 7 will prove that these interfaces also provide high performance in a variety of environments.

Acknowledgements: Material from this chapter, in part, appeared in “The Composite Endpoint Protocol (CEP): Scalable Endpoints for Terabit Flows,” Eric Weigle and Andrew A. Chien, Proceedings of IEEE Conference on Cluster Computing and the Grid (CCGRID), 2005. The dissertation author was the primary investigator and author of this paper.

Chapter 6: Implementations

The remaining evaluation in this dissertation is largely through experiments. Thus, understanding the implementation pertains directly to proving the claims we have made. We follow the design discussed in the prior chapter and include the best transfer scheduling algorithms from Chapter 4. Our primary interest is in the greedy weight-based algorithm and its derivatives, and the linear programming algorithm for comparison purposes. We do not consider network flow or erasure coding algorithms further.

To minimize implementation cost, we leverage existing technology when possible. We use existing reliable transports as the underlying protocol– such as TCP, GTP, and others available in the Globus XIO framework. We exploit the MACEDON/MACE [62, 100] project for network overlays. We use various languages and libraries as appropriate.

At a high level, there are three common approaches to designing network libraries. The multi-process (MP) architecture uses a process to handle each task. The multi-threaded (MT) architecture uses a thread for each task. The single-process event-driven (SPED) architecture uses a single process to handle all tasks; they are broken into sub-tasks which are handled by a global event loop and call-back functions. Other variants, such as the asymmetric multi-process event-driven (AMPED) architecture [85], are a hybrid. AMPED uses a main SPED process with a MP pool of helper processes;;this avoids problems with blocking IO encountered on some systems. With careful tuning, all approaches can potentially provide similar performance– assuming a well-behaved operating system.

This chapter describes two implementations, the infrastructure common to both, and relevant engineering issues. Our first implementation uses the SPED architecture and standard Unix sockets. Our second implementation uses the MT architecture; it has better performance, supports more network protocols, scheduling algorithms, and other features. The latter implementation is used for our simulations and applications.

6.1 Version 1: Event-Based Implementation

The initial implementation used a `select()` loop driven event model following the state machine from Figure 5.1. Development was on Linux (Debian and Red Hat) targeting the ROCKS cluster environment, which is the standard foundation for OptIPuter system software. We used the Gnu C++ compiler, Flex/Bison, the standard template and boost [35] libraries, and standard Unix/Posix sockets interface. Core code is about 5,000 lines of mixed C/C++ and a small amount of lex/yacc for parsing configuration files.

This prototype implementation was based on message passing and use of nonblocking calls. As proof of concept this was fine, but it had many limitations; lack of support for algorithms, interfaces, and other features discussed in prior chapters. Source code was not made publicly available.

The first limitation was that user programs could not be event based: two select loops in one program will generally not work. This is why other event-based libraries provide their own event model to which an application must conform. For example, `libasync` [77] and Globus XIO [84] use event registration and callback functions[1].

Second, the implementation was inherently not thread safe; it was designed around a single flow of control. It did not work well with the Globus XIO use model, and working around the problem with locks and thread serialization performed poorly.

Third, the code was complicated, hard to debug, and too fragile to use as a research platform. This resolves to the event model: any operation which might conceivably block required its own entrance point, generally a new function. It gives a large code base where related code is distributed undesirably– giving strange call traces. Program state must also be maintained across these call-back functions, by explicitly allocating them, passing them around in task-specific structures, and later deallocating them. Management of all this state in is complicated and difficult in C++.

[1]Libasync and Globus do not interoperate; as we wish to use Globus XIO we can not use libasync.

For example, consider reading raw network data and parsing it into variable-length structured data. Chunks of data depend upon those read earlier; as with reading the length of a structure, then its fields, which may also be of variable length. Implementing this requires multiple call-back functions– when logically it is a single serial operation.

6.2 Version 2: Threaded Implementation

The second, threaded implementation was developed from scratch to avoid the limitations of the event-based implementation. It immediately solved many of those problems. This version is provided as a shared library and several applications for common use cases. The library is used both for real-world tests, as the backend scheduler in our simulations, and as a tool to add CEP features to existing applications.

We handle scheduling for satellite/terrestrial transfers as a special case, described in Chapter 8. This code was implemented targeting simulation use only and is not in the library: we have no real-world access to satellites.

6.2.1 Version 2a: Threaded Stand-Alone Library

This version completely rewrites CEP internals using threads (pthreads) instead of events. This allowed us to refactor the code into a more rational object-oriented design. The implementation includes a number of enhancements, including:

- Cleaner and more robust code base; better separation between library, API, and user program code;
- Conversion to a dynamically linked, shared library;
- Support for multiple platforms, hardware: Multiple Linux Distributions, FreeBSD, Mac OS X;
- Support for Globus XIO [5] and GTP [126] (Section 6.3);

- Full API implementation (see Section 5.2);
- Full test suite with regression testing (see Section 6.4);
- Improved overall performance (see Chapter 7);
- Improved documentation.

This implementation does **not** include some features described earlier. We do not (1) perform rate-based transfers; we perform explicit transfers. Using stock TCP as an underlying transport protocol makes finely specified rate-based transfers effectively impossible. We do not (2) centrally collect network constraints; the scheduler uses NIC speeds to generate an explicit schedule and nodes locally optimize. We also do not (3) use dynamic maximum transfer size detection (Section 3.1.1) or (4) the interval tree structure (Section 3.1.2); instead we set the maximum transfer a priori and use lists in the segment graph.

For development we use the same Linux platform and tools as in the prior version, but the software works on any Unix-like system. The core code more than doubled to implement the new features. We also added several supplemental pieces of application code, scripts for common tasks, and documentation.

The threaded version avoids the problems encountered in the development of the event-based version by naturally keeping linearly executed code together. We integrate with user programs in a thread-safe manner, require less complicated state maintenance, and so forth. The event handling structure is now implemented by dispatching events to specific threads, and their structure takes care of the rest.

Another benefit is that this library implementation can be linked directly into a simulator, allowing us to use much of the same code for both real world and simulation tests. This helps validate our results. We use simulation for large-scale exploratory research; emulation or real-world tests are infeasible for some of our target environments, such as those with multiple 10Gbps links or satellite transponders. To test performance on such networks we use the `ns-2` [116] network simulator, version 2.29.

The only unusual feature implemented for simulation purposes was the addition of a small amount of randomized jitter. This avoids unrealistic global synchronization effects when setting timers, selecting peers, or dropping packets. This not a problem in the real-world, which by nature includes some nondeterminism.

This software was first made publicly available to users in 2005, via a Rocks [102] Roll package. It is part of the OptIPuter system software package [105], which is used for the development of future high performance grid software.

6.2.2 Version 2b: BitTorrent/Application Integration

One goal for this work was to support legacy applications– to be able to drop in the CEP library and improve performance with few other changes. This motivated the block-level extensions described in Section 5.2.5, and these were used to add transfer scheduling functionality to BitTorrent [29].

We chose BitTorrent as it is the most popular tool for content distribution– accounting for as much as 80% of the background traffic on the Internet [87] with approximately 50 implementations [124]. Of these, we selected the BitTornado client [2] for our implementation. While not the most popular client, it has source code available, reasonable baseline performance, and the features we wanted. BitTornado is written in Python, making exploratory modifications simple.

BitTornado and other current BitTorrent implementations manage metadata with a hybrid centralized/peer-to-peer approach. One or more *tracker* nodes maintains a list of peers and minimal state metadata. Peers fetch a list of other peers, and then individually exchange information on block locations. Data is transferred between peers directly using a local feedback mechanism.

Structurally, this meshes well with the CEP approach. With slightly more information, the tracker can perform the data scheduling task. Given a schedule, peers can continue using the feedback mechanism for downloads, provided they upload at a sufficient rate. Thus, half the implementation task was enhancing a BitTorrent client and tracker with mechanisms to support transfer scheduling, partial-sharing, and sub-file transfers. The other half was implementing the API extensions, algorithm tuning, and glueing everything together. These changes nearly capture the desired semantics, with the exception of using blocks instead of byte ranges.

With some BitTorrent clients now supporting selective file download, which induces partial sharing at the whole-file level, this work is particularly relevant. BitTornado supports selective download in a limited way: one may specify priorities for files in a set or disable their download. These priorities are merely suggestions. Blocks from disabled files may be downloaded anyway, because they are needed to checksum desired files. Low priority files may be downloaded first, because higher priority files' blocks are unavailable. This selective download code was expanded and rewritten to allow specification of specific block requests, enabling arbitrary partial-file downloads. It was also changed to enforce hard limits on block downloads.

We extended the tracker metadata management to determine locally optimal data transfers between peers using the greedy algorithm. Code to collect and provide peers with more useful information was also added (e.g. collecting network bandwidth measurements, and ensuring replies contain peers which can supply their requests).

Our results show that this approach is general, for arbitrary levels of sharing from disjoint to whole-file transfers; efficient, achieving low-latency and high capacity; and scalable, to thousands of nodes. It provides good results on high performance networks.

6.3 Networking Infrastructure

There are two independent network stacks in the threaded implementation, one for sockets and another based on Globus's eXtensible Input/Output Library [5]. XIO allows CEP to exploit a large number of protocols being developed within its framework. Internally, all CEP file and network input/output is done through a common IO class with a simple read/write interface; this hides the details of the individual transport protocols. We rely upon only the existence of a reliable, in-order protocol such as TCP.

As CEP's core contribution is efficient data transfer in high performance systems, the performance of the networking portion of the implementation fundamentally affects the performance results for the system as a whole. The rest of this section discussed the implementation in more detail; the sockets/XIO transport stacks and message format.

6.3.1 Network Stacks and Transport Protocols

CEP provides two separate network stacks internally, which are selected at compile-time by the user. The first is a sockets-based stack targeting portability, while the second is a Globus XIO-based stack targeting customizability.

Targeting portability limits the sockets stack to using TCP (the kernel's SOCK_-STREAM), even on systems which natively support other protocols such as FAST or SCTP. We also avoid OS-specific calls, e.g. `sendfile()`. Nonetheless, TCP's limitations mean that some tuning is required to achieve high performance: we must disable Nagel's algorithm and modify socket buffer sizes [90, 113].

This type of limitation is one reason why Globus XIO was developed. By providing a unified development framework and API under which *all* IO can be accomplished, the same code can be ported to a variety of systems with different underlying protocols. By utilizing the XIO network stack, we can leverage any reliable in-order I/O protocol XIO provides without concern for the details.

XIO provides a simple byte-stream interface with standard open-close-read-write (OCRW) semantics. It astonishing that such a library is required 40 years after Unix introduced the everything-is-a-file abstraction. Nonetheless, XIO is becoming the de facto standard interface for bleeding edge research protocols and access to high end storage devices. CEP's XIO stack allows the user to select the appropriate protocols for their application. As discussed earlier, we also export an XIO-*like* interface parallel to the sockets interface; unfortunately we can not provide exactly the XIO interface specification due to the semantic difference between point-to-point and many-to-many communication.

By implementing CEP using these approaches, we get the best of both worlds. We use the sockets version when Globus is not installed, is incorrectly installed, or when latency/overhead are an issue. It provides a basic, portable implementation. The Globus version is more powerful, but more complicated and less widely available. Its performs better on certain workloads due to internal buffer management and protocol optimizations, but worse on others for the same reasons. XIO also has some issues with overhead and it is difficult to tune protocols effectively through the simplified OCRW interface.

One important special case is CEP's support for GTP [126], the Group Transport Protocol. GTP is a many-to-one rate-based UDP scheme for fairly allocating download capacity across multiple flows. When fairness between flows is important, users can compile GTP rather than TCP for the sockets stack. Unfortunately, conflicting goals between the scheduling techniques used by CEP and GTP tend to produce poor performance– CEP sees fair flows as underperforming, and tries to schedule around them as bottlenecks.

Lastly, these low-level interfaces are completely separate from the APIs provided to users (Chapter 5). Those APIs are available regardless of the underlying technology used by CEP to transport data and metadata.

6.3.2 Message Format

We introduced the main CEP messages in Figure 5.1 and Table 5.2. To implement that state machine, we use the simple message format shown in Table 6.1 (page 102). All messages begin with four common fields, and then one or more fields specific to each message type. For efficiency reasons, we use a custom binary message format rather than a text transfer mechanism (as with FTP or XML-based protocols), and perform serialization/de-serialization directly rather than via an external library.

We have one field for the message type (e.g. "spawn reader"), one to identify the database (the logical flow to which we are referring), and then the range within that database this message concerns. We use a large field for the message type to keep the remaining fields 32-bit aligned, to leave room for future extensions, and to allow fast type parsing.

6.4 Engineering Issues

This work focuses on the interesting research questions surrounding the problem of transfer scheduling. However, the way we address the plethora of engineering issues when developing a large piece of software directly affects the quality of our results. This section discusses our regression testing framework and how we handle failures.

6.4.1 Infrastructure and Platforms

CEP's target platform is the ubiquitous 32-bit x86 architecture, running Linux. We also target 64-bit linux on Xeon/Opteron and OS X on PowerPC; these are found in some OptIPuter storage and visualization clusters. After addressing issues with endian-ness, word-length, differences between "standard" libraries, the core code runs on all these architectures. GTP and Globus do not support 64-bit or PPC architectures, so so that code is 32-bit x86-specific. In other words, full functionality available only on Linux/x86; elsewhere only the TCP transport can be used.

Table 6.1: Individual Message Data Fields

Message Type		
	Size	**Field**
All Messages		
	4 Bytes	Message Type
	8 Bytes	Database Identifier (Flow ID)
	8 Bytes	Start byte in global namespace
	8 Bytes	End byte in global namespace
Reader Spawn, Writer Spawn Messages		
	4 Bytes	Peer ID length
	2 Bytes	Port
	Variable	Peer ID (Name or string IP address)
	8 Bytes	Offset within file
	4 Bytes	File name length
	Variable	File name
Get, Put, Take Messages		
	4 Bytes	Peer ID length
	2 Bytes	Port
	Variable	Peer ID (Name or string IP address)
	Variable	Other metadata (transfer rate, etc.)
Write Message		
	Variable	Binary data. Length known from message start/end byte.
Read and Quit Messages		
	0 Bytes	No extra data. Peer is implicit in transfer state.

After the initial development effort, we created a series of regression tests to verify common CEP functionality. These are listed in Table 6.2. These test specific features from the basic– does it compile, does it tolerate files larger than 2^{32} bytes– to the complex– can it optimally schedule and execute a transfer with no 1-1 mappings.

Finally we assembled a regression testing framework for running all tests after any change to the CEP code to detect whether bugs had been introduced. A back-end database and PHP-based web frontend provide a useful way to check results and progress. This code has been running since early 2006 and has helped ensure the results shown in this document are consistent and representative even with evolution of the codebase.

Table 6.2: CEP Regression Tests

Test	**Purpose**
Null	Does nothing. Does the regression code work?
Compilation	Does CEP compile in the given configuration?
Small file	Does a 1-1 small file (≈2KB) transfer work?
Medium file	Does a 1-1 medium file (≈100MB) transfer work?
Large file	Does a 1-1 large file (≈5GB) transfer work?
1 server 4 client	Does a 1-4 scatter (disjoint data fetch) work?
4 server 1 client	Does a 4-1 gather (disjoint data assemble) work?
3 server 5 client	Does a many-to-many overlapping data transfer work?
Multiple file	Do serial transfers (proper state cleanup) work?

6.4.2 Handling Failures

There are three classes of failures in a collective transfer system: scheduler, server, and client failures. Our implementation tolerates only server failure, the second class.

Surviving scheduler failure after peers have received their metadata is relatively simple; however we have chosen not to support this. Instead, killing the scheduler in the current implementation cleanly terminates transfers on remote nodes. This allows the user to stop the distributed transfer easily without additional infrastructure or commands.

Server failure is survivable when two or more servers replicate the same data or when the data they serve is not required by any client. In either case, the aggregate transfer can still complete successfully. In our implementation, such failures are transparent to the user– so long as at least one replica exists for desired data, the transfer will continue. Section 7.7.1 looks at the performance of transfers under failure conditions.

To detect these failures, we rely heavily on the transport layer– i.e. TCP– reporting lost connections or failing to connect. In the best case, a RESET packet will be sent allowing for immediate recovery. In the worst case, a series of timeouts must occur. This may take on the order of minutes.

Client failure normally means that the transfer will not complete. As we are interested in aggregate termination, failure of any part means by definition that the aggregate fails. The only exception is when using the weakly constrained transfer extension– which provides sufficient semantic flexibility to allow use of other peers for recovery.

We chose not to support this due to problems detecting that a failure had even occurred. Once work is given to peers, the scheduler has no efficient way to do so. The best options are timeouts on “heartbeat” messages or at the expected completion time. Heartbeats can become very expensive for large numbers of peers. A single timeout is cheap, but by waiting so long we multiply the transfer time for each peer that fails.

6.5 Summary and Conclusion

This chapter has explained the structure of the two main implementations. The next chapter, the longest in this dissertation, uses these implementations to experimentally validate our ideas and analysis. This background detail puts our performance into context and helps prove its validity.

Acknowledgements: Material from this chapter, in part, appeared in “The Composite Endpoint Protocol (CEP): Scalable Endpoints for Terabit Flows,” Eric Weigle and Andrew A. Chien, Proceedings of IEEE Conference on Cluster Computing and the Grid (CCGRID), 2005. The dissertation author was the primary investigator and author of this paper.

Chapter 7: Evaluation

This chapter presents experiments and analysis exploring our performance claims: *scalability*, *efficiency*, *high performance*, *robustness*, and *generality*. It is roughly separated into two parts; the first part covers performance of core features, while the second part covers performance in a specific content distribution application, BitTorrent.

The first part begins with an overview of our evaluation approach, then some baseline performance measures. We show linear system *scalability* on cluster environments, achieving up to 30Gbps in local clusters and over 10Gbps in the wide-area. After that, we look at microbenchmarks and results in a variety of other environments; showing over 100× more *efficient* computation using our Greedy algorithm than the baseline LP algorithm. We see *high performance* including 40× faster transaction processing than Apache, and 4-5× higher bandwidth on heterogeneous configurations than uniform striping (as in GridFTP). Finally, we show we are *robust*; we have only a 2% performance penalty under real world server failures and negligible performance penalty with inaccurate metadata.

The second part looks at content distribution scenarios in the context of peer-to-peer transfers with our BitTorrent/CEP hybrid. We quantify performance under a variety of partial and whole-file sharing configurations. We again show *high performance* and *efficiency*, up to 8× higher bandwidth and 10× lower latency than BitTorrent; *robustness*, succeeding when 20% of BitTorrent nodes fail; and *generality*, exceeding BitTorrent performance under a variety of user constraints.

7.1 Overview and Approach

While Chapters 3 and 4 supported our claims analytically, here we focus on empirical evidence. Using the implementations developed in Chapter 6, we exhibit the features and performance we have claimed on real world, emulated, and simulated networks.

Our real-world tests were performed on the typical high performance clusters listed in Table 7.1. “High performance” refers to x86-based machines with 1Gbps or faster links in the LAN and 10Gbps in the WAN. The largest cluster, FWGrid [89], has over 300 nodes (only 128 can be used at once) accessed through a batch scheduling interface. This cluster has 32-node racks with switched 1Gbps Ethernet, and 10Gbps Ethernet between racks.

Table 7.1: Available Cluster Hardware

Name	**Nodes**	**Network**	**CPU ($\times$2)**	**Memory**
csag-slow	32	100Mb	450MHz Pentium-II	1GB
csag-fast	24	1Gb	2.4GHz Xeon	2GB
fwgrid-opteron	94	2$\times$1Gb	1.6GHz Opteron	2GB
fwgrid-dell-1	64	2$\times$1Gb	2.8GHz Xeon	4GB
fwgrid-dell-2	160	2$\times$1Gb	3.2GHz Xeon	4GB

For wide area tests, such real-world infrastructure has been unreliable and access to remote resources has been otherwise problematic. Instead, we turn to emulation and simulation. For emulation we use DummyNet [99] to create virtual high-latency links between nodes on the csag-fast cluster. Unfortunately, emulation scales poorly in terms of raw bandwidth. Even using recent developments with Xen [9] and time dilation [50] emulation of many 10Gbps Ethernet links is infeasible.

We use `ns-2` [116] simulation for our largest tests. We validate these simulations through several mechanisms to gain confidence that they accurately capture salient problem features. First, we reproduce real-world experiments in simulation, verifying there are no disparities. Second, results match theoretical predictions from earlier chapters. Third, manual analysis matches simulation output on small configurations. Finally, the `ns` validation suite, which tests core features (e.g. TCP behavior), reports no problems.

7.2 Baseline Performance

This section evaluates the performance of underlying technology used by CEP. All the results we present here use TCP as a transport, so understanding its performance characteristics on this hardware is important. Similarly, tests in Sections 7.8-7.9 use a Python application, so understanding the Python network stack and related operations is important.

7.2.1 TCP Performance

Understanding TCP performance characteristics is necessary to understand the CEP results we present below. This section provides data for stock TCP in Linux and in the ns-2 simulator. Two programs are used to test on the live network: `iperf` [82] and a custom Python benchmarking tool. We use Dummynet to emulate higher delay links. Figure 7.1 shows the bandwidth achieved for a 10 second transfer under various network delays.

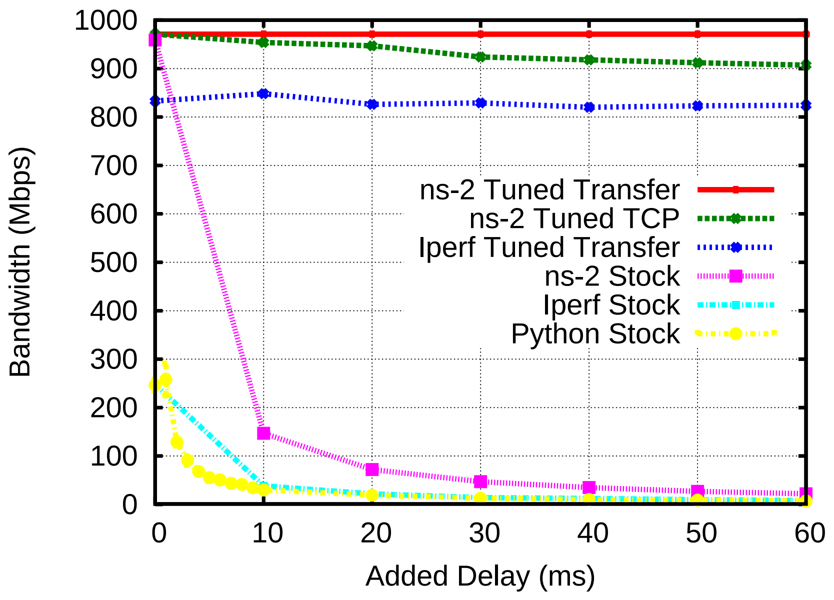


Figure 7.1: TCP performance in the LAN/WAN

The bottom three lines show the performance of TCP without tuning. These exhibit TCP's well known performance problems in the wide area [68, 114, 119]. Its flow and congestion control feedback mechanisms give poor performance when the $bandwidth \times delay$ product is large. For example, performance falls by a factor of 100 with only 20ms of latency added through DummyNet. Also note that DummyNet, even configured to add zero delay, still cuts performance by over 70%: stock TCP achieves 890Mbps directly, but only 250Mbps through DummyNet.

Achieving good performance on even faster or higher delay links is more difficult [25, 90, 113]. The top three lines show TCP performance with tuning. By increasing TCP buffer sizes, using parallel flows, and increasing the length of the transfer, we can maintain performance under increasing delay. the two "Tuned Transfer" lines show this behavior. In contrast, if we limit the flow length to 10 seconds as before, TCP's ramp-up time becomes an increasing proportion of the total transfer time, and performance falls slowly as delay increases– the "Tuned TCP" line.

Also note the 150Mbps difference between the performance of tuned transfers in theory (ns-2 simulation) and practice (Iperf). Even with tuning, these machines cannot achieve more than around 900Mbps (90% of the 1Gbps theoretical) due to hardware and software limitations. This is not uncommon, particularly for machines running a firewall or inexpensive NICs which perform some operations in software.

In terms of CPU load, Iperf tests used approximately 20% of the CPU and Python tests used approximately 13%. The difference is due to the Python benchmarking tool caching transfer buffers which Iperf regenerates. `ns-2` does not model CPU load.

7.2.2 Python Performance

Python, as a partially interpreted language, may seem inappropriate for high performance tasks. However, our results show transfer speeds comparable to native C code. There is only one critical aspect to consider: buffer management. Typically strings, which are an immutable type in Python, are used as buffers. Therefore common buffer modifications– such as adding headers or reassembling fragmented data– are potentially expensive.

Using a node from the csag-fast cluster, we look at the speed of concatenating 8KB buffers. Figure 7.2 shows the results using two algorithms: “iterative” joins buffers together one-by-one while “one-shot” collects buffers into a list and joins them with a single call (to `join`). We see that for older versions of Python this processing can be a significant but avoidable performance bottleneck.

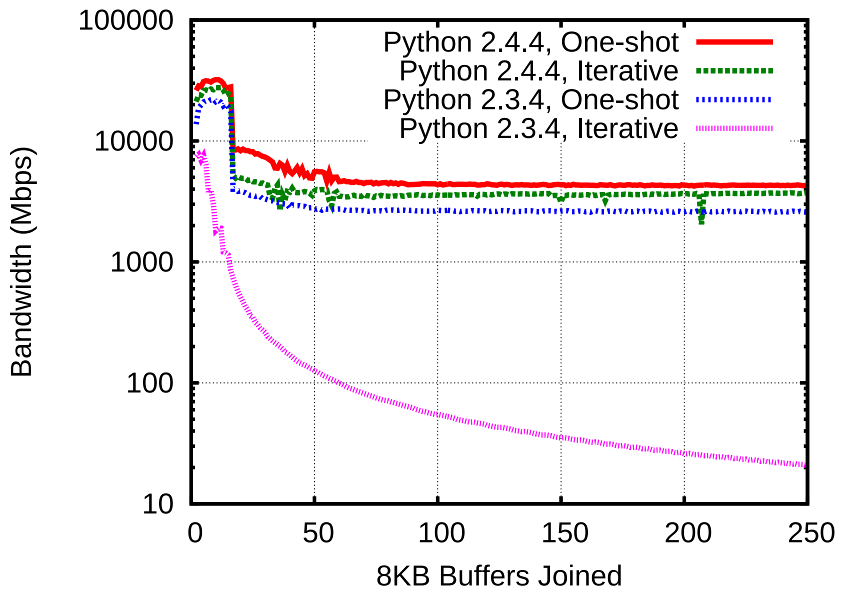


Figure 7.2: Performance of Python Buffer Concatenation

7.3 Node Scalability: Composition

Our motivating examples (Section 2.2) included high speed transfer between clusters of computers and large-scale content distribution. This section examines CEP performance in such environments; in particular, aggregate bandwidth. This is our most basic goal: being able to scalably compose multiple nodes' flows into a single logical connection.

We test transfers using various numbers of homogeneous nodes to see how well the system performs. Input constraints are set such that nodes will equally share load: node n_i transfers bytes $[\lfloor \frac{i}{n} total \rfloor ... \lfloor \frac{i+1}{n} total \rfloor - 1]$. This is the schedule produced by the weakly constrained transfer extension from Section 5.2.4.

7.3.1 Exploiting Local-Area Nodes

First we evaluate local area performance, as between clusters on a single campus. We set the total amount of data sent, $total$, such that each node transfers 2 Gigabytes of data user-level memory to user-level memory. Figure 7.3 graphs the aggregate bandwidth CEP achieves on the fwgrid-opteron cluster and in simulation, for various numbers of nodes.

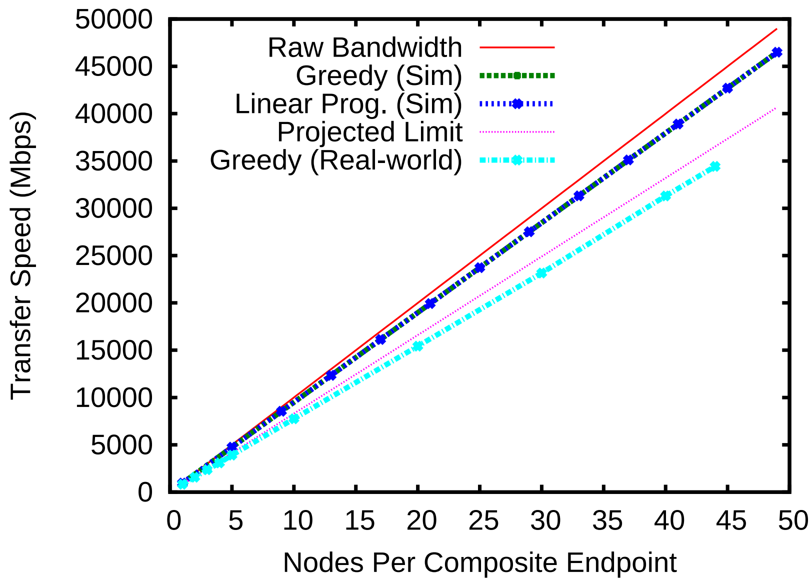


Figure 7.3: CEP Composition Efficiency

The “Raw Bandwidth” and “Projected Limit” lines are for comparison purposes. Raw bandwidth is the unattainable maximum capacity, ignoring necessary overhead such as packet headers. The projected limit is the ideal capacity were all nodes to run at the rate ($\approx$ 830Mbps) achieved by the benchmarks in Section 7.2.1.

In simulation, the Greedy and LP algorithms produce identical results. Performance increases linearly with the addition of nodes, running at approximately 96% efficiency. The remainder is due to packet headers– 40B/1KB = 4% overhead.

Performance is slightly below the projected value in the real world, but still grows linearly. The difference is due to asynchrony between transfer termination and overhead in scheduler communication. Unfortunately, we can test at most 44 node pairs (88 nodes) in the real world due to contention for cluster resources.[1] Unsurprisingly, in simulation these results scale linearly as high as one cares to test; we have observed up to 1Tbps.

In terms of load, during these tests CPU load was approximately 60% on cluster nodes. CPU load on the scheduler node was less than 1%. Memory utilized is proportional to number of flows and a user-specified amount of precached data; for these experiments it is about 32MB per node. While not a limiting factor, CPU load is is three times that of Iperf. Further examination shows that much of this is due to an expensive data randomization operation; transport buffers are filled with random bytes for this benchmark. Without this, load falls to approximately 30% but bandwidth achieved is unchanged. The remaining ‘extra’ CPU overhead is likely due to thread contention.

[1]Of the 94 nodes, 6 were misconfigured limiting us to 88 nodes. Larger numbers could not be concurrently allocated for our tests.

7.3.2 Exploiting Wide-Area Nodes

The last section showed good performance on local-area networks. Next we show the performance a very wide-area network: the OptIPuter [117] WAN. This is a computational grid environment. Our results shows that we can exploit high capacity links (10Gbps) to transfer large amounts of data. Due to problems with the physical hardware, we present simulation results for this environment. Figure 7.4 shows the relevant portion of the network we model.

For this test we use the clusters in San Diego (csag-fast cluster), Chicago ('Scylla' cluster), and Amsterdam ('vangogh' cluster). These are the sites which have shared computational resources available. The Scylla and Van Gogh clusters are roughly equivalent in size and configuration to the csag cluster described earlier, but in simulation the only relevant characteristic is nodes' network speeds: 1Gbps Ethernet per node.

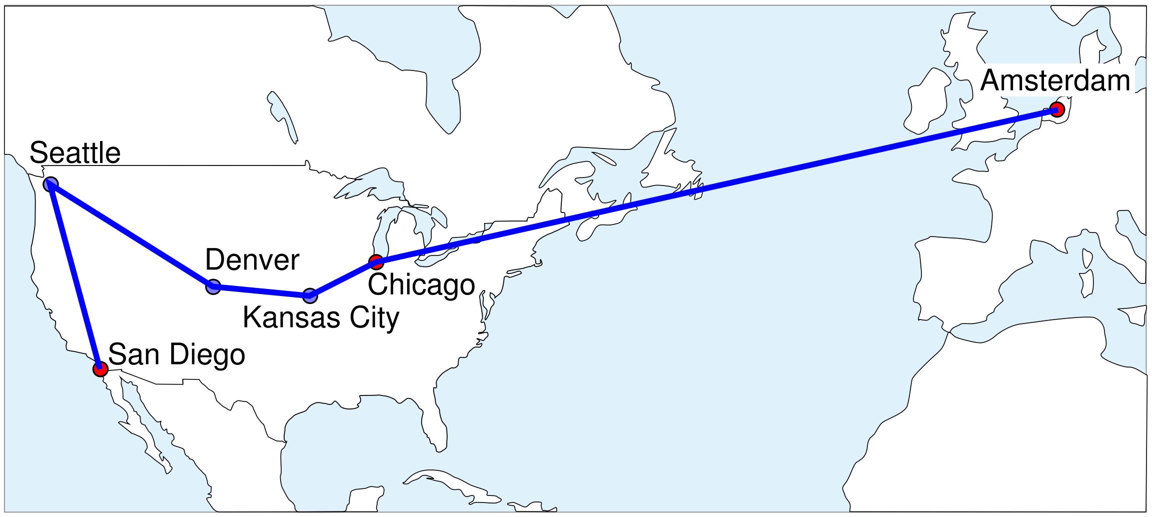


Figure 7.4: The OptIPuter Wide-Area Network

The San Diego $\leftrightarrow$ Chicago link is 10Gbps with 60ms round-trip latency, while the Chicago $\leftrightarrow$ Amsterdam link is 10Gbps with 103ms round-trip latency. This is approximately 100 times the latency of the network in the previous section, due to unavoidable speed-of-light and queuing delays over such long distances.

We configure a transfer as in the previous section; cluster-to-cluster transfer with servers in San Diego and Amsterdam, and clients in Chicago. The theoretical max transfer rate is therefore 20Gbps: 10Gbps aggregated over each server cluster. We use 16 nodes in San Diego and Amsterdam, 32 nodes in Chicago. In aggregate we have two "barbell" shaped cluster-to-cluster transfers, with bottleneck of 10Gbps. Figure 7.5 shows the results for transfers of varying size.

The peak rate experienced is about half that theoretically possible. The average transfer rate as determined by the last flow to finish is about an eighth of the theoretical maximum. While undesirable, this is actually exactly what we expect due to the behavior of stock TCP in the WAN (see Section 7.2.1). The problem is that a few flows encounter losses and take a long time to recover; dragging the aggregate performance down. Unfortunately, increased parallelism can not help this due to the time it takes for TCP windows to open.

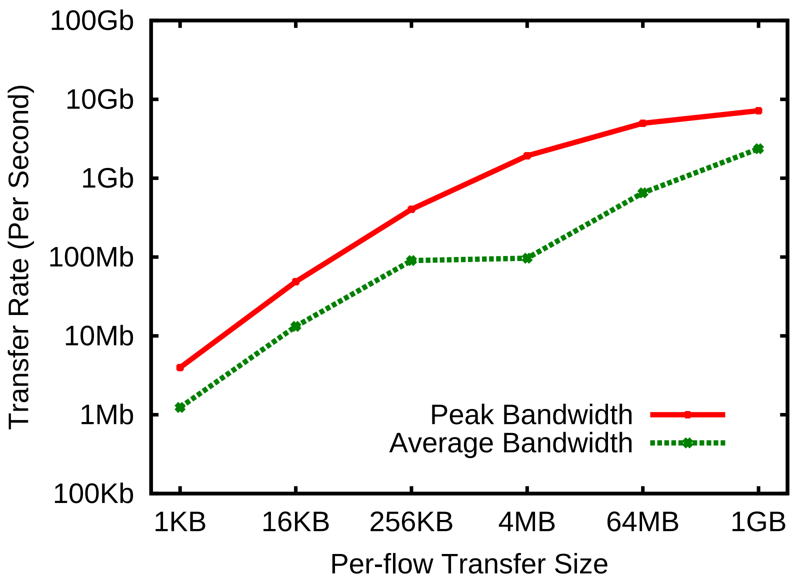


Figure 7.5: Exploiting Wide-Area Networks with CEP

Consider the Chicago-Amsterdam link. A link at 10Gbps with 100ms delay means a 125MB bandwidth-delay product. With 1KB packets, saturating such a link means senders must have 125,000 packets "in flight" at any given time. Say a single TCP flow were attempting to saturate such a link. A loss cuts the congestion window in half (62,500 packets) and additive increase must recover at one packet per RTT. That's 6,250 seconds to recover: an hour and 45 minutes.

Saturating the link therefore requires uncommonly long (10s of terabytes) transfers, parallel flows, and large buffers (up to the bandwidth-delay product) throughout the network. Parallel flows divide the recovery time and cost of a single packet loss, but tend to produce burstier traffic and induce correlated losses as router queues overflow. Beyond such tuning there is little we can do to improve performance while using TCP as the underlying transport. On such high bandwidth-delay networks, protocols such as HS-TCP [41], FAST [56], UDT [49], or RBUDP [52] are preferable. Unfortunately, they are not widely supported in the real world. See section 9.4.1 for more information.

In summary, these two sets of results demonstrate the scalability results we expected based on our high-level analysis and baseline TCP performance. To justify claims for larger number of nodes we must turn to other benchmarking techniques.

7.4 CPU Scalability: Computation Cost

This section complements the prior by showing CPU scalability. We measure the run time of our scheduling algorithms/infrastructure. We test schedulers on a trivial input problem: simple striping. That is, p nodes have data and another p want that data. An optimal solution is any set of 1-1 node pairings (or corresponding solutions at the segment level). As greedy scheduling time is invariant under data layout changes, this input favors other schedulers. It is trivial to solve by hand and should be easy for e.g. the LP library to solve. Figure 7.6 show the run-time of algorithms on a 3.2GHz Pentium 4 processor. We include best-fit curves for some cases, as testing them became infeasible.

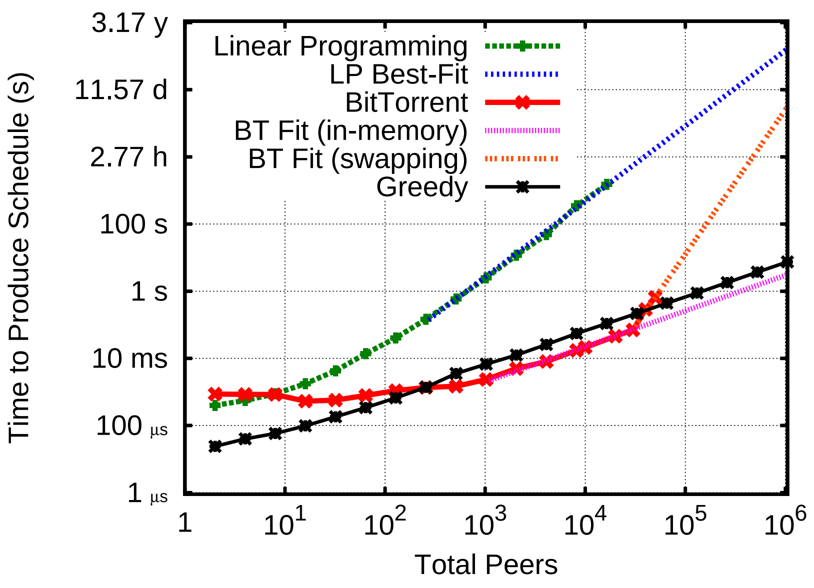


Figure 7.6: Scheduling Cost vs. Number of Nodes

We are most interested in the LP and Greedy lines: the two main algorithms implemented in CEP. The LP algorithm performs much worse than expected. The results of solving the 8192-node equations took several minutes, which is unacceptable for any but the longest transfers. This is particularly true as the transfer solution is no better than the greedy algorithm's schedule. That is, the time to implement the transfer given either schedule is the same. Note that the absolute time to generate a greedy schedule is less than 100 milliseconds for up to 10,000 nodes, our target scaling goal. This is less than the RTT in the current OptIPuter WAN (103ms San Diego↔Amsterdam, as described in Section 7.3.2).

The remaining curves are for BitTorrent– which does implicit scheduling. This resolves to selecting random peers (at the tracker) and maintaining sorted lists of communication partners (at the peer). For the implementation discussed in Section 6.2.2, these operations are efficient up to 50,000 nodes. Beyond that, overhead of automatically sized and type-tagged data structures use too much memory. Over 1.2GB is used for the 50,000 node case, causing the machine to go into swap. Minor implementation changes could easily avoid this to give better results for large problem instances.

Lastly, this graph can also be interpreted as scalability in terms of segments; there is one segment per peer in the input constraints. If we instead schedule two peers and vary the number of segments, we get the same results. Altogether, CPU capacity is not a limitation for our target environments – provided we use the greedy scheduler.

7.5 Network Scalability: Metadata and Bounds

This section breaks down the network features limiting scalability. This resolves to metadata limitations for the whole system, as data transfer bounds can be avoided by adding more nodes. We walk through the timeline for a CEP transfer, show that the scheduler can theoretically support over 100k transfers on a 1Gbps link, and present network benchmarking results supporting these values.

7.5.1 Transfer Timeline

This section shows that, unsurprisingly, metadata transfer time is insignificant compared to data transfer time for large transfers. Given the CEP architecture, even slow centralized schedulers have sufficient bandwidth to support large numbers of peers. Table 7.2 shows the timeline of events during a CEP transfer and overhead involved. This is for the centralized 'push' use model.

Table 7.2: Transfer Time Analysis (milliseconds)

Time (ms)	**Percent**	**Task**
3-5 rtt ≈ 50	.28	Spawn remote client/server threads
1 rtt ≈ 10	.06	Request scheduling data, peers $\rightarrow$ master
$n/100$ ≈ 1	.0056	Calculate schedule, See Section 7.4
1 rtt ≈ 10	.06	Reply with schedule, master $\rightarrow$ peers
3-5 rtt ≈ 50	.28	Peers connect to each other
$\approx$17650	99.26	Lower bound data transfer time; 2GB @ 1Gbps Ethernet
1 rtt ≈ 10	.06	Update master; termination message
$\approx$ 17782	100	Total

First, the scheduler spawns peers on all machines. This takes around 4 round-trip times; time for TCP handshaking plus data send/acknowledgement. Peers then request orders from the master and wait for its reply. Finally they perform the desired transfer and tell the master when they finish.

The important things to note are that (1) data transfer dominates the total transfer time and (2) almost all events occur in parallel. Peers are spawned in parallel, possibly at the same time scheduling is done. Peers can be spawned with immediate transfer tasks. Peers can begin transfers before others have been spawned. The only ordering constraints are that scheduling must occur before peers receive their schedule, and individual peers must start, receive their schedule, transfer, and then stop.

7.5.2 Calculating Bandwidth Required

The analysis above assumed the scheduler had sufficient capacity– only network delay mattered. Now we show bounds on bandwidth required by the scheduler node: it can theoretically manage over 120,000 peers/second using a 1Gbps link. Peers have no related scalability issues; they generally communicate with only a handful of other nodes.

Setting up a TCP connection typically requires three 40-byte packets, and peer metadata is around 60 bytes per record. The spawn, request/reply, and termination each carry the record. In sum, this is less than 1KB/peer for a simple striped transfer (1 record per peer). This network bandwidth will not be a problem for our target environments- a 1Gbps can transfer over 120,000 1KB records per second.

In more detail, a data constraint record requires: an 8 byte transfer ID, 2×8-byte integers specifying the range, and the name/port of the peer. This is commonly around 60 bytes, depending on the peer name length. With multiple records/peer, we can amortize that cost and asymptotically approach 16 bytes (begin/end of range) per record. Network constraint records are similar, about 70 bytes: two peer names and 4 bytes for bandwidth. However, the centralized scheduler does not collect this information in these experiments.

All data collected needs to be stored. If each record averages about 60 bytes, 10,000 records only take 600KB. Including management overhead, this may grow to 1MB. This memory is obviously not going to be a system bottleneck. On the other hand, kernel memory is also used for network buffers. By default, each TCP socket can claim up to 64KB of memory. This can be tuned down to 16KB, which is enough for 16,000 peers in 256MB of memory. Other tuning is required to handle this many concurrent TCP connections in practice [60]. An alternative approach is use of UDP with manually added reliability. In either case, memory will not be a problem for our target environments.

7.5.3 Testing Network Scalability

To physically test the networking portion of the centralized scheduler, we created a benchmarking client. It makes metadata requests but does no actual data transfer. For comparison purposes we include benchmark data for Apache [44] and the Pastry DHT [101].

Apache results are Apache 2.0.58 benchmark data. This is useful as one alternative to CEP's custom scheduler is a web service implemented using commodity software; a reasonable choice would be Apache, the most commonly used web server. We use the Apache 2.0.52 server with up to 15 preforked processes and the ApacheBench client version 2.0.41.

Pastry results are from benchmarks of a simple metadata server written using the MACE [62] implementation of Pastry. Multiple peers form the metadata "server" and clients make requests as before. This is another potential alternative approach as described in Section 5.3.3. Note that clients do *not* themselves join the overlay. This naive approach performs very poorly– up to 100$\times$ worse than the results we present here– as lookup costs grow with the size of the DHT and keys are shuffled among nodes.

Figure 7.7 shows the transaction processing (read) rate for CEP, Apache, and Pastry between csag-fast cluster nodes. We perform five trials, with 5000 requests of 64-bytes each. Tests stop at a concurrency of 1020 or lower due to file descriptor/thread limitations.

This figure shows CEP performance comparable to the rough 120,000 estimate above. CEP performance falls after 64 due to thread management limitations. Apache performance improves and stays roughly stable as the concurrency grows; a desirable property. Pastry performance follows roughly the same pattern. For larger DHTs (i.e. the 64-node line shown here) performance falls due to increased lookup cost.

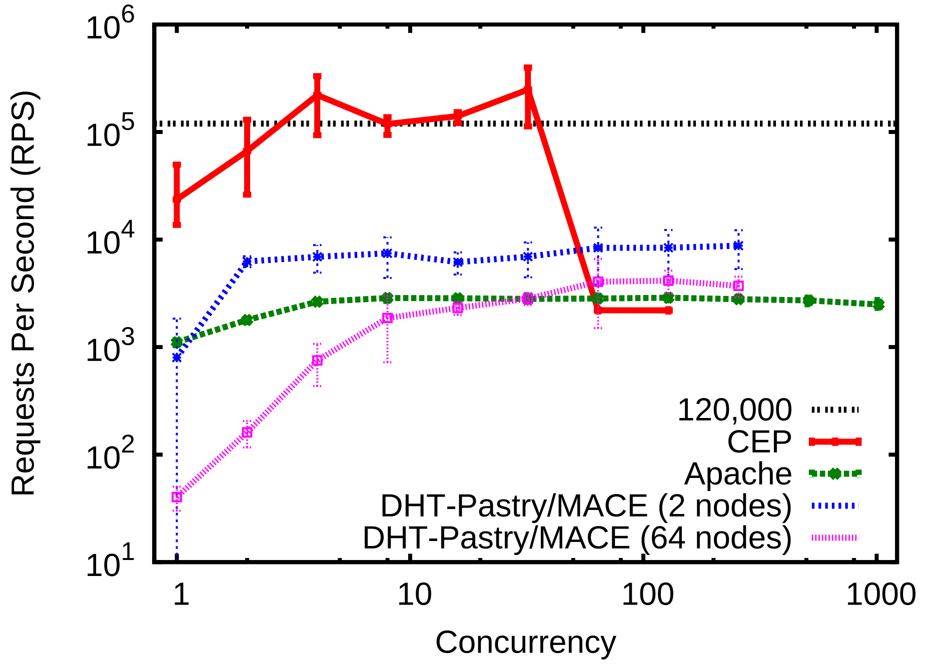


Figure 7.7: Transaction Processing in CEP, Apache, and Pastry

We have also tested with other DHT implementations. First was OpenDHT [96], a service-oriented implementation of Bamboo running on PlanetLab [91]. Unfortunately, the system is heavily used; storing or retrieving even small values was unreliable due to capacity limitations. It was also “embarrassingly slow” [95], taking on average 7.25 seconds to retrieve a value over a dozen trials.

MIT’s Chord [108] implementation was also not suitable for our purposes. By default it uses erasure codes; storing a single block requires saving data to 16 nodes. This provides excellent robustness under failure but at the cost of performance.

Together, these results show the desired scalability in terms of bandwidth. A single scheduler can handle large numbers of metadata requests. Large numbers of peers can be serviced with low latency provided their requests are not synchronized– ideally no more than 64 requests occur at once.

The last several sections have shown that system capacity– CPU, Memory, and network capacity– are sufficient to scale to desired problem sizes. Environments with up to tens of thousands of peers can be handled on commodity hardware, as we have claimed. Now we turn to our claims of generality and high resolution showing that CEP can handle more complex constraints.

7.6 Exploiting Widely Varied Constraints

This section shows the generality of our approach by evaluation on a wide variety of environments. CEP performs well regardless of node capacity, network capacity, or data constraints. First we show the effects of heterogeneous nodes– we can exploit all nodes efficiently, regardless of their capabilities. Then we show the effects of heterogeneous data access– we can exploit differing data layouts to optimize performance.

7.6.1 Exploiting Node Heterogeneity

To show the advantages of utilizing heterogeneous nodes rather than enforcing one to use a homogeneous cluster, we test CEP versus the baseline uniform striping mechanism that tools like GridFTP [4] use.

Using 16 total nodes, 10 taken from the csag-slow cluster and 6 from the csag-fast cluster, we perform a 1GB transfer. Using the weakly-constrained transfer extension, half of the nodes from each set are allocated as senders and the other half receivers. We begin by testing transfers using just the slower set until we run out, then we allocate nodes from the faster set. Figure 7.8 shows the bandwidth achieved in this transfer.

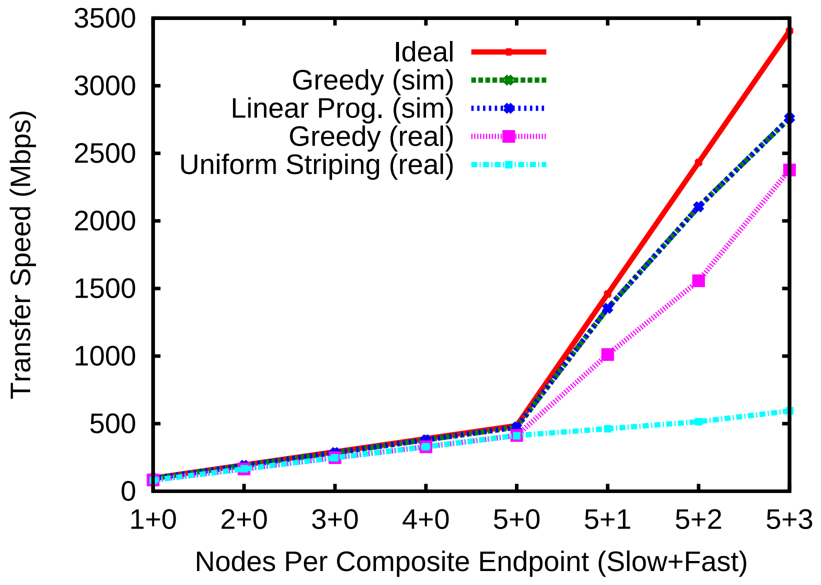


Figure 7.8: Exploiting Node Heterogeneity with CEP

The CEP and uniform striping schemes give identical performance when nodes are homogeneous. When faster nodes are added (at point “5+1” referring to all five slow senders plus one fast sender in use), the greedy/LP schedulers achieve far better performance. We effectively shift to the performance curve for the faster nodes. The simple, uniform striping scheme remains limited by the slowest node used. Seen another way, we can enforce fairness by providing tight constraints. But given transfer flexibility, we can optimize– in fact we need the flexibility to enable optimization with heterogeneity.

The simulations reveal diminishing returns as the number of nodes increases. That is, node management and scheduling overhead become a larger proportion of the transfer time as the number of nodes grows. This was lost in the noise of the real-world tests.

One side effect of this behavior is that we need not worry about accurately allocating nodes based on a priori knowledge. With CEP a user can attempt a transfer, and if performance is inadequate, simply add resources until performance is acceptable. This addition can be done without concern for details of homogeneity, e.g. whether the additional resources are fast new nodes or a much slower cluster from several years ago.

7.6.2 Exploiting Data Access Flexibility

Prior tests used the weakly constrained transfer extension, i.e. assumed a fixed level of data access, to allow optimization under varied node heterogeneity. Now we look at the other half of the problem: given a fixed set of nodes, how to optimize under varying levels of data access. Logically, the more server replicas, the more optimization potential. Zipf-like distributions are common due to changing "hot"/popular data over time.

Given a fixed set of 8 heterogeneous nodes, four each from the csag-slow and csag-fast clusters, we explore two different data layout schemes. Each varies the amount of data each node may access from disjoint- $1GB/n$ per node, to total- the whole $1GB$ per node.

The first layout sets allocations serially, starting with the faster nodes. Thus overlap falls only on the slower nodes. This is the optimal layout, as the fast nodes can be maximally exploited in low-sharing environments, but is somewhat unrealistic. Figure 7.9 shows the results for this layout scheme.

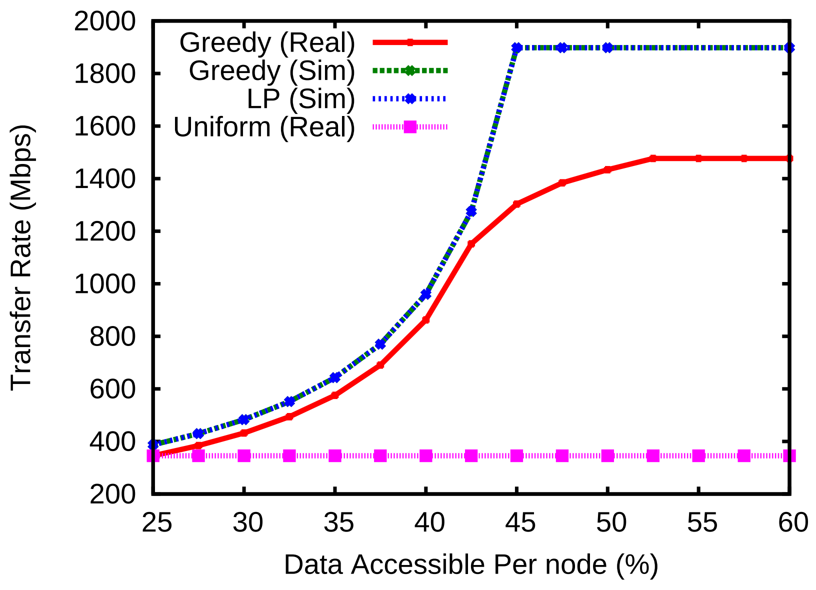
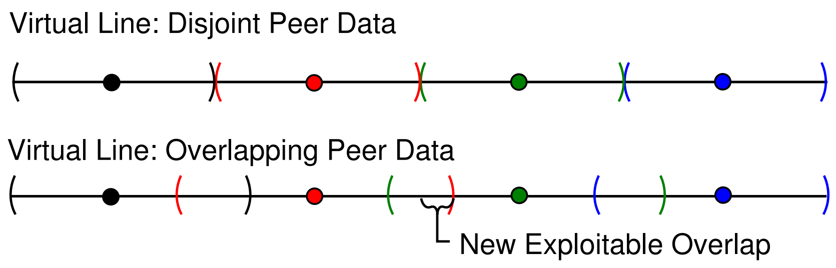


Figure 7.9: Exploiting Data Access Flexibility with CEP

With uniform striping, regardless of the data access flexibility, performance is the same. With flexibility, the more overlap, the higher performance. Improvement flattens out as the hardware is saturated. This gain is due to the scheduler shifting work to more powerful nodes until all nodes are equally saturated. Beyond that point, additional flexibility cannot give additional performance.

The second layout sets allocations equally. First we spread each of p peers p_i along on a virtual line of length s (s=total transfer size). A peer's data request is centered at $c = (i + 1/2) * (s/p)$. We then vary the amount of data, d, each peer downloads from 1 to $2s$. For a given d each peer p_i downloads from $max(0, c - d/2)$ to $min(c + d/2, s - 1)$. For $d < (s/p)$ these are disjoint; for larger d peers' requests have increasing overlap.


Virtual Line: Disjoint Peer Data
Virtual Line: Overlapping Peer Data
New Exploitable Overlap

The performance of transfers using this second layout is a linear transformation of the results above. In particular, it stretches the graph horizontally. The only value that matters is how much of the data the fast nodes can access- in this layout that is exactly the upper bound of the second fast node's data: $c + \frac{d}{2}$. As we increase the percent of data accessible, only half of the second nodes' allocation newly overlaps that of the slow nodes. Thus performance improvements are one quarter those observed in the first layout.

At a higher level, we learn two things from these results. First, that CEP can capture both explicit and flexible user constraints– forcing the scheduler to transfer use uniform striping, or allowing it to allocate capacity more intelligently. Second, that the weakly-constrained transfer extension is exactly the 100% flexibility point at the right side of this graph. On homogeneous nodes, the scheduler ignores that flexibility and produces a striped layout. On heterogeneous nodes, the scheduler optimizes using node capacity.

7.7 Performing Under Stress

This section determines the performance of CEP when things go wrong. In particular, we look at the effects of node failure on transfer performance, and how inaccurate data affects transfer scheduling. In particular these address the claims made in Section 4.3.3 that nodes can make progress under failure and that we tolerate inaccurate information.

7.7.1 Tolerating Node Failure

This section looks at the effectiveness of failure recovery as discussed in Section 6.4.2. Performance under failure obviously depends greatly on the type of failure. As we are interested in the mechanism itself, we focus on a simple configuration which should ideally show little or no performance degradation under failure.

This configuration consists of a receiver-limited transfer; for a set of disjoint segments, two servers completely replicate each segment, and one receiver desires each segment. All nodes are homogeneous; thus servers are only on average half utilized, and the failure of either server in a pair should be recoverable without affecting performance.

Using 16 nodes (10 servers, 5 clients, 1 scheduler) from the csag-fast cluster, we configure a transfer such that each receiver downloads 2GB of data. After starting the transfer, we manually kill from zero to half the servers. Figure 7.10 shows the results.

As desired, performance is roughly constant regardless of the number of servers lost. Any transfers in progress to a dead node receive a TCP reset message, and they immediately shift their work to the next available replica. This failover occurs on the order of milliseconds. The only costs involved are some wasted work when the failure occurs right before a chunk (by default, 16MB) is finished– that chunk must be downloaded again. In general the cost includes overhead to initiate new connections and wait for TCP windows to open. This potentially wastes several round-trip times before performance recovers.

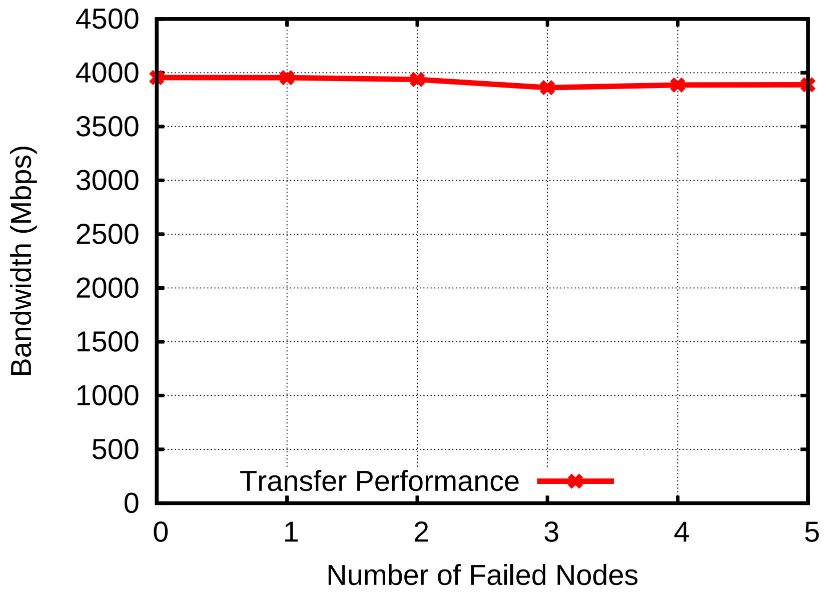


Figure 7.10: CEP Performance Under Node Failure

7.7.2 Tolerating Inaccurate Scheduling Data

Section 4.5.2 outlined several ways in which metadata could be wrong. Data constraint metadata can only be wrong due to malicious peers, in which case the entire system can be brought down, or due to misconfiguration, in which case only the given peer is harmed. In contrast, network constraint metadata is easily misestimated.

As before, performance results depend heavily on data constraints. Here we use a simple topology: two servers with 2GB of identical data and one client for that data. The client has an independent 1Gbps link to each server. We vary the capacity estimation for one of the links from 0 upwards. Figure 7.11 shows the output for this configuration.

The ideal line shows the bound due to actual physical capacity, while the "as scheduled" line shows the transfer performance expected exactly following the greedy scheduler output. An accurate estimation of 1Gbps would equally download data from both servers, producing maximal performance. Overestimating link capacity overloads that server, while underestimating link capacity overloads the other server. This curve shows that the greedy scheduler is tolerant to error: in dividing capacity among peers, it divides errors.

The “actual” line shows the performance actually running this simulation. Regardless of the schedule provided, the client receives information on both available servers. It begins downloading blocks from both as fast as possible– using the optimistic parallel chunk download mechanism described in Listing 4.8 and also on Page 70.

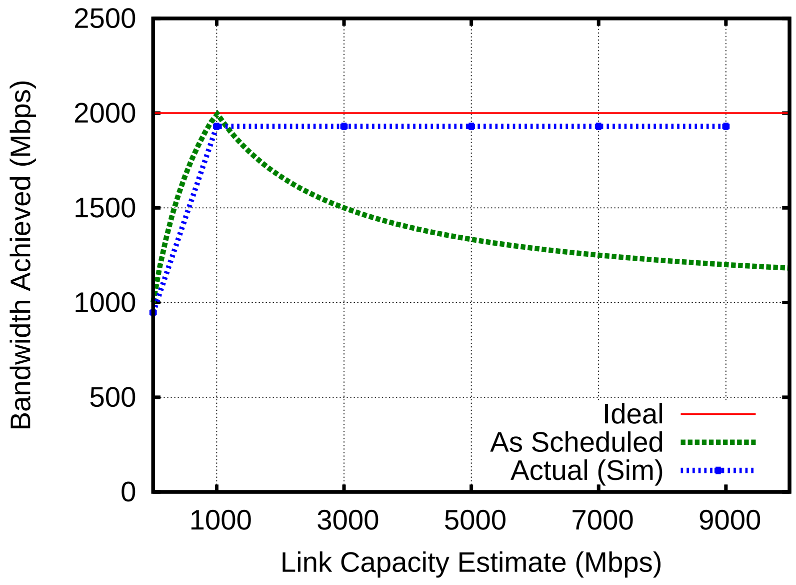


Figure 7.11: Performance Effects of Capacity Misestimation in CEP

In this trivial network, no other peers compete for bandwidth, so performance is limited only by actual link capacity. Put another way, the erroneous estimate is quickly corrected based on actual transfer performance; this behavior is part of the reason we focus on data constraints rather than network constraints. In more complex network configurations other peers may also place demands on servers. These second order effects may limit us to the rates as scheduled, however inaccurate.

7.8 Outperforming Traditional Content Distribution

The remainder of this chapter focuses on integration with the BitTorrent peer-to-peer content distribution application. We show how the addition of enhanced CEP features (see Section 6.2.2, page 97) to the BitTornado client increases bandwidth by a factor of 4-8 depending on the level of sharing, decreases latency by an order of magnitude, provides higher fairness, requires less tuning, and provides the richer partial-file transfer semantics we desire.

Tests are performed with cold caches to allow fair comparison across tests. Short transfers are ultimately limited to disk bandwidth. Longer transfers can theoretically exploit caching, particularly in high-sharing configurations, but BitTorrent's RRF block selection interacts poorly with caching algorithms. These per-node features are not immediately obvious as our experiments focus on aggregate performance.

[The remainder of this page has been intentionally left blank to ensure subsequent figures fall on the same page as their associated text.]

7.8.1 BitTorrent Baseline Performance

First we look at performance of stock BitTorrent. This places the results for our later modifications into context, and shows several features relevant to later discussion. Using the csag-fast cluster, we determine completion times for various numbers of nodes performing a 256MB transfer. For reference, the performance of a 128-node multicast tree is given as a baseline.[2] Figure 7.12 shows a graph of these results.

Completion time is roughly logarithmic in the number of nodes. BitTorrent scales well with high receiver sharing. However, the total completion time is longer than necessary; a naive multicast tree would get data to all peers much more quickly. Also, there is a large startup time: for most of a transfer few peers complete, then suddenly the majority of peers finish with a few stragglers. The primary reason for these results are the source's behavior in BitTorrent: it has to verify the entire file before it can be shared, and tries to distribute blocks across the whole network versus just to carefully selected peers.

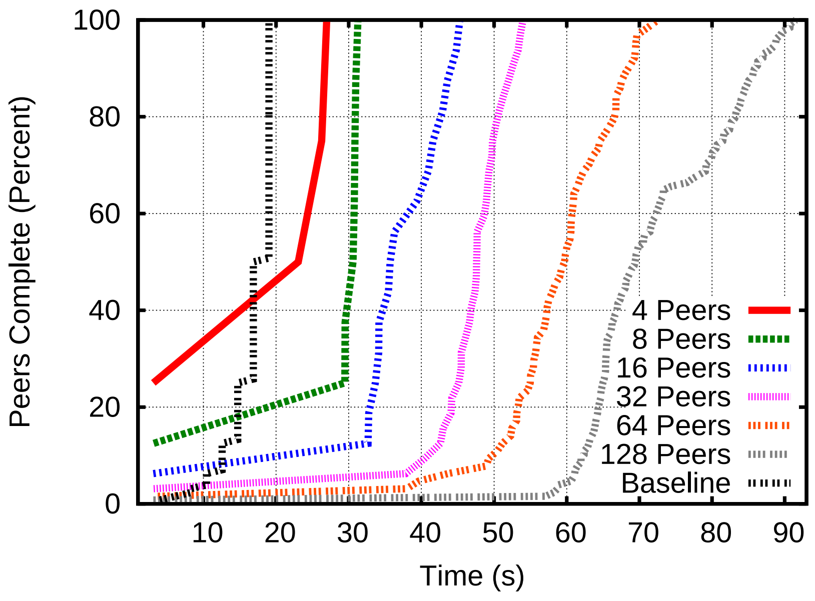


Figure 7.12: BitTorrent Peers Complete vs. Time

[2]This is without pipelining; more intelligent multicast trees or meshes can do even better.

7.8.2 Performance of CEP's High Sharing Optimization

Now we look at aggregate completion times of BitTorrent and CEP, both for the original "default" greedy algorithm and with the "high sharing" optimization. The LP algorithm's performance is equivalent to the default greedy algorithm's and is not shown. Figure 7.13 shows the results using the same transfer configuration as the previous test.

For small numbers of peers, the stock CEP algorithm is best– these are effectively low-sharing environments and the source can supply all peers at high speed. As the number of peers grows, however, the source is saturated and the completion time grows (source transfer rate is consistently around 800Mbps for all tests). BitTorrent scales as in the prior graph. The High Sharing optimization initially performs intermediate the others; with few peers the two-phase transfer induces higher overhead than the default one-phase CEP transfer. As the number of peers grows, however, it outperforms BitTorrent by 25-40%.

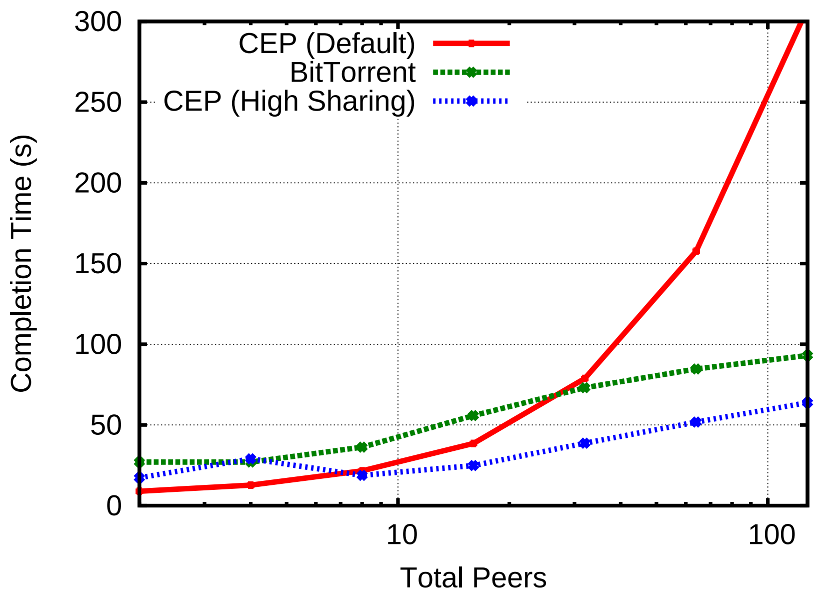


Figure 7.13: Peers vs. Aggregate Completion Time for CEP and BitTorrent

7.8.3 Fairness and Performance Variability

The distribution of completion times for peers is useful for measuring the effectiveness of BitTorrent and CEP fairness mechanisms. Figure 7.14 shows the actual and sorted peer completion time distribution for another run of the 128-node case presented above. We evaluate fairness via Jain's measure [55]; the farther this value is from 1.0, the less fair the system. With t_i as the termination time for peer i, fairness is: $(\sum t_i)^2/(n \cdot \sum t_i^2)$.

The spread of the completion times is quite large; the first peer receiving a full copy in 50 seconds while the slowest taking almost 90 seconds– 80% longer. While we would like to see more consistent results, the majority of peers fall into a uniform distribution about the mean quite nicely. We get get fairness values of 0.992 for CEP and 0.988 for BitTorrent– effectively the same.

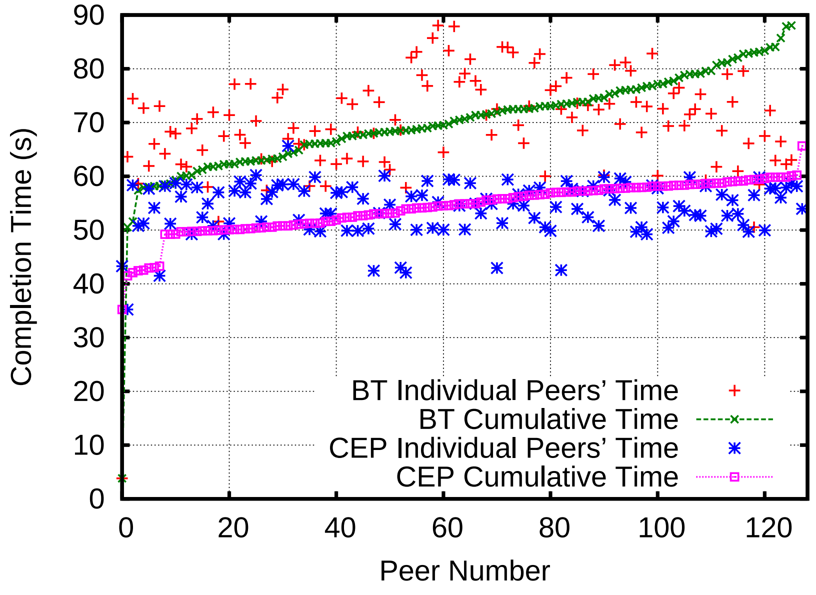


Figure 7.14: Distribution of Completion Time for CEP and BitTorrent

7.9 Tuning BitTorrent/CEP for High Performance

Section 7.2.1 showed TCP tuning necessary for high bandwidth in the WAN. Here we focus on the parameters available within BitTorrent. Some are accessible via the command-line while others require source code changes. CEP has fewer user "knobs." For a fair comparison, we try to maximize performance varying different parameters. By comparing results we discover which provide the most benefit and which are insignificant.

7.9.1 BitTorrent Tuning Parameters

This section evaluates the effects of BitTorrent tuning; CEP has fewer parameters (see Section 4.5.3) and performance is insensitive to their values. We perform a 256MB whole-file transfer with 32 nodes. "Baseline" is the default BitTorrent performance from CVS, "No Double Check" turns off potentially costly download verification, "More Uploads" increases the number of concurrent uploads to 20, "More Unchokes" allows more data transfers outside the tit-for-tat scheme, "Super-Seed" sends a copy of each block into the network before repeating, and "Larger Slices" sets the maximal transfer size to 1MB.

Table 7.3: BitTorrent Tuning Parameters

	Time (s)			**Time (s)**	
Tuning Method	**Mean**	**Ratio**	**Tuning Method**	**Mean**	**Ratio**
Baseline	48.1	100%	More Unchokes	52.5	109%
No Double Check	48.9	101%	Super-Seed	90.6	188%
More Uploads	52.3	108%	Larger Slices	154.9	322%

Without exception, all of this "tuning" hurt performance. In fact all variations of parameters we tried, other than the defaults, hurt performance– the system is surprisingly well tuned even though it was designed for Internet topologies versus high performance networks. Super seed in general performs poorly as it is artificially limiting transfer speeds in favor of block diversity. Larger slices perform poorly due to problems with blocking and string management overhead in Python (see Section 7.2.2).

7.9.2 Block and Chunk Size

The most critical BitTorrent tuning parameter, typically selected automatically, is block size. CEP is range-based internally, but we use maximum transfer size (the “chunk” size) which is roughly analagous to the block size in BitTorrent (see Section 3.1.1).

For this test, we look at the performance of transfers varying either the number of blocks (with a fixed 1KB block size) or the size of blocks (with a fixed 32 blocks per transfer) with various file sizes in BitTorrent. For CEP we set the maximum chunk size to be $\frac{1}{32}$ of the file size; the same idea as having a fixed 32 blocks per transfer. We run 5 trials and report the mean.

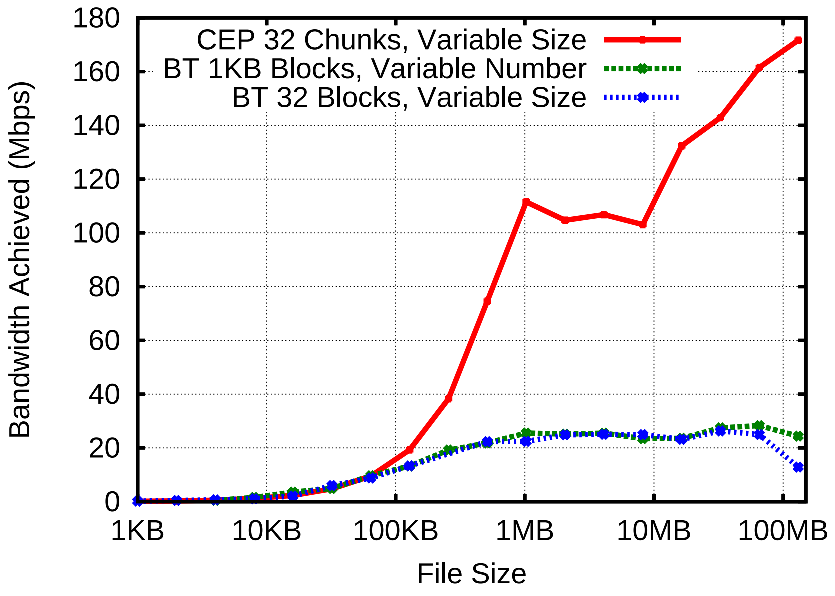


Figure 7.15: Performance Effects of BitTorrent Block Size

Figure 7.15 demonstrates three things for this environment. First, BitTorrent transfers of 1MB are necessary and sufficient to amortize global overhead such as system initialization, file read time, network connection time, etc. Second, BitTorrent works well with block sizes from 1KB to 2MB, but have issues with blocks 4MB or larger. Third, metadata management can handle up to 100,000 blocks but does not scale beyond that.

For CEP, the bigger the maximum transfer size the better: it produces lower overhead. The internal mechanisms, being range based, depend only weakly on the transfer size. Performance improves as file size grows, with performance eventually flattening off, but improving again as disk caching and readahead becomes effective. Disk bandwidth and file system performance provide an upper bound of around 200Mbps (25MBps) on transfer speed for large files.

To achieve the same performance, higher latency networks require proportionally larger transfers with higher parallelism– primarily to work around issues using stock TCP for data transport. Similarly, larger blocks are desirable on such networks to enable TCP congestion windows to fully open; this requires further tuning of internal BitTorrent parameters and changing buffer management schemes (see Section 7.9 for more information). Lastly, with fewer than 32 blocks, the scheduling techniques in the system can not be exploited effectively and performance also suffers.

Given the performance in Figure 7.15 and good selection of block size, BitTorrent will provide high bandwidth on transfers from 1MB and 200GB. BitTornado uses Table 7.4 to automatically select block size given total torrent size. Its selection keeps the number of blocks in the desired range for file sizes up this 200GB limit; however the larger block sizes induce high latency and overhead for partial-file transfers. CEP's default maximum transfer size (16MB) is enough to saturate a single disk's bandwidth, but does not induce the problems seen with similarly large block sizes in BitTorrent.

Table 7.4: Automatic Block Size Selection in BitTornado

Torrent Size	**Block Size**	**Blocks**	**Torrent Size**	**Block Size**	**Blocks**
[0-4MB)	32KB	1-128	[512MB-2GB)	512KB	1024-4096
[4MB-16MB)	64KB	64-256	[2GB-8GB)	1MB	2048-8192
[16MB-64MB)	128KB	128-512	[8GB-∞)	2MB	4096+
[64MB-512MB)	256KB	256-1024			

7.10 Enabling Partial Content Distribution

This section looks at the performance of partial-file transfers– the purpose and justification for CEP's transfer scheduling mechanisms. Our results show that CEP can optimize using transfer constraints to produce efficient, high-speed transfers under a variety of conditions. In contrast, the stock approach has poor performance for partial content distribution problems– it was designed for high-sharing environments. In conjunction with the prior section, these results show the fundamental differences between whole-file and partial-file transfer techniques.

7.10.1 Performance and Sharing

The first test examines performance based on sharing. This is the central feature distinguishing performance of different systems. Using the 'virtual line' data layout as described in Section 7.6.2, we vary the sharing factor from 0 to 1. We use a fixed 32 nodes and 256MB file size; these values are in the range where all systems performed reasonably. We plot the mean with error bars showing the standard deviation.

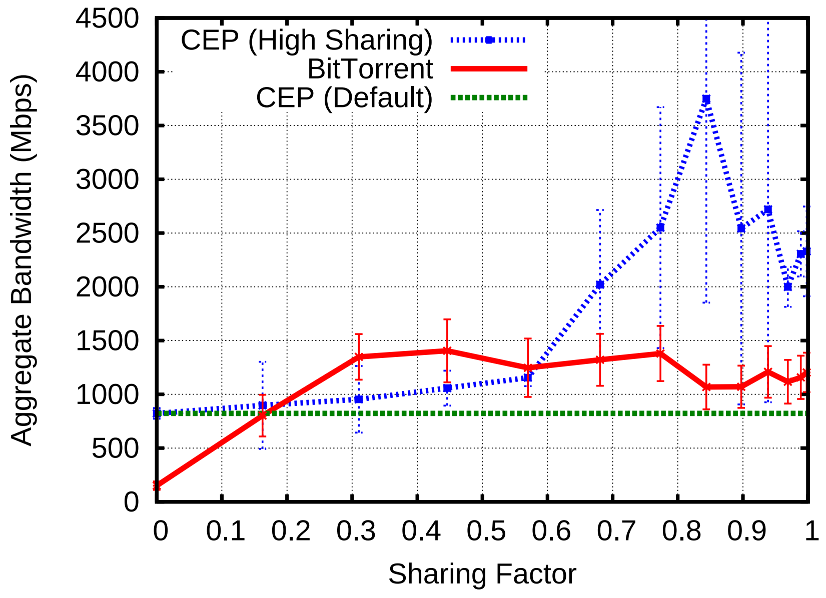


Figure 7.16: Bandwidth vs. Sharing for CEP and BitTorrent

With no sharing among peers, stock CEP is able to most efficiently use the source's upload capacity to satisfy peer demands. The bandwidth achieved is invariant under the level of sharing: it is limited by network link capacity and stack performance.

BitTorrent performs poorly for low sharing, but quickly improves. A sharing ratio of 0.3 means each peer shares approximately 1/3 of its data with every other (although different subsets), meaning there are a large number of possible sources for desired blocks.

CEP with the high sharing optimization is roughly equivalent to stock TCP for low sharing; for intermediate sharing the inefficiency of a two-phase transfer mechanism keeps performance below that of BitTorrent. For higher sharing, however, the initial seeding phase allows for very efficient transfers. The main problem we see here is high variability due to the well-known 'straggler' problem: the last peer to finish determines the system termination time, and hence aggregate bandwidth. We are looking ways to minimize the impact of such peers in the future.

7.10.2 Data Latency

Our next test focuses on the lowest-sharing case, when data is striped across a cluster. This is how GridFTP [4] transfers work. Each peer must locate and retrieve a small 1KB block of data from the single source. Metadata management rather than transfer performance is being exercised. Figure 7.17 shows the completion time for increasing numbers of peers. Error bars are shown for the min and max over 5 trials.

As the number of peers grows, the first peer to complete its transfer tends to terminate in about the same amount of time. However, the distribution of times continually grows, with the last peer to detect and download its block taking 10 times longer for 25 peers than for 2 peers (note the log scale). The stock management schemes in BitTorrent rely heavily on local peers being able to provide copies of blocks. When that assumption fails, the system can not efficiently satisfy complex transfer requirements. In contrast, CEP has an order of magnitude lower latency and smaller result deviation.

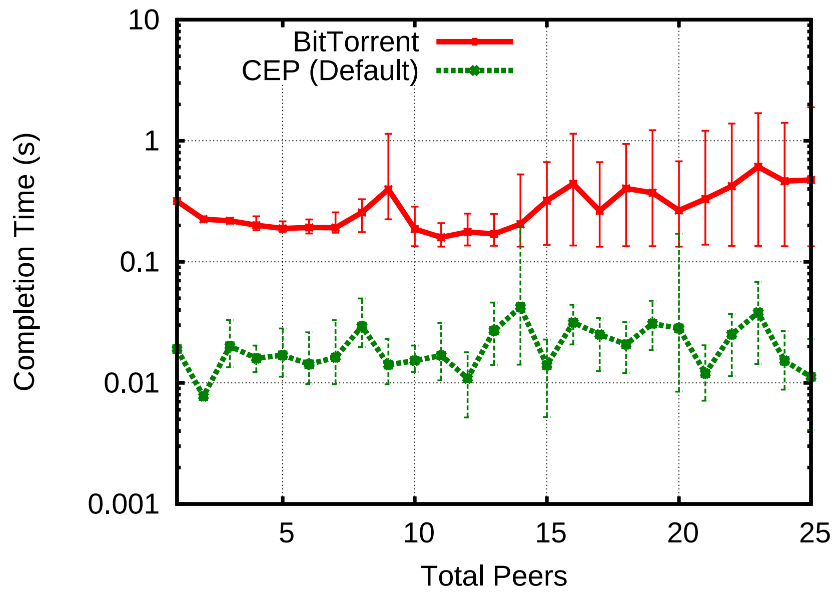


Figure 7.17: Completion Time vs. Total Peers for Partial Transfers

7.10.3 The Failure of High-Sharing Techniques

The next graph extends the ideas above, showing the effects of parallelized downloads. With a single sender we increase the number of peers requesting disjoint blocks out of a 128MB file and report the bandwidth per peer averaged over 5 tests. Error bars are shown to one standard deviation.

As before, there is a large amount of variation in the results; while the general pattern is what we expect, it is unpredictable. Furthermore, around 60 peers, the default number of peers the tracker replies with when a peer makes a metadata request, failures start occurring. Some peers do not get the server that has their block in the response set, and fail to do so before the 20 minute test is over.

The flat metadata model in BitTorrent can not cope with this type of disjoint data. Ironically enough, this is tied to a transitory performance increase; BitTorrent is empirically known to perform best with 40-60 peers (hence the default) and this is seen in the data. Yet as the number of peers continues to grow, the difficulty in finding a server with a desired block begins to dominate, and performance falls drastically.

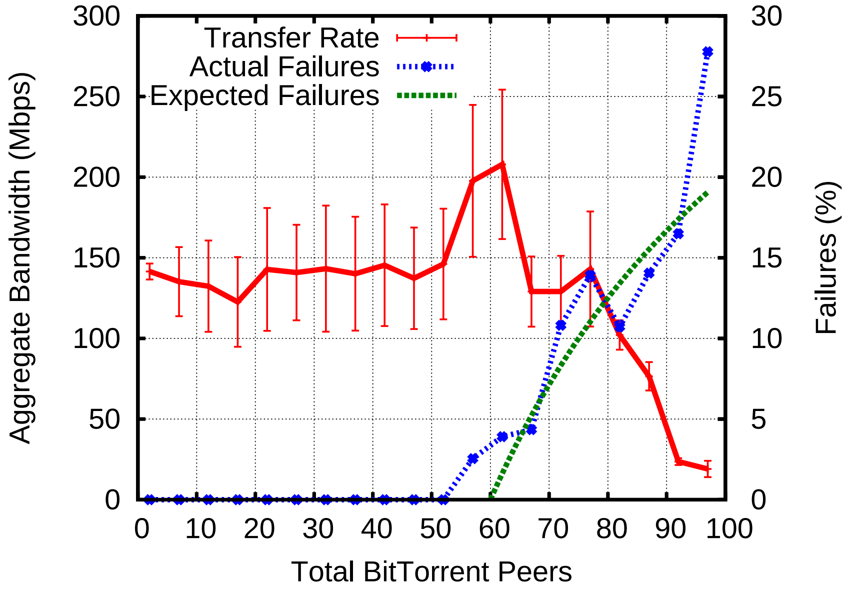


Figure 7.18: Partial Transfer Rate and BitTorrent Failures

The "expected failures" line plots the expected portion of peers to *not* find a server with their desired block. That is, the probability the server with their block falls outside the set of peers they know about: (p-60)/p. This is divided by the average number of peer re-request timeouts, 2 for this test.

The final graph of this section (Figure 7.19) shows the same data from the prior graph but includes stock CEP performance. Ideally, increased parallelism should increase performance up to the sender's capacity and remain consistent until overhead begins to dominate. We see exactly that: CEP performance peaks around 64 nodes (when each peer is downloading 4MB in parallel) and falls beyond that.

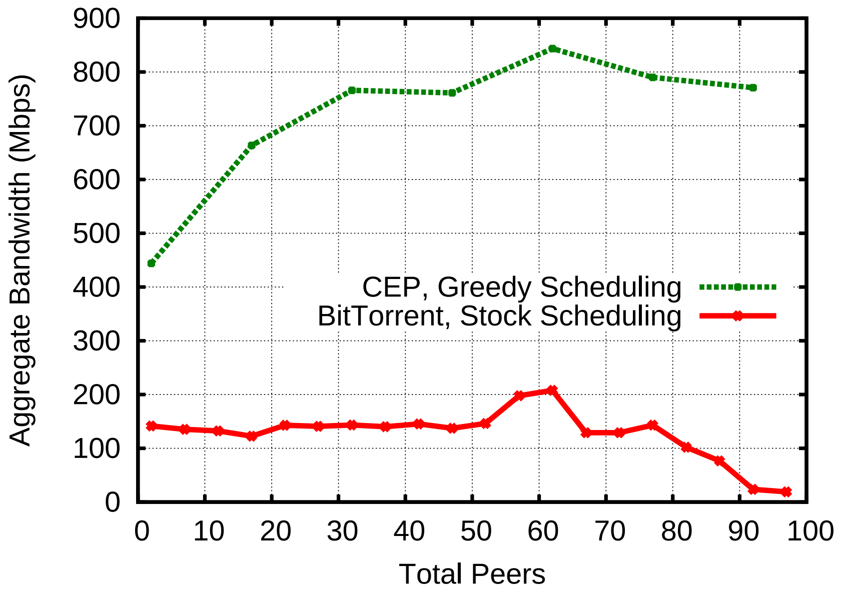


Figure 7.19: Partial Transfer Rate with CEP and BitTorrent

7.11 Summary and Conclusion

This chapter has presented results from simulation, emulation, and real-world tests of CEP and related approaches. These show that the core features of CEP provide high performance, efficiency, robustness, and generality across a variety of environments. The techniques scale to tens of thousands of nodes, provide 4-5× higher bandwidth on heterogeneous configurations than uniform striping, 10-40× faster transaction processing than Apache or DHT approaches, and 4-8× higher bandwidth with 10× lower latency than BitTorrent.

Acknowledgements: Material from this chapter, in part, appeared in “The Composite Endpoint Protocol (CEP): Scalable Endpoints for Terabit Flows,” Eric Weigle and Andrew A. Chien, Proceedings of IEEE Conference on Cluster Computing and the Grid (CCGRID), 2005. Other material from this chapter, in part, appeared in “The Composite Endpoint Protocol (CEP): Scalable Endpoints for Terabit Flows,” Eric Weigle and Andrew A. Chien, Proceedings of IEEE Conference on Cluster Computing and the Grid (CCGRID), 2005. The dissertation author was the primary investigator and author of both these papers.

Chapter 8: Satellite/Terrestrial Networks

The core work of this dissertation focuses on fast, wired networks for high performance computing. However, the infrastructure and transfer scheduling techniques are useful in more constrained environments. In particular, we are interested in content distribution on Satellite/Terrestrial Networks. Such networks are commonly used for distribution of large amounts of data– typically video– from a single source to a number of geographically dispersed destinations.

This chapter reuses the centralized scheduler, metadata collection, and transfer scheduling idea, but with a new algorithm and implementation. Section 8.1 explains why the problem and solution differ from the rest of this dissertation. Section 8.2 presents the new algorithms and infrastructure required. Section 8.3 evaluates these techniques using the criteria from Chapter 7, showing good performance.

8.1 Overview

Making the most of both satellite and terrestrial networks requires specialized extensions to the techniques described earlier. These dramatically improve performance, but are only appropriate in this particular case. The differences are due to the (1) satellite broadcast characteristics and (2) level of sharing- traditional content distribution means total sharing in demand, rather than the partial sharing that has been our focus. This section discusses the unique problems involved and the target environment in more detail.

Our approach has the following desirable properties: it (1) corrects for large error/loss rates- 5% or more; (2) scales in the file size- to gigabytes- and number of nodes- to 1000s+; (3) provides low latency- $<$400ms; (4) is efficient- low network, memory overhead, globally minimizes cost; (5) is fair- different peers share equally; (6) and it is user tunable- user may control peering behavior.

8.1.1 Uniqueness of Satellite/Terrestrial Content Distribution

In contrast to the general transfer scheduling problems described in Chapter 2, content distribution on hybrid satellite/terrestrial networks has these additional characteristics:

1. Continuous transmission: no *inter*-transmission "free" time to recover from errors.

2. Low *intra*-transmission delay tolerance: data must be received within a few seconds after it is initially sent.

3. Variable loss between uplink (to satellite) and downlinks (from satellite): loss is lower on uplinks (by up to $10\times$) due to higher transmit power and antenna gain.

The PlanetLab Grand Challenge with the Public Broadcasting Service (PBS) [14, 91] provides one concrete example. They transfer up to 450GB/day from PBS headquarters to approximately 180 affiliates across North America. Live transmissions require low latency-displaying frames within a few seconds of receipt and < 40ms inter-frame jitter.

8.1.2 Details of Target Environment

Our target environment has high bandwidth in the terrestrial core network (10-40 Gbps), low access link bandwidth (128 Kbps-1.5 Mbps), and moderate satellite bandwidth (20-40 Mbps). One common configuration for affiliates is to have T1 links (1.5 Mbps) to a fast core network and 40 Mbps satellite transponders. Of that, 17 Mbps is used for forward error correction (FEC) and 23 Mbps is available for user data. We will show the exact values are less important than the ratios between them, i.e., the ratio of satellite bandwidth to access link bandwidth defines the effectiveness of this approach.

Terrestrial parameters are roughly based on the OptIPuter [117] or AT&T core network [8]. The latter covers a significant portion of the core Internet in the United States. Extensive studies have shown minimal core loss due to congestion [40]. Terrestrial delay is insignificant compared to satellite delay, making peer locality less important.

Access link information is from the PBS/PlanetLab Grand Challenge correlated with station/location information from the PBS web site [93]. Such links are too slow to satisfy transfer requirements by themselves. Faster links can be acquired, but are expensive compared to satellite transponders which can receive broadcast data at higher speeds.

Satellite links are the primary source of both delay and loss. Traditional approaches use up to 50% of the link bandwidth for forward error-correction (FEC) and fall back on whole file retransmission for uncorrectable FEC errors. Satellite broadcast packets arrive in order, allowing immediate detection of loss for most cases. Error rates will vary between 0.05% to 5% depending on weather and the level of FEC.

8.2 Satellite/Terrestrial Algorithms and Design

Our approach exploits both satellite and terrestrial networks: broadcast via satellite, recover via the terrestrial network. Receivers exchange data to detect and correct errors in a peer-to-peer fashion. Broadcast/recovery are pipelined and overlap for most of a transmission.

We compare two designs. The first is a simple approach with a central scheduler node that collects and maintains metadata information, and the second is a fully peer-to-peer system. In both designs there is a single source uplink node with the original copy of the data. In all discussion, the one source (uplink) transmits data to the satellite, which broadcasts it to all nodes. Recovery is initiated as soon as a node detects a loss.

The scheduler provides performance enhancements when the number of nodes is on the order of thousands, but may be a bottleneck for larger networks. Its purpose is (1) to provide an accurate database of data location for peers to query and (2) to enforce global load-sharing and fairness. Both of these features are provided in a fully peer-to-peer system at the cost of slightly higher latency and potentially lower fairness.

The uplink broadcasts over the satellite and concurrently acts as a peer node in the recovery algorithm. Figure 8.1 shows this structure. Note that any peer or the uplink may act as the scheduler, or the scheduler may be a designated node in the core network.

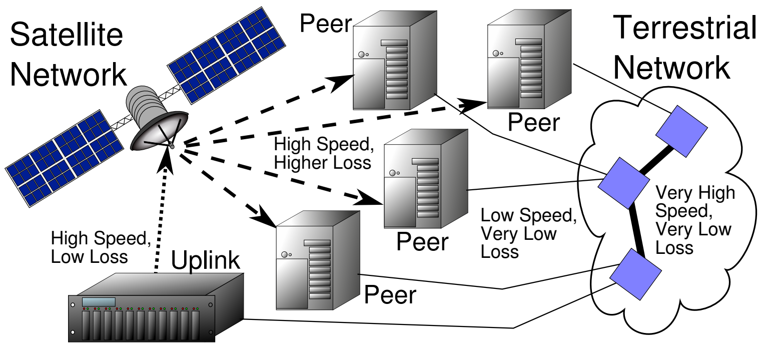


Figure 8.1: Satellite/Terrestrial Configuration

8.2.1 Primary Scheduling Algorithm

Transfer scheduling in this environment resolves to error recovery for packets lost in the satellite broadcast. The algorithms for this are presented in Table 8.1, and they address both metadata and data transfer. In both, data loss is detected based on a gap in the sequence numbers in the incoming packet stream. For the purposes of this algorithm, we treat data as blocks rather than segments; this is appropriate given total-sharing transfers.

Table 8.1: Error Recovery Algorithms

With Scheduler	**Without Scheduler**
Node detects data block loss	
Send Scheduler NACK	Request block from k peers
Scheduler picks best peer	Peers say if they have block
Request block from peer	Request block from first peer
Peer provides block	
Provide Scheduler info:	Request block from source
load, cumulative acks	Source re-broadcasts/replies

In the scheduler-based algorithm, the scheduler can tell a peer where to best find a missing block. Note that the best peer may be the original source node. Peers periodically provide information to update the scheduler's state used to assign recovery peers.

In the fully distributed peer-to-peer algorithm, peers randomly ask k other peers for block information. With independent and uniform downlink loss, the probability that no peer (out of k peers) receives a block from the satellite is $(1 - loss\ rate)^k$, which falls rapidly when k increases. If no peer receives the block, it is probably due to an uplink loss and the peers will ask the source.[1] If the source gets several such requests, it is effectively certain that the block was lost on the uplink and should be rebroadcast over the satellite link or multicast over the terrestrial network. Otherwise the block is sent to nodes individually.

Note that the peer-to-peer system incurs (1) an extra timeout for block recovery if data is lost on the uplink or a poor set of peers was selected, and (2) slightly higher network overhead for state communication. This is the price paid for the extra system scalability; DHTs have similar characteristics but high complexity and overhead for our purposes.

More rigorously, let:

N	=	Number of terrestrial nodes
BS	=	Block size
$Pr(L)$	=	Probability of loss on a satellite link
BW_s, BW_t	=	Bandwidth of satellite or terrestrial bottleneck
T_{send}	=	Time that a block is sent (absolute)
D_{sat}, D_{ter}	=	Max Delay to satellite (uplink↔nodes), terrestrial nodes
$D_{[s\|t]\_xmit}$	=	Max Delay to send a satellite or terrestrial block ($BS/BW_{s,t}$)
D_{tick}	=	Delay between clock ticks (timeouts)
T_{peer}	=	Timout for peer/scheduler response

Using this notation we can calculate aspects of the system behavior: maximal block delay, termination time, and so forth. With uniform independent losses, the chance that a node immediately gets any particular packet is simply $(1 - Pr(L))^2$ since packets may be lost on the uplink or downlink. This is the probability of "ideal" reception at time $T_{ideal} = T_{send} + D_{s\_xmit} + D_{sat}$. If only one copy of the packet is lost, it will be detected in at worst

[1]Well-known techniques can be applied to avoid the "ack implosion" problem.

one timeout, then metadata requested from the scheduler/peers, then the packet requested from a peer and received at time at most $T_{ideal} + D_{tick} + 4 \cdot D_{ter} + D_{t\_xmit}$. When multiple copies of a block are lost, at most $l = \lfloor (D_{tick} * BW_t)/BS \rfloor$ losses can be recovered per clock tick by each node that *has* received the block. Then, after another (at most) D_{ter} those nodes can provide the block to others.

In the worst case, (1) a block is lost on the uplink so only the source has it and (2) the source is on the terrestrial bottleneck. In this case, after one clock tick (plus network delay) $l+1$ nodes will have the block, after two ticks approximately l^2 have it, and so on. At worst, the last node will receive the block at time $T_{ideal} + (D_{tick} + D_{ter}) \cdot \lceil log_l N \rceil + 4 \cdot D_{ter} + D_{t\_xmit}$. This is a desirable bound as it grows very slowly, but it is subject to a few constraints.

The first constraint is that D_{tick} is large compared to D_{t_xmit} and D_{ter}. If not, the recovery algorithm breaks down – to make progress, replies to requests must arrive before the next timeout occurs.

The second is that the satellite loss rate must be "streaming-recoverable": the terrestrial links must be fast enough to always fix the satellite's losses within a few RTTs of when they occur. This is only true when $(1-(1-Pr(L))^2) \cdot BW_s < BW_t$ or equivalently when $Pr(L) < 1 - \sqrt{1-(BW_t/BW_s)}$. This is why it is the ratio between satellite and terrestrial speeds, rather than their absolute values, that is most important. In practice, the maximum loss rate recoverable at streaming rates is slightly lower due to congestion losses and transient load.

To make this concrete, when losses are streaming-recoverable, the maximum latency to receive a block in our simulations is less than half a second. If the satellite loss is *not* streaming-recoverable, then losses will accrue and the delay to replace a lost packet will grow without bound. Eventually, if the satellite transmission completes and there is no subsequent transmission, the errors can be recovered at the speed of the terrestrial network. We see these features in Section 8.3.

8.2.2 Error Recovery

The peer selection algorithm is the core of block recovery. With a scheduler, or other metadata, we can globally optimize this step. Without it, we cannot improve upon random selection. The best peer is selected as follows:

```
Select a random peer from
    the minimal cost peers from
        all peers which have the block.
```

The first selection is easy; any reasonable pseudo-random number generator suffices. Similarly, the third selection is easy when information is centralized or other infrastructure exists to maintain it. The second selection depends on "cost." This can be any formula, but for our tests it is simply load: lower load, lower cost. This selection is a trivial instance of the job scheduling or bin-packing problem where all jobs are the same size (transferring one block). In such a case, minimizing global cost resolves to globally balancing cost, which is exactly what this algorithm attempts to do. Thus, its optimality is limited only by metadata accuracy.

The delay in metadata propagation means the scheduler may err by up to the number of requests that arrive in the time it takes a packet to traverse the network. This is at most $D_{t\_xmit} + 4 * D_{ter}$: D_{ter} to return metadata, D_{ter} to request the block, D_{t_xmit} to send it, D_{ter} to traverse the network, and D_{ter} to return an ACK to the scheduler. The expected difference is only a few blocks (below D_{tick}/D_{t_xmit}) and will be self-correcting over time: the percent difference from optimal falls as the transmission size increases. Our empirical results show this behavior. Randomization avoids overloading individual peers during the update period.

8.2.3 Algorithm Features

This section explains a couple of the results that follow immediately from this algorithm design (we will return to these ideas in Chapter 7). First, one benefit of the concept of maximal streaming-recoverable loss rate is that when loss is dramatically lower, we can exploit that fact to send less data over the satellite link. These would then be treated as losses, and recovered on the terrestrial network. This makes sense when the cost structure is such that cost(satellite)$>\Sigma_N$cost(terrestrial$_i$), since for every satellite broadcast we must send N terrestrial blocks. This reduces the amount of satellite data to be sent by a few percent when loss is low.

For satellites whose level of redundancy can be adjusted, we should therefore set the FEC level to the minimum tolerable by the terrestrial error correction. This value is known from the analysis in the prior section, and with some knowledge of the satellite error rate we can set the level of redundancy appropriately.

Second, one particularly useful cost function for scheduling in practice is as follows: a simple (linear) weighted sum of load, link speed, link expense (i.e. cost in dollars), and distance between peers. Link speed is required when links differ significantly in speed; however in such a case global termination will always be determined by the slowest node in the network given uniform loss. If losses were more common on faster nodes, we could still perform well. Link expense ties cost to real-world price. Distance allows us to minimize latency; in particular for soft real time systems we can increase the cost of nodes farther away as deadlines approach.

This approach can also help engineer the minimum cost networks that satisfy user requirements. Specifically, when we know the parameters to the cost function and the expected/experienced error rates at different receivers (e.g., due to climate or being on the boundary of satellite coverage), we can determine the terrestrial bandwidth each node will require. For example, some nodes might only need DSL lines, while others may require T-

3. However, this must be globally optimized since we must ensure receivers have sufficient bandwidth to peer with others– techniques such as linear programming can be used with the cost functions to determine an appropriate global cost minimum.

8.3 Satellite/Terrestrial Evaluation

Our evaluation first demonstrates the system performance as a function of satellite link loss. Second, we demonstrate scalability in terms of file size and number of receiver nodes. Third, we show receipt latency is low. Fourth, we demonstrate the access links' utilization efficiency. Finally, we show that the protocol as a whole is very fair. Together these justify our claims.

8.3.1 Loss Recovery

The first question concerns how much loss we can recover using this system. We model errors as uniform, independent burst losses using a standard Gilbert-Elliot model per link (more complex correlated error models do not qualitatively change our results). These initial tests are performed on a relatively simple topology– access links connect to a single backbone router, creating a terrestrial star topology. Satellite links are 23Mbps and terrestrial links are 1.5Mbps (T1s). Figure 8.2 shows a graph of loss rate over the satellite links and completion time for a 100MB broadcast to 10 nodes.

As we expected, there is a sharp knee showing the point at which streaming recovery is no longer possible. This knee is at the same point for both schemes given the same access link speeds. With slower peer links, less loss can be recovered and knee moves lower.

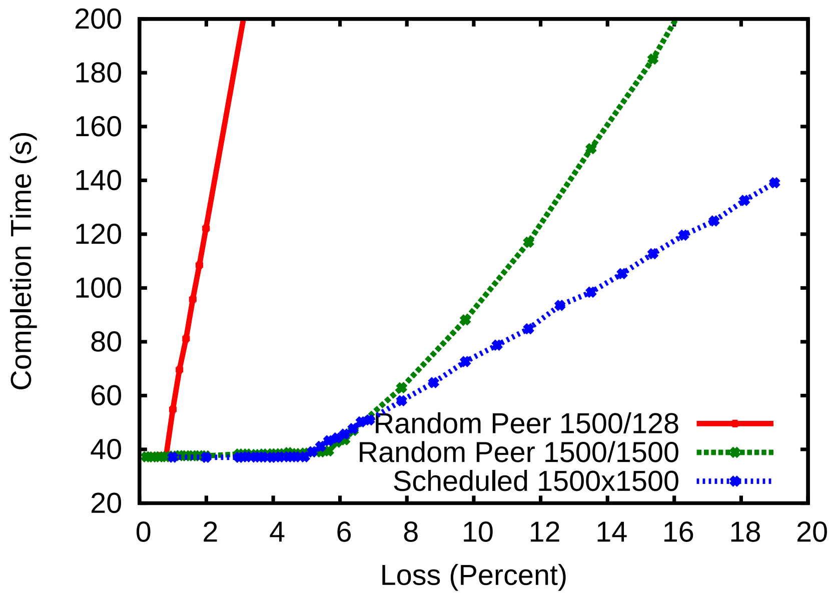


Figure 8.2: Bidirectional Loss Rate vs. Completion Time

According to our calculations in Section 8.2.1, the maximal theoretical streaming-recoverable loss rate is 6.5% ($23Mbps * .065 \approx 1.5Mbps$), but we see the actual maximum at 5%. This is about 75% of ideal recovery efficiency. The difference is primarily due to the application's in-order semantics; if some data in the recovery window cannot be retrieved, recovery may temporarily stall waiting for them. This may occur when multiple segments are lost on the uplink and exist only on the source; it requires multiple iterations to propagate them throughout the network.

Loss rates shown above are aggregate loss, with an equal drop probability on both uplink and downlink. There is no reason to expect the loss rates to be equal; in fact, uplink losses will tend to be lower. Moreover, recovery from uplink losses and downlink losses are distinct problems. For uplink losses, no node receives that data; it is only available from the source. For downlink losses, most peers can provide it. Luckily, our unified solution can solve both problems and performance is weakly tied to loss correlation. In the worst case, all nodes can recover from the source node in $log(N)$ iterations of the algorithm.

Consider Figure 8.3, which for a fixed 10% loss varies the proportion of the loss incurred on the uplink versus downlinks. In all cases, the total amount of data to be recovered is the same. All that varies is the correlation of the errors.

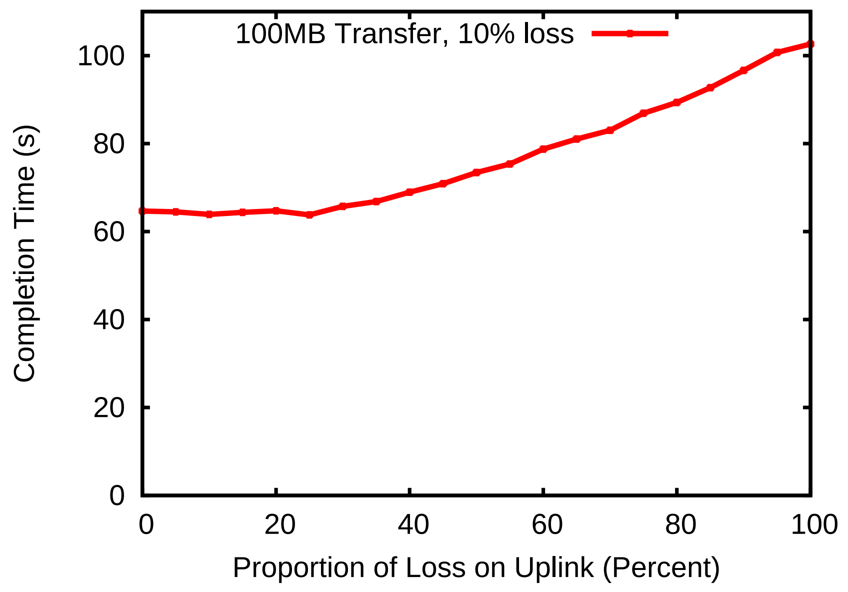


Figure 8.3: Effect of Uplink/Downlink Loss

As more losses occur on the uplink (loss correlation among peers grows), more must be globally rebroadcast on the terrestrial network. This leads to overloading of the source node, and requires multiple iterations of the peer-to-peer recovery mechanism to get the data to all nodes. Put another way, the set of losses which occur on the uplink must be broadcast on the terrestrial network. The difference in performance between 0% and 100% of the loss occurring on the uplink is the benefit of exploiting the satellite network.

The deeper cause of this performance difference is more subtle. With a uniform loss distribution on the downlink, as the number of nodes grows the probability that no node has a given segment (meaning it must be retrieved from the source) grows proportional to $loss\ rate^n$. On the other hand, the number of losses and the capacity to recover losses grows proportional to n. This means that as the number of nodes grows, we have an excellent chance of being able to recover using a peer-to-peer mechanism.

8.3.2 Scalability

The second question concerns scalability—both in terms of file sizes and in terms of system size (number of nodes). Ideally, completion time should be linear in the file size. Similarly, as the number of nodes grows, the completion time should stay constant (below the streaming point) and grow slowly above it (as nodes become overloaded).

To test the former, we broadcast files of sizes ranging from 1MB to 1GB under various amounts of loss (above and below the knee in the prior graph) using T1-speed peers, and present the results normalized to the percent of ideal time (zero loss on the satellite link).

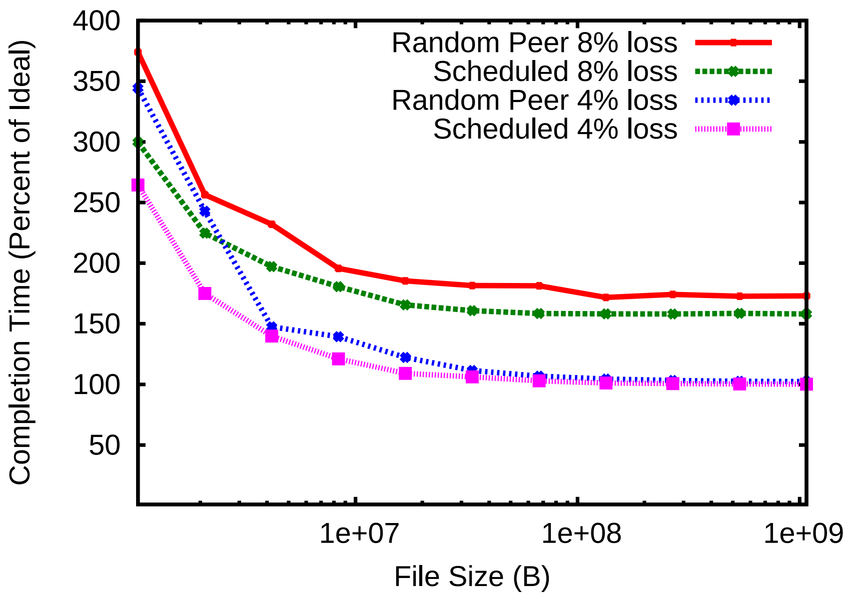


Figure 8.4: Completion Time vs. File Size

For 1-8MB transfers, network latency and other overheads dominate. Above that size, we can effectively meet the ideal transfer time for moderately high loss rates. The peer-to-peer system matches the scheduled scheme well but is slightly less efficient for high loss rates– the randomized location scheme has higher overhead.

To test scalability in terms of nodes, we broadcast 100MB under 2% loss to increasing number of nodes. Figure 8.5 shows completion time for this test.

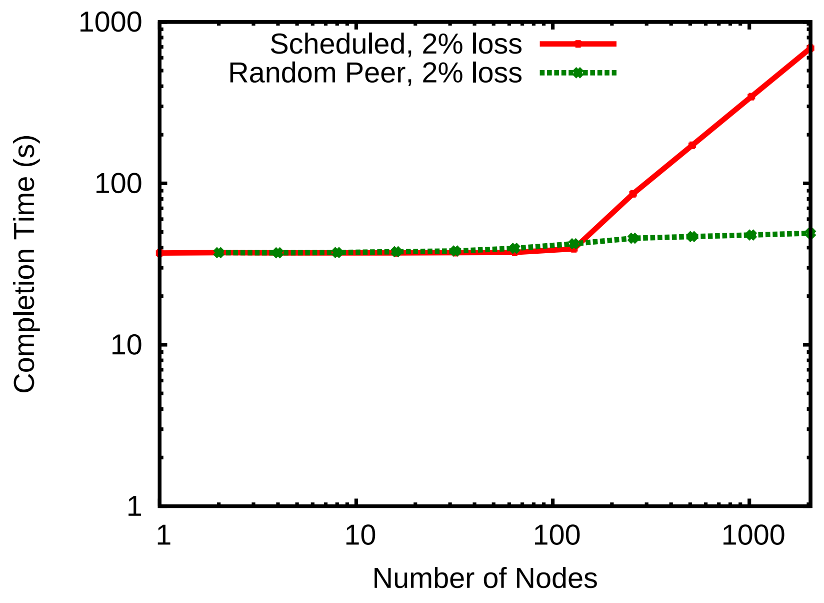


Figure 8.5: Completion Time vs Node Count

A 2% loss is streaming recoverable with 1.5Mbps peers, but at around 128 nodes, the capacity of the centralized Scheduler to track losses and reply with metadata is exhausted. Moving the Scheduler to a 100Mbps core link shifts that knee to around 1,800 nodes. The peer-to-peer system is slightly less efficient, but scales better.

The peer-to-peer curve grows approximately with the log of the number of nodes, as expected– this is due to data propagation requiring an additional iteration with each doubling of the number of nodes. Larger numbers of nodes were not simulated due to time constraints and other limitations with the simulations/simulator; however, we reiterate that these results satisfactorily meet the goals of our application.

8.3.3 Intra-Transfer Performance

The next logical question is how the system is performing within a transfer, i.e., how the different parts of the transfer progress and whether we meet our latency goals. Figure 8.6 shows the aggregate data received via the satellite and terrestrial networks over time for all 10 nodes under high-loss (10%) conditions.

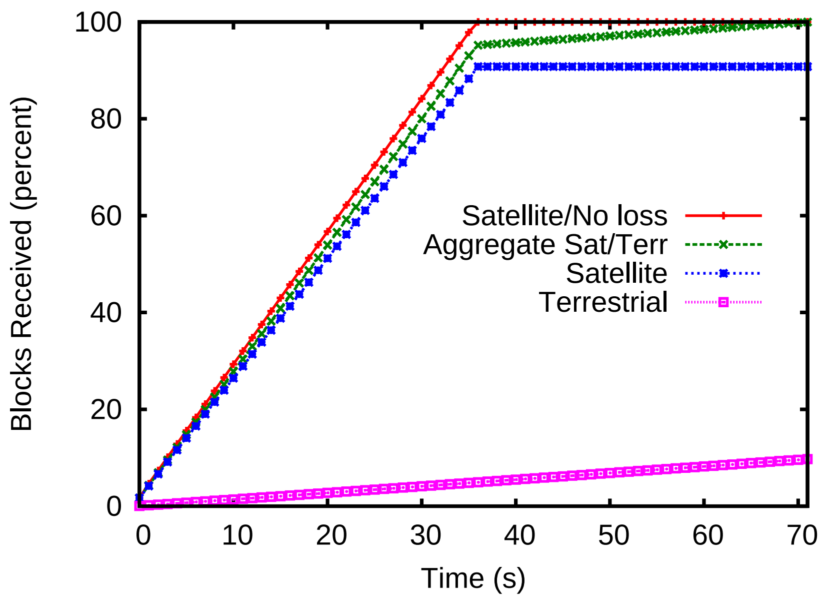


Figure 8.6: Initial Transfer and Recovery Phases

The ideal curve transfers all data over the satellite link. Our actual performance is effectively a linear combination of the degraded satellite signal and terrestrial recovery transmissions. For low-loss cases, both terminate at same time. For higher loss cases such as this, we spend time after the satellite transmission has completed to recover the errors.

This graph shows that while the system is able to recover higher losses at low cost, the satellite link may be underutilized. In such a case, we need to increase the forward error correction slightly to bring the net satellite errors down; otherwise the latency grows unacceptably (up to 32s for the last segment lost in this test).

The normal latency is captured in Figure 8.7, which shows the difference from the expected time of arrival for data in a 100MB broadcast to 10 peers with 2% satellite loss. The vast majority of the data (98%) is received when expected from the satellite. The 2% of segments lost are recovered via the peer-to-peer mechanisms, with latency up to 450ms but on average about 175ms. Each doubling of the number of peers increases the worst-case latency we observe by approximately 50ms. The total latency incurred is of the same order as the baseline latency to reach and return to a geosynchronous satellite.

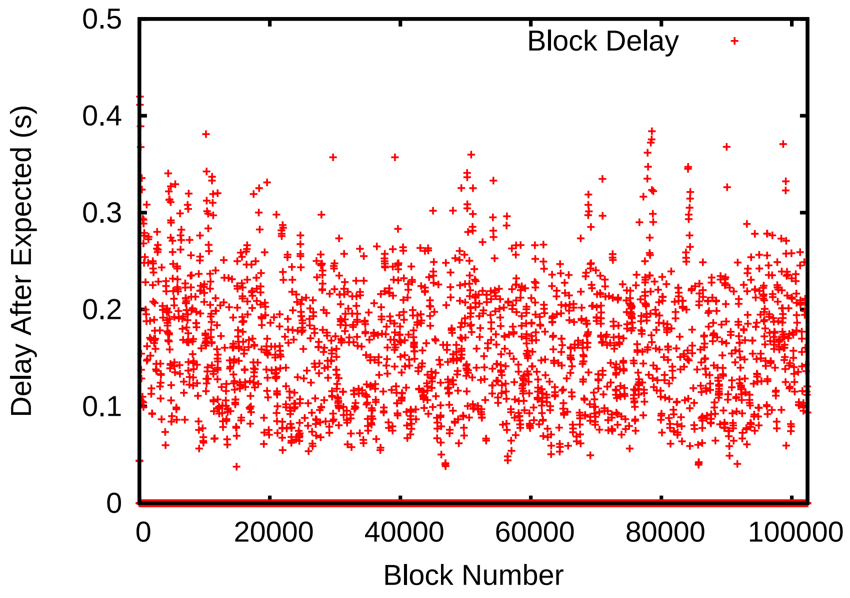


Figure 8.7: Delay Due to Loss

8.3.4 System Utilization

To show system efficiency we must have high access link utilization over the duration of the transmission (the core network is lightly loaded relative to its total capacity). Figure 8.8 shows the percent of access link capacity utilized by data; that is, not including packets corrupted, metadata, or packet header overhead using the the large core (Internet-like) network configuration. We perform a 1GB transfer under 2% loss rate on this network, parameters representing a realistic streaming-recoverable scenario. This test uses the centralized Scheduler. Full P2P results are qualitatively the same but have higher variability, making for messier illustrations.

The first thing to notice is that with streaming-recoverable losses, there is no phase where the satellite is idle. Both networks are utilized for the whole transfer. Second, the losses only require about 30% of the data capacity of the terrestrial network. Third, the results are not qualitatively different than those on simpler configurations (not shown due to space limitations). This has held true for all our tests. The primary difference is that the backbone structure creates a higher variation in the data than a simple benchmark topology.

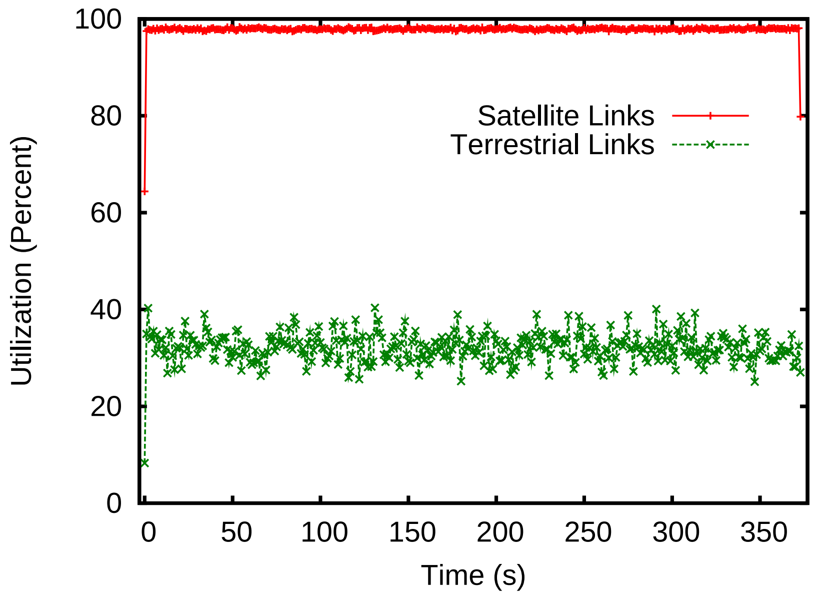


Figure 8.8: Percent Utilization of Core Access Links with 2% Satellite Loss

In general, demand for access link bandwidth is directly proportional to satellite loss, and satellite utilization (goodput) is inversely proportional to loss. The 2% loss rate in Figure 8.8 shows high satellite utilization and low access link utilization. The point at which transient access link utilization reaches 100% is the point at which streaming recoverability becomes impossible.

Furthermore, the data to be sent by each node, and hence the link utilization, is controlled internally by a token-bucket mechanism. Currently, it allows 100% of the link to be used for recovery, but the protocol is user-tunable. Users can limit this to any desired proportion to avoid competing with other traffic.

8.3.5 Metadata Update Efficiency

There are two reasons a node may be unable to satisfy a data request: if it is heavily loaded or cannot be located. Both of these problems can occur due to stale metadata. If the scheduler does not know which nodes have which blocks, it may overload certain nodes while others go idle.

Figure 8.9 shows the staleness of metadata in terms of segments known to receivers but not the Scheduler. This figure shows the efficiency of metadata update.

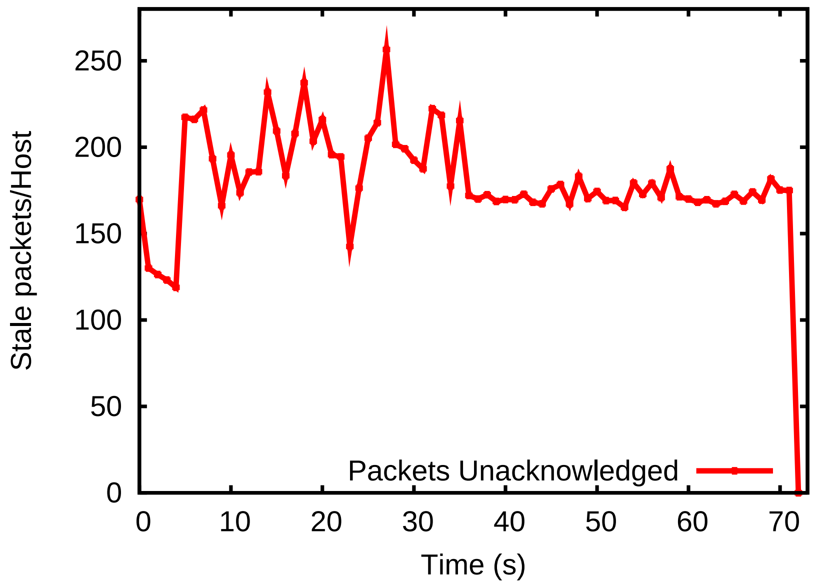


Figure 8.9: Number of Unacknowledged Segments

The Scheduler information lags about 180 segments behind actual system state. This resolves to about 60 milliseconds, of which block timeouts make up 10-20ms, network latency about 30ms, and the remainder is due to lost metadata packets and queuing delay. We conclude that our metadata update mechanism is efficient; with most of the delay due to unavoidable physical constraints. Furthermore, this delay is sufficiently low such that accurate data location is known to the scheduler by the time lookup requests arrive.

8.3.6 Fairness

Finally, fairness using Jain's measure [55] has been high (close to 1) in all cases. This is in terms of data uploaded, work done that is of note direct benefit to a node, and termination time, the actual node's performance. This provides incentive for users.

With the Scheduler, for the simple topologies and blocks sent it was uniformly over 0.999, meaning all nodes sent out almost the same number of blocks. Similarly, for recovery blocks received it was over 0.999, in this case due to the global termination constraints and uniform loss behavior. Under nonuniform loss, by definition, some nodes will unfairly load the system to recover their data, but the blocks sent will still be evenly allocated (i.e., no tit-for-tat behavior).

The more realistic core topology had fairness values consistently around 0.987, also very high but somewhat lower than the simpler topology. The difference was again due to the structure in the core network; some nodes were closer to the Scheduler. This implies the Scheduler data was consistently more fresh for those nodes, and they would tend to have slightly higher load.

Without a centralized scheduler, 65% of our tests showed fairness over .99, 94% over .95, and 100% were over .88 fair. In general, the smaller the transfer the greater the chance that the random peer query will create an unfair work allocation; the last 6% above were transfers where peers uploaded less than 20 blocks/node.

In sum, these results coincide with our prior analysis and support our claims as to the system's performance under a variety of conditions.

8.4 Summary and Conclusion

This chapter has shown two specialized dynamic transfer scheduling algorithms for traditional content distribution over hybrid satellite/terrestrial networks. These are extensions to the techniques described elsewhere in this dissertation which optimize for the unique constraints encountered in such an environment. We have shown that the approach provides high bandwidth, low latency, and high fairness for realistic environments.

Acknowledgements: Material from this chapter, in part, appeared in “Peer-to-Peer Error Recovery for Hybrid Satellite-Terrestrial Networks,” Eric Weigle, Matti Hiltunen, Rick Schlichting, Vinay A. Vaishampayan, and Andrew A. Chien, Proceedings of 6th IEEE International Conference on Peer-to-Peer Computing (P2P), 2006. The dissertation author was the primary investigator and author of this paper.

Chapter 9: Related Work

This chapter discusses the body of work related to CEP. We divide it into four categories: classical distributed systems, grid systems, overlay networks, and supplemental technology. We highlight the differences between CEP and other work, showing that we provide (1) a richer partial-file, many-to-many distribution model and (2) improved performance.

Classical distributed systems includes work on parallel computers and communication libraries. Grid systems include advances in parallelization, file systems, network systems, and high-speed file transfers. Overlay network schemes include content distribution networks, caches, multicast meshes, and most current peer-to-peer research. Supplemental technology includes work on erasure codes, high performance transport protocols, and other high performance hardware/software; these are related in the sense that they enhance the functionality of the CEP or other approaches, but address orthogonal issues.

9.1 Classical Distributed Systems

Distributed systems research historically focused on coordination, agreement, and fault tolerance [11, 34, 61, 72, 111]. Work with parallel processing and parallel communication libraries is the most relevant, but their assumptions differ from CEP. We provide weaker consistency semantics which favor high performance transfers.

CEP is a write-once system. Once shared, data cannot safely be changed. In our implementation, this resolves to a last writer wins mechanism– but we do not define the write ordering. Replicated data is by definition identical. If users require stronger consistency or features such locks, they can be provided by other software or use of disjoint segments in the 64-bit CEP address space.

9.1.1 Parallel Computers

Today's high-end computers use Symmetric Multi-Processing (SMP) or threaded processors, or both. Historically, systems were based around large parallel vector architectures, hypercubes (such as nCUBE [36]) or fat-tree interconnection mechanisms. Recent systems such as IBM Blue Gene [94] have revisited these problems.

Such systems have many-to-many communication properties, but their focus on low latency and strict consistency leads to different solutions than CEP. Internally they use simple data busses, flat or strictly hierarchical networks, global caches, and simple back-off/retry algorithms. These suffice for a single, local, low-latency system. Unfortunately, they do not scale to large distributed systems; finding an appropriate data replica, optimizing the transfer, and dealing with WAN issues require the techniques found in CEP.

9.1.2 Parallel Communication Libraries

Applications that are written for computational clusters use parallel communication libraries to simplify their programming. MPI, PVM, and distributed shared memory libraries are commonly used.

MPI, the **Message Passing Interface** [78] standard supports the notion of a communication target, a communicator, which can be used by a collection of nodes as the target for a sequence of messages– an aggregate logical flow. Similarly, it contains ways to express shared memory and remote direct memory access (RDMA) which allow implicit many-to-many communication.

Many MPI features are implementation-dependent; the standard does not specify their internal mechanisms. In particular, implementations generally require homogeneous peers, do not provide transfer scheduling or wide-area transport, and provide only a low-level interface. In contrast, CEP provides a global name space, transfer scheduling, a high-level interface, and other features.

MPI and CEP are targeted toward slightly different environments; MPI for local, low-latency intra-application communication, CEP for inter-application communication supporting larger transfers, larger networks, and more peers. The two approaches are mutually complementary. Conceivably the algorithms described in this dissertation could be used to extend future MPI implementations.

PVM, the **Parallel Virtual Machine** [109] project supports the creation of a virtual computer comprised of multiple heterogeneous nodes and networks. It is intended to support less tightly coupled applications than MPI. While the PVM framework could naturally support replication, transfer scheduling, or other ideas from CEP, these features do not exist. PVM instead focuses on distributed computation issues: management, coordination, etc., rather than communication problems.

Most applications choose MPI rather than PVM, due primarily to support from hardware vendors and system integrators. The virtual machine concept is still powerful, however, and may have a revival with increased use of the Globus Toolkit [84] and Distributed Virtual Computer [110].

Distributed shared memory (DSM) extends the virtual memory hierarchy to remote nodes. This is a popular interface– nearly thirty active projects (including the MPI standard) include DSM features. In some sense, CEP provides a write-once DSM mechanism via the linear byte range abstraction: any node can read or write into that space. An API to memory map byte ranges would make this concrete, but is left to future work.

DSM systems tend to focus on latency over bandwidth, some to the extent of being designed around data prefetching. One such system is LambdaRam/JuxtaView [66], in the OptIPuter project. It includes many-to-many communication features, but does only implicit transfer scheduling. Instead, it includes a block-based prefetching mechanism targeting visualization applications. CEP supports heterogeneous nodes, full transfer scheduling, and has a more general interface.

Other popular DSM systems, such as TreadMarks [61], focus on the creation of the memory for a virtual machine. They target memory consistency guarantees and simple management functions. They do not work well over the wide area, with loss, heterogeneity, or replication, nor do they do transfer scheduling.

9.2 Grid Systems

Grid systems are the natural evolution of local area clusters. They "coordinate resources not subject to centralized control, using standard, open, general purpose protocols and interfaces, to deliver nontrivial qualities of service" [43]. This requires complex software, which itself falls into several sub-categories: application parallelization mechanisms, parallel file systems, and file transfer mechanisms.

9.2.1 Application Parallelization Mechanisms

Running a single logical application on a cluster requires parallelization– often a nontrivial task, if the application was designed for a single processor. Toolkits to simplify this job often include many-to-many communications mechanisms, and hence are relevant to CEP. One such project is **Chaos/MetaChaos** [71, 125] from the University of Maryland.

MetaChaos is has similar motivations and includes an interface similar to CEP. In particular, it includes a global linearization abstraction equivalent to the distributed byte stream we developed, and a different but roughly equivalent API. Internally, the mechanisms used are quite different.

In particular, it is another system focusing on low latency. They use a 'fuzzy' time-based segment matching scheme and a producer/consumer mechanism. This approach does not support replication, nor missing segments, nor failures. They do not use account for dynamic feedback, node location, or node heterogeneity. They do not support third party transfers. CEP supports all of these features.

Their linearization representation is based around arrays, but supports arbitrary structures via a more expensive representation. CEP's internal representation is less efficient for dense strided arrays, but more efficient for the arbitrary case. They rely on an all-to-all broadcast, which works poorly over the wide area and is not scalable.

Finally, they do provide two features which we have left to future work. These include caching of transfer mappings and support for multiple programming languages. Currently, CEP does no caching and supports only C/C++.

9.2.2 Parallel File Storage and Transfer

Parallel transfers are commonly used to avoid bottlenecks with physical media, such as spinning disks. Some current parallel file systems and file transfer applications support replication and heterogeneity; similar to CEP. However, they focus on locking, data consistency, and small local-area networks. They can not optimize in the same way a content distribution system can– they have insufficient metadata. Block request streams do not provide enough information on global supply and demand, even with "hinting." CEP focuses on primarily static data, more complicated distribution demands, supports large-scale distributed systems, and provides global transfer optimization.

The **Parallel Virtual File System** (PVFS) [21] is a simple striping-based mechanism for storage of files across a number of cluster nodes. It focuses on raw performance does not support node heterogeneity, replication, or other features. Red Hat's **Global File System** (GFS) [54] is similar but also supports replication, fault tolerance, and scalability to 256 nodes. It does not support wide area transfers, perform intelligent replica selection, transfer scheduling, or scale to large systems.

The **Lustre** [27] file system is more powerful and extensible, including some support for heterogeneous nodes, replication and replica selection. It does not yet support strong transfer scheduling mechanisms or wide-area transfers. Similarly, the **Google File System** (also, confusingly, called "GFS") [47] supports replication and large file transfers, but with the same drawbacks. It also has problems with small transfers (under 64MB).

The **Grid Datafarm** (Gfarm) [112] parallel file system targets “petascale data-intensive computing.” They plan to implement CEP-like transfer scheduling mechanisms, but currently support only homogeneous nodes, uniform striping, and socket-parallel transfers. CEP supports much richer transfers, heterogeneity, and larger numbers of nodes.

Globus **GridFTP** [4] is a user-level parallel file transfer application. It supports striped N-to-N whole-file transfers between clustered nodes above a shared file system, targeting high speed in the wide area. GridFTP assumes homogeneous nodes and globally available data. It incorporates only static striping across nodes and parallel sockets[1], and does not tolerate failures. The Globus Reliable File Transport (RFT) [92] mechanism was developed for that purpose. In contrast, CEP supports heterogeneity in node and data accessibility, partial-file distribution, and is natively fault tolerant.

9.3 Overlay Networks

One way of viewing the structure of many-to-many transfers is to cast it as an overlay network- a logical network built above the physical network. Overlay nodes are our peers and links represent their communication paths. This abstraction lets one apply well known routing, group membership, and agreement protocols to transmission problems.

This section discusses full overlay networks, which focus on routing, and content distribution networks, which focus on multicast and caching. These overlays all support many-to-many communication patterns and some form of data transfer, but do not provide the performance or semantics of CEP.

9.3.1 Full Overlay Networks

Full overlay networks explicitly build a network structure and base their operations on it: searching is done via network algorithms, downloads propagate through the network structure, and so forth. The most commonly known overlays are **Distributed Hash Ta-**

[1]GridFTP plans to support variable-sized striping in the future.

bles (DHTs) such as Pastry [101], or Chord [107]. These provide a useful abstraction for distributed data management, but do not in themselves support transfers of large amounts of data, nor provide a single point at which transfer optimization can occur. DHTs suffer problems with high lookup latency, hot spots, and data consistency, although these are actively studied. We discussed these issues in detail in Section 5.3.3.

KaZaA [59] builds upon a variant of the roughly tree-structured FastTrack network. Searches and results propagate up and down the tree, while data is downloaded in parallel from nodes which respond. In contrast, **Gnutella** [97] is a weakly-structured, decentralized overlay network, which uses flooding for searches. This limits scalability but makes the network very fault tolerant. Such peer-to-peer overlay networks do not include mechanisms for global bandwidth control or optimization, as they target individual node performance: many-to-one block downloads for whole file replication. CEP supports many-to-many communication, global optimization, and partial-file replication.

Freenet [26] uses key-based routing, caching, encryption, and other mechanisms. It focuses on ways to allow users to anonymously publish and retrieve data, rather than performance. Freenet is still under development so final evaluation is impossible, however given these design decisions we expect CEP will provide much better performance and transfer scalability.

9.3.2 Content Distribution Networks

Content distribution networks provide a one-to-many communication channel for whole-file distribution of popular content. Common approaches include building a multicast tree/mesh overlay or using special caches. Unfortunately, such approaches do not support CEP's partial-file sharing or rich semantics, and typically have problems with heterogeneity or node failure.

Scalable Reliable Multicast (SRM) [39] provides scalable one-to-many whole-file transfer of static data. They use a simple timeout scheme with feedback and randomized back-offs. SRM tolerates packet loss, provides a congestion control mechanism, and supports various underlying topologies. They do not support partial-file sharing or show scalability as with CEP.

Bullet [64, 65] is a mesh-based multicast network; in effect it adds cross-links to a multicast tree, allowing more block retrieval choices and room for optimization. Bullet works by striping requests across multiple nodes and pre-distributing data randomly across the mesh. While similar to the striping mechanisms in CEP, Bullet focuses on TCP friendliness, special encoding, and state distribution problems. Bullet provides good download performance and inter-node fairness. Again, it is solving a more limited problem than CEP: only whole file replication.

Commercial content distribution networks such as **Akamai** [3] typically employ a set of well-connected machines distributed across the network. These nodes replicate, typically via multicast, and cache the source data. Clients have a 1-1 download relationship with the closest cache. Akamai uses over "14,000 servers in 1,100 networks in 65+ countries" for this purpose. The **Open Media Network** [81] is a similar CDN for distribution of free digital video. It uses **Kontiki** [63], a "secure, commercial alternative to BitTorrent" as the underlying distribution mechanism. Such networks are an engineering feat, but CEP provides good transfer performance and more powerful partial-file semantics without this infrastructure.

Coral [45] is a CDN constructed from a set of caches and a name server. Data is published to the CDN implicitly. Users request a specially mangled URL, which is resolved by the Coral name servers and forwarded to a local coral cache. That cache uses a "Sloppy DHT" mechanism to efficiently retrieve a copy of the desired data and forward it to the client. Coral's sloppy DHT searches in concentric 'rings' roughly correlated with node location; this better exploits node locality, reduces load on the original server, and improves performance. Structurally, this architecture of centralized metadata server (Coral DNS) and distributed data transmission (Caches) is similar to CEP's implementation. However, Coral again targets whole-file content distribution rather than the rich partial-file transfers and global optimization supported by CEP.

Lastly, **BitTorrent** [29] is used for content distribution, but structurally uses simple peer lists and heuristics rather than creating a full overlay network. We have discussed BitTorrent and its limitations throughout this dissertation, starting in Section 1.1.3.

9.4 Supplemental Technology

This section discusses supplemental technology useful in combination with CEP or might easily be confused with features found in CEP. This includes work on high performance point-to-point transfer protocols, and other high speed hardware/software. Each of these provides features which CEP can exploit. While discussion of erasure or network coding techniques would fit here, we have already covered that material in Section 4.4.

9.4.1 High-Speed Transport Protocols

TCP, originally designed in the 1970s, has problems on today's networks. These include poor performance on high-latency paths, those with cross traffic, those with high loss (e.g. wireless) or those with large bandwidth$\times$delay products. There is a variety of work seeking to improve transfer performance with TCP variants or reliable UDP protocols. As CEP needs only a reliable in-order transport, it can exploit this work to improve performance.

The **FAST** [56] protocol is a popular rate-based variant extending work on TCP Vegas [15]. It provides stable, high bandwidth transfers on high bandwidth-delay networks. **HS-TCP** [41] is another variant which effectively does slow-start at all times when the TCP congestion window is large. This maintains performance on high bandwidth-delay networks, but may induce loss and network instability.

The **UDP-based Data Transfer** protocol (UDT) [49], is a rate-based protocol over

UDP which offers high performance and smooth rate transitions. This is good for fairly static networks but may have issues in highly dynamic configurations.

Reliable Blast UDP (RBUDP) [52], is a simple mechanism for transmitting large amounts of data (a blast) over UDP and then recovering losses. It uses epochs and rate control to minimize metadata transmissions. This aggressive behavior performs well on private/reserved links, but with competing traffic it will produce congestion.

The **Group transport Protocol** (GTP) [126], is a rate-based UDP scheme for fairly allocating traffic across concurrent downloads. It is a many-to-one mechanism where each parallel stream is allocated min-max fair bandwidth. No connection need be aware of any other, and performance is smooth and quickly adapting. The current version of CEP can utilize GTP as an underlying transport when such fairness is a goal.

Lastly, **Globus XIO** is a communication library meant to provide a simple open-read-write-close interface on top of a variety of different underlying protocols– such as those discussed in this section. CEP supports an XIO network stack, and hence can exploit any protocol that provides an XIO interface (See Chapter 6).

9.4.2 High-speed Hardware/Software Libraries

Lastly, some high-speed hardware and software both motivates CEP and includes similar functionality. Hardware vendors provide libraries for message passing (e.g. MPI), rDMA, synchronization, etc. which enables large-scale many-to-many transfers. Similarly, many networks include multi-protocol label switching or optical links with switchable wavelengths; software to schedule these links is similar to CEP.

Quadrics [88] is the “dominant interconnect technology in the world’s top 10 supercomputers” and provides provides rDMA and other features as mentioned above. **Myrinet** [13] is another popular cluster interconnect which allows passing arbitrary length messages (similar to CEP’s segments) between nodes. Neither provide transfer scheduling, fault tolerance, or wide-area transfers like CEP.

Infiniband [22] is the newest high-performance interconnect, designed to be a serial, switched network. The algorithms for setting up the switches and routing are similar to ideas in CEP and include parallelism despite the serial design. Infiniband does not perform transfer scheduling or work in the wide area, although there is work in this direction.

Cheetah [118], the Circuit-switched High-speed End-to-End Transport ArcHitecture, preallocates point-to-point circuits to support large file transfers. Derived from work with SONET and network engineering, it provides weak scheduling and reservation features. It does does not support many-to-many communication or any other CEP feature. **BigBangWidth** [75] similarly provides hardware and software to support automatic detection and offload of large TCP flows onto an optical circuit. CEP can efficiently use the dedicated circuits created by such software to improve transfer performance.

9.5 Summary and Conclusion

This chapter has shown the work most similar or otherwise relevant to the ideas developed in this dissertation. We have included work that initially seems to provide a competing approach, but is actually complementary. We have shown that CEP provides a richer many-to-many partial-file distribution model and better performance than related work.

Chapter 10: Conclusion

This chapter provides a summary of the claims we have made in this dissertation: *high resolution*, *simplicity/flexibility*, *high performance*, *efficiency*, *scalability*, *robustness*, and *generality*. We discuss each claim and the evidence provided in support of it. We show that we surpass common software such as BitTorrent, Apache, DHTs, or GridFTP on these metrics, and give our final conclusions.

10.1 Summary of Claims and Evidence

We claimed CEP provides **high resolution**: nodes can transfer any arbitrary set of bytes, not necessarily in blocks, not necessarily a whole file. At the core of our transfer scheduling algorithms is a graph-based canonical form (§ 3.3) using byte ranges. All the algorithms developed in Chapter 3 and 4 accept arbitrary data constraints in this way. We have also shown good performance under different levels of sharing– from whole files down to disjoint portions of a file (§ 7.10.1). In contrast, other techniques provide weaker whole-file (§ 9.3.2) or block-based transfers (§ 9.2.2).

We claimed CEP is **simple/flexible**: good interfaces exist for describing user and system constraints. We described a variety of straightforward APIs tailored for different purposes (§ 5.2). The low-level (§ 5.2.1), file transfer (§ 5.2.2), sockets (§ 5.2.3), weakly-constrained transfer (§ 5.2.4), and block interface (§ 5.2.5) APIs are each appropriate for a specific environment. Similarly, we provide a library (§ 6.2) with few dependencies that makes this functionality available to user programs. CEP also has few "knobs" and appropriately tuned default values, compared to the variety of tuning parameters in, e.g., BitTorrent (§ 7.9). Other flexibility claims are addressed in our discussion of system generality, below.

We claimed CEP has **high performance**: transfers converge to bandwidth and latency near hardware/user limits. We can compose nodes to achieve high bandwidth, over 30Gbps on local clusters (§ 7.3.1). We provide 4-8$\times$ higher bandwidth than traditional content distribution (§ 7.10.1), with high inter-node fairness; .99 using Jain's measure (§ 7.8). The system has an 10$\times$ lower latency than other approaches (§ 7.10.2), particularly those using nonsystematic erasure codes (§ 4.4).

In general, Chapter 4 shows that our algorithms execute rapidly, and Chapter 7 shows that these schedules perform well. In particular, the greedy algorithm output is equivalent to the the known optimal LP output for these environments (§ 7.3.1, 7.6.1, 7.6.2). The system has a high performance design (§ 5.3, 6.4) and implementation (Chapter 6), particularly the core message passing protocol (§ 7.5.3); with only about 4% overhead (§ 7.3.1). Lastly, for the special case of hybrid networks (Chapter 8) we have shown high bandwidth, within 1% of ideal (8.3.2), low latency, with 99% of blocks recovered within 3 satellite RTTs (8.3.3), and high fairness, above .999 using Jain's measure (8.3.6).

We claimed CEP is **efficient**: computation time is worst-case $O(n^2)$, commonly $O(n\ log\ n)$, and query response in $O(1)$. Sections 3.1-3.3 show conversion to canonical input form commonly takes time $O(n\ log\ n)$. In the worst case it may take time $O(n^2)$, but limits on segment size bound it to $O(n\ log\ n)$. This is an unavoidable tradeoff between generality and performance. Chapter 4 showed several algorithms to efficiently solve transfer scheduling problems. The core greedy algorithm (§ 4.3) runs in time linear in the number of edges given canonical-form input, and nodes' queries can be handled in $O(1)$. In contrast, a problem which takes less than a second for the greedy algorithm to solve would take over 23 hours for approaches using linear programming (§ 7.4). While some sub-problems are NP-complete, we can approximate them arbitrarily well in most cases (§ 3.4). Finally, on hybrid satellite/terrestrial networks, we achieve 75% or better transfer efficiency (§ 8.3.1, 8.3.4) and fast metadata updates (§ 8.3.5).

We claimed CEP is **scalable**: the system works for transfers involving tens of thousands of nodes and 10Gbps+ links. Our efficiency results above show that the core system algorithms are efficient enough to run for large problem sizes; the greedy algorithm can schedule a transfer of 100,000 nodes in less than 1 second. (§ 7.4) We have shown network scalability to over 100,000 requests/second; 10× better than DHT implementations, 40× better than Apache, and that the memory required was reasonable– less than 256MB (§ 7.5). We have achieved over 30Gbps on real local clusters, up to 1Tbps in simulation (§ 7.3.1), and over 10Gbps in the wide area (§ 7.3.2). Finally, we showed that the hybrid network approach scales in terms of file sizes and to thousands of nodes (§ 8.3.2).

We claimed CEP is **robust**: failures which do not eliminate data required by another peer are tolerated. We have shown that we can tolerate real world server failures with only a 2% performance penalty (§ 7.7.1), and inaccurate metadata with negligible performance penalty (§ 7.7.2). We perform flawlessly in environments where 20% of BitTorrent peers fail (§ 7.10.3). Our regression testing framework (§ 6.4) enhances reliability in practice, and we have shown how techniques such as erasure coding could be used to enhance data robustness (§ 4.4). For hybrid satellite/terrestrial networks, we have also shown tolerance of loss rates as high as 10% (§ 8.3.1).

Lastly, we claimed CEP is **general**: we produce desirable results in a wide variety of environments and user constraints. We have shown that we can exploit nodes' capacity differing by an order of magnitude (§ 7.6.1) and data layout constraints varied from disjoint to total overlap (§ 7.6.2); achieving 4-8× BitTorrent's bandwidth for low and high sharing configurations (§ 7.10.1). We achieve bandwidth near hardware capacity in both high performance and peer-to-peer configurations (Chapter 7) as well as hybrid satellite/terrestrial networks (Chapter 8). We support stand-alone and application-integration (§ 6.2.2), with APIs supporting different use models (§ 5.2); other approaches target a single API or use model (Chapter 9). We support a variety of different underlying protocols, including TCP, GTP, and those implemented using the XIO framework (§ 6.3). We support a variety of hardware; CEP has been tested on x86, Opteron, and PowerPC machines (§ 6.4.1).

Returning to our main thesis statement: fully utilizing high speed links (§ 2.2.1, 7.3.1) for large, complex transfers (§ 2.2.2, 7.6.2) requires metadata management infrastructure (§ 5.3) and simultaneous transfers between multiple nodes (§ 7.3.1). Composite endpoints using hybrid centralized/decentralized transfer scheduling (Chapter 4), via graph-structured algorithms (§ 3.3) and feedback heuristics (§ 4.5), provide a general, high-performance and robust approach (see above).

10.2 Future Work

While we have met our goals– performance and a rich transfer model– all research exposes further interesting questions. This section discusses ways in which this work could be extended or improved, and potential future research topics.

First, consider **transfer extensions**. There are a wealth of possible techniques that have not been deeply explored. These include prefetching, such as that done by LamdaRam [66], data compression with well-known algorithms, such as LZW [123] or block-sorting with Huffman coding [17, 103]. Other techniques include transfer of deltas for mutable data, cross-flow data caching, or weakening of the transfer semantics to allow "super nodes" to serve data in which they were not otherwise interested.

Each of these techniques has the potential to dramatically improve performance for certain types of applications, but how to apply them effectively in a distributed transfer system such as CEP is an open question. Do prefetching algorithms designed for disks scale to the wide-area? How large must data transfers be and how slow must the network be to justify compression? How best to encode compressed data so that portions of it can be forwarded independently, as may be necessary given our rich sub-file transfer model? Is avoiding cross-constraint dependencies (see page 65) enough? How would this impact the use of erasure coding? What about integrating a multicast tree/mesh for distribution of highly shared sub-file ranges? What are the trade-offs using super nodes; when is the extra copy worthwhile? Can they be used to provide TCP connection splitting [98] functionality?

Second, consider **system interaction**. At a high level a CEP transfer may look a lot like a denial of service attack. While CEP provides TCP-fairness at the transport level, its scheduling mechanisms do not guarantee fairness *in aggregate* when competing with other traffic. Individual transport flows may exhibit complex dynamic interactions on peculiar networks: CEP works well in all the environments we have tested, but competitive and malicious environments may show other behavior.

Can we guarantee stability or fairness? How does the system interact with multiple users competing maliciously? What happens when multiple transport protocols are being used concurrently, e.g. TCP, GTP, and RBUDP? Can CEP's constraint mechanism be used to ensure proper network behavior? Can it give quality-of-service guarantees itself or be integrated with systems providing QoS functionality? Can traffic be tunnelled through a constrained CEP transfer as a network engineering tool? Can cross-CEP-transfer communication be used to provide strong semantics (fairness, performance) across composite endpoints? What about structured or hierarchical aggregation of composite endpoints into increasingly complex workflow-like graphs? Can the algorithmic results presented here be extended to provide stronger guarantees under these weaker assumptions? While we have evidence hinting at answers, these remain open questions.

Finally, the question of **distribution**– one we had hoped to address in greater depth. Can partial-content transfer tasks be distributed yet guarantee the generality and performance for *partial* content distribution? How do we efficiently perform distributed sub-file matching without using blocks? Can we do so in such a way that we gain scalability or increase robustness? Current mechanisms such as primary/backup or clustering with voting (e.g. Paxos [70]) fall back to using a single node (the primary, or distinguished learner) for performance reasons. CEP has tried to provide a distributed transfer abstraction similar to the distributed hash table abstraction; is there a better underlying model or system design to accomplish this?

10.3 Final Remarks

This dissertation has described the interrelated problems experienced when trying to transfer data at high speeds with a rich interface. We have addressed problems with capacity: achieving scalability, high bandwidth, and low latency; problems with complexity: achieving a simple, flexible interface without too many user knobs; problems with crashes: robustly surviving system faults, errors, and inaccurate data, and done so in a comprehensive, general fashion. Our analysis and evaluation show that CEP successfully addresses each of these problems and provides all of the desired features.

The core of this dissertation is the set of graph-structured transfer scheduling algorithms and their implementation. We have provided a rigorous definition of this problem and its complexity, as well as several solutions to it. We have shown efficient, high-performance transfer schedulers; analyzed from a theoretical perspective with implementations tested empirically. Comparison with related work (e.g BitTorrent) shows great performance improvement.

The ideas in this work, implemented in CEP, provide a mechanism allowing multiple processes to join in a single logical connection. Users specify constraints at a high level through a simple, flexible interface, and the system runs the desired high performance transfers. CEP allows one to terminate disproportionately large network transfers on relatively weak nodes, efficiently utilize heterogeneous hardware in an arbitrary configurations, and gracefully tolerate errors and failures.

Bibliography

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] John Hoffman (a.k.a. TheShad0w). BitTornado. http://www.bittornado.com/.

[3] Akamai Technologies Inc. Akamai content distribution system. http://www.akamai.-com/.

[4] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high performance computational grid environments. In *Parallel Computing: Advances and Current Issues*, 2001.

[5] Bill Allcock, John Bresnahan, Rajkumar Kettimuthu, and Joseph Link. The Globus extensible input/output system (XIO): a protocol independent IO system for the grid. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2005.

[6] Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Kai Li. Design and implementation of NX message passing using Shrimp virtual memory mapped communication. In *Proceedings of the International Parallel Processing Symposium*, volume 1, pages 111–119, April 1996.

[7] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludäscher, and Steve Mock. Kepler: An extensible system for design and execution of scientific workflows. In *16th Intl. Conf. on Scientific and Statistical Database Management*, pages 21–23, June 2004.

[8] AT&T. Global IP network. http://ipnetwork.bgtmo.ip.att.net/pws/index.html.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles*, October 2003.

[10] Alex Bassi, Micah Beck, Erika Fuentes, Terry Moore, and James S. Plank. The logistical file system: A network file system designed for scalable resource sharing. Technical Report UT-CS-00-469, University of Tennessee, August 2001.

[11] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[12] BIRN. Biomedical informatics research network. http://www.nbirn.net.

[13] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local-area network. *IEEE Micro*, January-February 1995.

[14] Mic Bowman, Jeff Sedayao, and Rick McGeer. Bakeoffs. http://www.planet-lab.-org/Talks/2005-05-01/Bakeoffs_final.ppt, Sep 2005.

[15] Lawrence Brakmo and Larry Peterson. TCP vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communication*, 13(8):1465–1480, October 1995.

[16] Robert G. Brown. *Engineering a Beowulf-style Compute Cluster*. Duke University Physics Department, May 2004.

[17] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Systems Research Center, May 1994.

[18] John W. Byers, Michael Luby, and Michael Mitzenmacher. Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads. In *INFOCOM*, pages 275–283, New York, NY, march 1999. IEEE.

[19] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM*, pages 56–67, 1998.

[20] CANARIE. CAnet 4. http://www.canarie.ca/canet4/, 2002.

[21] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, October 2000.

[22] Daniel Cassiday. Infiniband architecture tutorial. Hot Chips 12 Tutorial, August 2000.

[23] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. One ring to rule them all: Service discovery and binding in structured peer-to-peer overlay networks. In *Proceedings of the SIGOPS European Workshop*, Sep 2002.

[24] SIO Visualization Center. OptIPuter at the SIO Viz center. http://www.siovizcenter.-ucsd.edu/optiputer/index.html.

[25] Wu chun Feng, Justin (Gus) Hurwitz, Harvey Newman, Sylvain Ravot, R. Les Cottrell, Olivier Martin, Fabrizio Coccetti, Cheng Jin, Xiaoliang (David) Wei, and Steven Low. Optimizing 10-gigabit ethernet for networks of workstations, clusters, and grids: A case study. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, 2003.

[26] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, pages 46–66, 2001.

[27] Cluster File Systems Inc. Lustre: scalable, secure, robust, highly-available cluster file system. http://www.lustre.org, 2006.

[28] Kerry G. Coffman and Andrew M. Odlyzko. Internet growth: Is there a 'Moore's Law' for data traffic? http://www.research.att.com/∼amo/doc/internet.moore.pdf.

[29] Bram Cohen. The BitTorrent file sharing protocol. http://bittorrent.com/.

[30] Bram Cohen. Avalanche. http://bramcohen.livejournal.com/20140.html, June 2005.

[31] Internet2 Consortium. Abilene: Advanced networking for leading-edge research and education. http://abilene.internet2.edu/, 2007.

[32] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, editors. *Introduction to Algorithms*. McGraw Hill, 2nd edition, 2002.

[33] NEC Corporation. NEC DWDM transmission capacity record. http://www.nec.co.-jp/press/en/0010/0403.html, October 2000.

[34] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison Wesley, August 2000.

[35] Beman Dawes, David Abrahams, and Rene Rivera. Boost C++ libraries. http://-www.boost.org/.

[36] Erik DeBenedictis and Juah Miguel del Rosario. nCUBE parallel I/O software. In *11th International Phoenix Conference on Computers and Communications*, pages 117–124, April 1992.

[37] Randall Dougherty, Chris Freiling, and Kenneth Zeger. Unachievability of network coding capacity. *IEEE Transactions on Information Theory*, 52(6):2365–2372, June 2006.

[38] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Transactions of the ACM*, 19:248–264, 1972.

[39] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, Dec 1997.

[40] Sally Floyd. Measurement studies of end-to-end congestion control in the internet, 2002. http://www.icir.org/floyd/ccmeasure.html.

[41] Sally Floyd. Highspeed tcp for large congestion windows, 2003. http://www.icir.-org/floyd/hstcp.html.

[42] Lester R. Ford and Delbert R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[43] Ian Foster. What is the Grid? A three point checklist. http://www-fp.mcs.anl.-gov/∼foster/Articles/WhatIsTheGrid.pdf, July 2002.

[44] The Apache Software Foundation. The apache HTTP server project. http://httpd.-apache.org/.

[45] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[46] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Freeman, 1979.

[47] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[48] Chris Gottbrath, Jeremy Bailin, Casey Meakin, Todd Thompson, and J.J. Charfman. The effects of moore's law and slacking on large computations. Technical report, Steward Observatory, University of Arizona, 2000. http://www.gil-barad.-net/∼chrisg/mooreslaw/Paper.html.

[49] Yunhong Gu and Robert L. Grossman. UDT: UDP-based data transfer for high-speed wide area networks. *The International Journal of Computer and Telecommunications Networking*, 51(7):1777–1799, May 2007.

[50] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To infinity and beyond: Time-warped network emulation. In *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006)*, May 2006.

[51] Evan Hansen. Google wants "dark fiber", January 2005. http://news.com.com/-Google+wants+dark+fiber/2100-1034_3-5537392.html.

[52] Eric He, Jason Leigh, Oliver Yu, and Thomas A. DeFanti. Reliable blast UDP: predictable high performance bulk data transfer. In *Proceedings of IEEE International Conference on Cluster Computing*, pages 317–324, 2002.

[53] Ian Holyer. The NP-completeness of edge-coloring. *SIAM Journal on Computating*, 10:718–720, 1981.

[54] Red Hat Inc. Global file system. http://www.redhat.com/software/rha/gfs/.

[55] Rajendra K. Jain, Dah-Ming W. Chiu, and William R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared systems. Technical Report TR-301, Digital Equipment Corporation, Littleton, MA, 1984.

[56] Cheng Jin, David Wei, Steven H. Low, Gary Buhrmaster, Julian Bunn, Dawn H. Choe, R. Les Cottrell, Johe C. Doyle, Harvey Newman, Fernando Paganini, Sylvain Ravot, and Suresh Singh. FAST kernel: Background theory and experimental results. In *Proceedings of the First International Workshop on Protocols for Fast Long-Distance Networks*, 2003. http://netlab.caltech.edu/pub/papers/pfldnet.pdf.

[57] Edward G. Coffman Jr., Michael R. Garey, David S. Johnson, and Andrea S. LaPaugh. Scheduling file transfers in a distributed network. In *Second Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 254–266, 1983.

[58] Stamatis V. Kartalopoulos. Elastic bandwidth. http://www.ieee.org/organizations/-pubs/newsletters/leos/apr02/elastic.html, April 2002.

[59] KaZaA. KaZaA file sharing network, 2002. http://www.kazaa.com.

[60] Dan Kegel. The C10K problem. http://www.kegel.com/c10k.html, September 2006.

[61] Peter Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter Technical Conference*, January 1994.

[62] Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: Language support for building distributed systems. In *Proceedings of PLDI (PLDI 2007)*, June 2007.

[63] Inc. Kontiki. Kontiki video on demand. http://www.kontiki.com/.

[64] Dejan Kostic, Ryan Braud, Charles Killian, Erik Vandekieft, James W. Anderson, Alex C. Snoeren, and Amin Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proceedings of USENIX 2005*, April 2005.

[65] Dejan Kostic, Alex Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP 2003)*, volume 37, pages 282–297, October 2003.

[66] Naveen K. Krishnaprasad, Venkatram Vishwanath, Shalini Venkataraman, Arun G. Rao, Luc Renambot, Jason Leigh, Andrew E. Johnson, and Brian Davis. JuxtaView - a tool for interactive visualization of large imagery on scalable tiled displays. In *IEEE International Conference on Cluster Computing*, pages 411–420, September 2004.

[67] Oak Ridge National Laboratory. The linpack benchmark. http://www.netlib.org/-linpack/.

[68] T. V. Lakshman and Upamanyu Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking*, 5(3):336–350, June 1997.

[69] National LambdaRail. National LambdaRail: light the future. http://www.nlr.net.

[70] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[71] Jae-Yong Lee and Alan Sussman. High performance communication between parallel programs. In *Proceedings of 2005 Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models (HIPS-HPGC 2005)*. IEEE Computer Society Press, April 2005. Appears with the Proceedings of IPDPS 2005.

[72] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[73] Shuo-Yen R. Li, Raymond W. Yeung, and Ning Cai. Linear network coding. *IEEE Transactions on Information Theory*, 49(2):371–381, February 2003.

[74] LionShare Project. LionShare: Connecting and extending peer-to-peer networks, October 2004. http://lionshare.its.psu.edu/.

[75] Stuart Lomas. Breaking the limits of packet switching. BigBangwidth Incorporated, March 2003.

[76] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical loss-resilient codes. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of Computing*, pages 150–159, 1997.

[77] David Mazières. A toolkit for user-level file systems. In *Proceedings of Usenix Technical Conferences*, pages 261–274, June 2001.

[78] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, MPI Forum, 1994.

[79] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(2), April 1965. http://www.intel.com/research/silicon/moorespaper.pdf.

[80] NCMIR. National center for microscopy and imaging research. http://ncmir.ucsd.-edu.

[81] Open Media Networks. Open media network: Media that matters. http://www.omn.-org/.

[82] NLANR Distributed Applications Support Team. Iperf TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/.

[83] University of California Santa Barbara. The network weather service. http://nws.-cs.ucsb.edu/.

[84] University of Chicago. The Globus toolkit. http://www.globus.org/toolkit/.

[85] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of USENIX Technical Conference*, June 1999.

[86] Christos Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[87] Andrew Parker. P2P in 2005. Technical report, Cache Logic, 2005. http://www.-cachelogic.com/research/p2p2005.php.

[88] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The quadrics network: High-performance clustering technology. *IEEE Micro*, January-February 2002.

[89] Andrew A. Chien (PI). The FWGrid project. http://fwgrid.ucsd.edu/.

[90] Pittsburgh Supercomputing Center. Enabling high-performance data transfers on hosts. http://www.psc.edu/networking/perf_tune.html.

[91] PlanetLab Consortium. Planetlab: An open platform for developing, deploying, and accessing planetary-scale services. http://www.planet-lab.org/.

[92] The Globus Project. Gtp 4.0 reliable file transfer (rft) service, 2005. http://www.-globus.org/toolkit/docs/4.0/data/rft/.

[93] Public Broadcasting Service (PBS). PBS — Station Finder. http://www.pbs.org/-stationfinder/index.html.

[94] IBM Research and Others. Special issue on Blue Gene. *Journal of Research and Development*, 49(2/3), 2005.

[95] Sean Rhea, Byung-Gon Chun, John Kubiatowicz, , and Scott Shenker. Fixing the embarrassing slowness of opendht on planetlab. In *Proceedings of USENIX WORLDS*, December 2005.

[96] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, , and Harlan Yu. OpenDHT: A public DHT service and its uses. In *Proceedings of ACM SIGCOMM*, August 2005.

[97] Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. Technical Report TR-2001-26, University of Chicago, July 2001.

[98] Philip Rizk, Cameron Kiddle, and Rob Simmonds. Improving gridftp performance with split tcp connections. In *Proceedings of the First International Conference on e-Science and Grid Computing*, pages 263–270, 2005.

[99] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[100] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostic, and Amin Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proceedings of the USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, March 2004.

[101] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–, 2001.

[102] SDSC Cluster Development Group. ROCKS cluster distribution. http://www.-rocksclusters.org.

[103] Julian Seward. bzip2 and libbzip2. http://www.bzip.org/, 2007.

[104] Farhad Shahrokhi and David W. Matula. The maximum concurrent flow problem. *Journal of the ACM (JACM)*, 37(2):318 – 334, April 1990.

[105] Larry L. Smarr, Andrew A. Chien, Tom DeFanti, Jason Leigh, and Philip M. Papadopoulos. The OptIPuter. *ACM Blueprint for the future of high-performance networking*, 46(11):58–67, November 2003.

[106] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, 1998.

[107] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM 2001*, pages 149–160, Aug 2001.

[108] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *To appear in IEEE/ACM Transactions on Networking*, 2005.

[109] Vaidy S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[110] Nut Taesombut and Andrew A. Chien. Distributed virtual computer (DVC): Simplifying the development of high performance grid applications. In *Proceedings of the Workshop on Grids and Advanced Networks (GAN04)*, April 2004. Held in conjunction with the IEEE Cluster Computing and the Grid (CCGrid2004) Conference.

[111] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, January 2002.

[112] Osamu Tatebe, Youhei Morita, Satoshi Matsuoka, Noriyuki Soda, and Satoshi Sekiguchi. Grid datafarm architecture for petascale data intensive computing. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, pages 102–110, 2002.

[113] Brian Tierney. TCP tuning guide for distributed applications on wide-area networks. In *USENIX & SAGE Login*, http://www-didc.lbl.gov/tcp-wan.html, February 2001.

[114] Peerapol Tinnakornsrisuphap, Wu-chun Feng, and Ian Philp. On the burstiness of the TCP congestion-control mechanism in a distributed computing system. *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS'00)*, April 2000.

[115] Aad J. van der Steen and Jack J. Dongarra. Overview of recent supercomputers, 2004. http://top500.org/ORSC/2004/.

[116] Various. ns, the network simulator, version 2.29. http://www.isi.edu/nsnam/ns/.

[117] Various authors. The OptIPuter project. http://www.optiputer.net.

[118] Malathi Veeraraghavan, Xuan Zheng, Wu chun Feng, Hojun Lee, Edwin K.P. Chong, and Hua Li. Scheduling and transport for file transfers on high-speed optical circuits. *Journal of Grid Computing*, 1(4):395 – 405, December 2003.

[119] András Veres and Miklós Boda. The chaotic nature of TCP congestion control. In *Proceedings of IEEE Infocom 2000*, March 2000.

[120] Eric Weigle and Andrew A. Chien. The composite endpoint protocol (CEP): Scalable endpoints for terabit flows. In *Proceedings of IEEE Conference on Cluster Computing and the Grid (CCGRID)*, May 2005.

[121] Eric Weigle and Andrew A. Chien. Partial content distribution on high performance networks. In *Proceedings of the IEEE International Symposium on High-Performance Distributed Computing (HPDC2007)*, June 2007.

[122] Eric Weigle, Matti Hiltunen, Rick Schlichting, Vinay A. Vaishampayan, and Andrew A. Chien. Peer-to-peer error recovery for hybrid satellite-terrestrial networks. In *Proceedings of 6th IEEE International Conference on Peer-to-Peer Computing*, September 2006.

[123] Terry A. Welch. A technique for high-performance data compression. *Computer*, 17:8–19, June 1984.

[124] Wikipedia. Comparison of BitTorrent software. http://en.wikipedia.org/wiki/-Comparison_of_BitTorrent_software.

[125] Joe Shang-chieh Wu and Alan Sussman. Flexible control of data transfers between parallel programs. In *Proceedings of the Fifth International Workshop on Grid Computing (GRID 2004)*, pages 226–234. IEEE Computer Society Press, November 2004.

[126] Xinran Wu and Andrew A. Chien. GTP: Group transport protocol for lambda-grid. In *Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid*, April 2004.