Cooper: Expedite Batch Data Dissemination in Computer Clusters with Coded Permutation Gossips

by

Yan Liu

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Department of Electrical and Computer Engineering
University of Alberta

Abstract

Data transfers happen frequently in server clusters for software and application deployment, and in parallel computing clusters to transmit parameters in batches among servers between computation stages. This thesis presents *Cooper*, an optimized prototype system to speedup multi-batch data transfers among a cluster of servers, leveraging a theoretically proven optimal algorithm called "permutation gossip" which employs randomly permuted node connections to best utilize bandwidth and random linear code to maximize the useful information transmitted. By chunking the file into a proper number of blocks, we present a pipelining technique to parallelize the coding operation and network transfer on the process level, realizing the theoretically promised benefits of random linear codes. More importantly, for batch-based or multi-file transfers, we propose priority-based scheduling algorithms to overlap the transfers of different batches, which further reduce the transfer finish time of each batch, while only delaying the first batch for a constant time. We present an asynchronous and distributed prototype implementation of *Cooper* and deploy it on Amazon EC2 for evaluation. Based on results from real experiments, we show that *Cooper* can significantly speedup data transfers and reduce redundant transmissions in server clusters as compared to state-of-the-art content distribution tools, including BitTorrent and an optimized random-block transfer strategy based on buffer negotiation in a wide range of practical settings.

Acknowledgements

As the time flying away, the years shuttling again. It has been two years since I have been studying and working in University of Alberta. No double, it will be a memorable experience concert trip in my entire life. Here, I would like to present my acknowledgement to all these that help me during these two years.

Firstly and most importantly, I would like to highly appreciate my supervisor Dr. Di Niu, for offering me the opportunity to study in UofA, for his advice and guide on my research work, and for all his help in these two years. Besides, I would like to thank Dr. Majid Khabbazian, for his innovate theoretical support for this thesis. Also, I would like to thank all my course instructors for opening a window for me to learn advanced theory and technologies.

Finally, I would like to thank my parents, for their support and understanding on my decisions, and also for their selfless love and devotion since I was born.

Contents

List of Figures

Abbreviations

Cooper	**CO**ding **O**n **PER**mutation **T**opologies
RLNC	**R**andom **L**inear **N**etwork **C**oding
LAN	**L**ocal **A**ear **N**etwork
RB	**R**andom **B**lock
BT	**B**it**T**orrent

Symbols

k	number of blocks	
$k*$	the right number of blocks that achieving the lowest broadcast finish time	
n	number of nodes	
c	a constant	
C	a constant	
q	parameter for field size	
F	file size	MB
B	block size	MB
W	download and upload capacity	Mbps
T	time span of each round	
α β	positive constants when calculating computation or transmission time	

Chapter 1

Introduction

Data transfers frequently happen in server clusters comprised of loosely or tightly connected computers that work together to conduct scientific experiments, perform enterprise operations, or accomplish parallel computing tasks. For example, in a university laboratory, educational program and software may need be distributed to every computer for student lab sections, while on another day all the laboratory servers may need be reinstalled with customized applications to run scientific experiments. As another example, in iterative parallel computing clusters such as *Spark* [2] and *Dryad* [3], which are designed to solve machine learning tasks in parallel, large amounts of parameters and intermediate calculations must be transferred among servers between computation stages, as shown in Fig. 1.1. Measurements [1] show that data transfers can consume $40\% - 50\%$ of the total algorithm running time in such computing clusters, for tasks like movie recommendation through collaborative filtering [4], and spam link identification on Twitter [5]. In all the abovementioned jobs, an efficient content distribution utility is needed to expedite data dissemination among a cluster of servers.

This thesis presents *Cooper* (COding On PERmutation topologies), an optimized prototype system to speedup data dissemination in server/computing clusters. *Cooper* arranges all nodes in time-varied random permutations so that each node transmits a coded block to its successor by randomly combining all the blocks it holds using random linear network coding (RLNC) [6]. Since computers in a cluster are usually connected

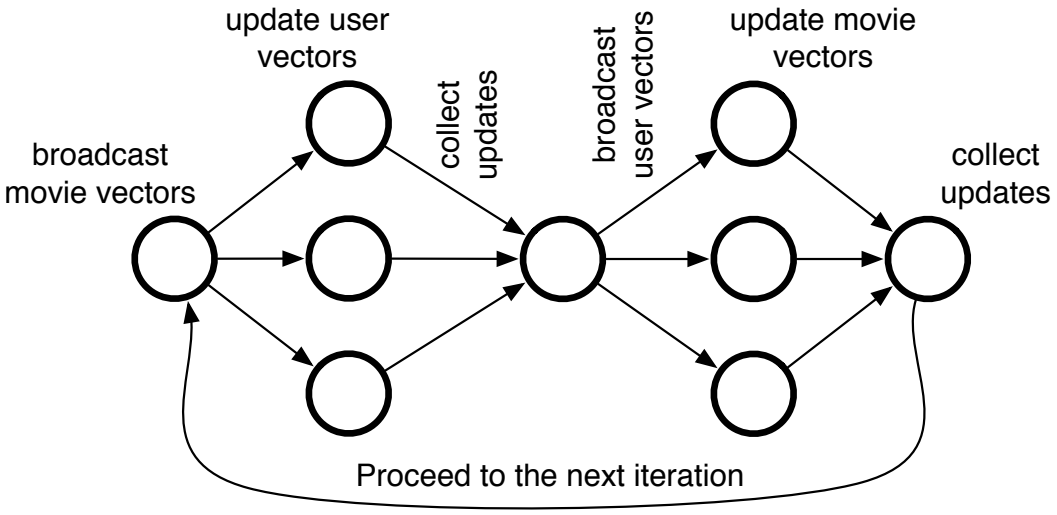

FIGURE 1.1: The workflow of a collaborative filtering algorithm in a computing cluster [1].

through fast local area networks (LAN) with homogenous and easily configurable network and disk I/O capacities, permuted connections are not only feasible, but also able to utilize network bandwidth optimally.

In fact, prior theoretical studies have analyzed the time of broadcasting k blocks from a single source to $n-1$ nodes, where each node can upload 1 block and download 1 block per round. It is well known that the optimal broadcast finish time in such a scenario is $k + \lceil \log_2 n \rceil$, which can be achieved by a fully centralized and thus impractical sender-receiver pairing and block selection schedule [7]. In comparison, with coded permutation gossip, where a random permutation of all nodes is formed in each round and each node transmits a coded block to its immediate successor, the broadcast can finish within time $k + O(\log n + \log k)$ with high probability. Simulations further show that the coded permutation gossip finishes within time $k + \lceil \log_2 n \rceil + C$ [8], with $C \approx 4$, which is close to the theoretical limit of $k + \lceil \log_2 n \rceil$ rounds. Therefore, *Cooper* enjoys the advantages of gossiping protocols in terms of robustness and ease to manage, while still achieving near-optimal utilization of network resources as in tree-packing structured multicast.

This thesis takes one step further to implement and optimize the coded permutation gossip in reality, while asking the following questions: 1) With encoding and decoding latencies considered, can coded transmission indeed achieve its theoretically-promised benefit and how? 2) Given a file of a fixed size, how many blocks should the file be chuncked into to achieve the best performance in practice? 3) Is it faster to distribute

a large file as a whole or divide the file into smaller batches and distribute the batches instead? 4) If multiple batches of data are distributed, can we intelligently schedule different batches so that each batch will finish earlier than if transferred sequentially one after another?

To answer these questions, this thesis incorporate multiple inventions in the design of *Cooper* to boost up performance in practice. *First*, a pipelining technique is proposed to perform coding operations and data transfers as parallel processes on each node, which successfully hides coding latency when chunking the file into a fine-tuned number of blocks. *Second*, an asynchronous transfer control model is adopted to convert the time-slotted theoretical model into real implementation. *Third*, this thesis consider multi-batch data distribution, and propose a priority-based scheduling strategy to smartly overlap the distribution of consecutive batches. Based on some derived theoretical insights, this thesis proposes a smart overlapping scheme for multi-batch scheduling to reduce the total broadcast finish time. *Fourth*, the feasibility of using off-the-shelf commodity multi-core processors is discussed on an asynchronous yet fine-tuned coding-transfer pipeline to further enhance performance.

I have prototyped *Cooper*, using *Apache Thrift*[9], a software library developed at Facebook to expedite development and implementation of efficient and scalable backend services as well as Boost C++ libraries [10] on Linux Systems. While leveraging insights from time-slotted theoretical analysis, *Cooper* does not require synchronized implementation, but allows nodes to perform encoding, decoding, local transfer controls and inter-batch scheduling in a completely *asynchronous* manner. Specifically, *Cooper* runs multiple processes on each node and exploits process-level parallelism to optimally pipeline coding operations and sending/receiving actions. In addition, it can also utilize the multi-core processors widely available on most modern servers to launch multiple light-weight encoders, which further expedite the broadcast process.

I deploy *Cooper* on Amazon EC2 [11] and perform extensive experiments to verify the various proposed theoretically inspired algorithms and demonstrate that *Cooper* can reduce the time to broadcast a single file by 40% over a state-of-the-art BitTorrent [12] system and a random block dissemination scheme with buffer negotiation in real computing clusters in the cloud. Moreover, I show the counter-intuitive result that by temporarily prioritizing later batches or files, the inter-batch scheduling algorithm

in *Cooper* can beat the more intuitive early-batch-first strategy, and significantly outperforms sequential dissemination of multiple batches, which is a common practice in today's systems.

Chapter 2

System Model and Background

Consider a theoretical model for data transfers in clusters. The server cluster over which data dissemination is performed is modeled as a network of n nodes, where each node can transmit packets to any other node through a TCP connection. The data to be disseminated is divided into k uniform-sized *blocks*. Assume the time is slotted and data transfer happens in *rounds*. Each node can send (upload) at most 1 data block to another node and receive (download) at most 1 data block from another node in each round. The reason is that servers installed in a same generation can be viewed to have homogeneous I/O capacities and network bandwidth. The objective is to disseminate all k blocks to all n nodes from a single source or multiple source nodes. Apparently, the most time-consuming case is *one-to-all* transfer or *broadcast*, where only a single source node holds all k blocks initially. In this thesis, I focus on such a challenging case of broadcast transfer, as illustrated by the example in Fig. 1.1.

The above model is more suitable to a server cluster or computing cluster than an Internet overlay network. Nodes in a server cluster usually have similar configurations of CPU, memory and network I/O. Furthermore, cluster nodes are often collocated geographically in enterprises, campus networks or datacenters, so that it is easy to estimate network condition and reserve bandwidth. These facts eliminate the need for sophisticated bandwidth allocation schemes (such as tree-packing) as required in structured multicast over a heterogeneous network and justifies the feasibility of using simple gossip-like protocols to best utilize network bandwidth.

2.1 Gossips and Permuted Gossips

It is a classical result that in broadcast transfer, if each node can upload at most 1 block to another node and download at most 1 block from another node per round, the best possible broadcast finish time is $k + \lceil \log_2 n \rceil$ rounds, which can be achieved by a fully centralized sender-receiver pairing schedule and block selection strategy [7]. Since centralized scheduling is not practical, decentralized gossip schemes have been heavily studied to approach the above limit.

The random phone call (or random contact) is the most popular model in gossiping literature [13], where in each round, each node chooses a random node as its receiver (or sender) to push (or pull) a block. Under this model [13], if each node holds one of the k blocks initially, and in each round transmits a coded block (using RLNC) to a random receiver, the finish time is $ck + O(\sqrt{k}\log(k)\log(n))$ rounds with high probability, where c is a constant between $3-6$. This bound is further tightened in [14], which proves a finish time of $k + o(k)$ for the pull version. However, these results hold only if every node holds a subset of k blocks initially, which is not broadcast transfer. More importantly, they assume each node's download capacity (or upload capacity in the pull case) can exceed 1 block per round, since there may be multiple nodes pushing to (or pulling from) a node at the same time. This assumption is hard to justify for servers in a cluster with roughly uniform capacities. On the other hand, with random phone call, some nodes may not be chosen by any senders (or receivers) in a round, which will cause link underutilization.

A new class of "controlled" gossips is described in [15], [8] based on a simple permutation rule that does not violate the node upload/download capacity of 1 block per round while achieving better link utilization:

RLNC + Random Permutation: In each round, a uniformly random permutation, $u_1, u_2, \ldots, u_n$ of all n nodes is formed, such that node u_i, $1 \leq i < n$, sends a coded block (encoded with RLNC) to node u_j where $j = i + 1 \mod n$.

The permutation rule can be easily realized in a computing cluster, where we have almost homogeneous server download/upload capacities with full control of the server connection topology. On the other hand, with permutation-based receiver selection, data transmission is still fully distributed via RLNC. Therefore, this semi-decentralized algorithm acts like a "controlled gossip", taking advantage of full control over topologies,

while avoiding the hassle of scheduling large amounts of block transfers. In other words, the coded permutation gossip enjoys the high network utilization of structured multicast as well as the robustness and ease of distributed implementation in gossip-like protocols.

In theory, Random Permutation with RLNC has dramatically reduced the broadcast finish time to $k + O(\log n + \log k)$ rounds (with high probability for a field size $q > n$), which is close to the aforementioned theoretical limit of $k + \lceil \log_2 n \rceil$, without violating the node upload/download capacity constraint. Simulation has shown that for a field size of $q = 2^8$, $k = 200$ and n varied between 20 and 300, the broadcast finish time of Random Permutation with RLNC is almost always within $k + \lceil \log_2 n \rceil + C$ rounds, with $C = 4$. In contrast, the finish time of Random Permutation with a Random Block scheme (that like BitTorrent, transmits a random block held by the sender but not the receiver) could be $30 - 50$ rounds more than the theoretical limit $k + \lceil \log_2 n \rceil$.

Chapter 3

Pipelining

In this chapter, I ask — can we still achieve the theoretically alleged benefit of coded permutation gossip in practice, if we consider encoding and decoding latencies? I show that if we arrange the coding and transmission in a computing pipeline, where the output of the coding (transmission) is the input of transmission (coding), coding and transmission can be done in parallel, with coding latency hided, when the number of blocks k reaches a sweet spot k^*.

To answer the above question, realistic parameters are assigned to the time-slotted model described in Chapter 2. Assume the file is F MB, chunked into k blocks, each of size $B = F/k$, and the download and upload capacities of each node are both W Mbps. Now the real time span of each round is $T = B/W = F/kW$. Since the broadcast finish time of Random Permutation + RLNC is always within $k + \lceil \log_2 n \rceil + C$ rounds with $C = 4$, its real finish time appears to be given by

$$\begin{aligned} T_{\mathsf{Transfer}} &= (k + \lceil \log_2 n \rceil + C) \cdot T \\ &= (k + \lceil \log_2 n \rceil + C) \cdot \frac{F}{kW} \\ &= \left(1 + \frac{\lceil \log_2 n \rceil + C}{k}\right) \frac{F}{W} \end{aligned}$$

When the file size F, bandwidth W, number of nodes n are fixed, the larger the k is, the smaller the finish time. When $k \to \infty$, surprisingly, the finish time of the last node *and every node* will be F/W. This means that when $k \to \infty$, each node can download the file from the source as if there are no other nodes competing for bandwidth. In

comparison, recall that in sequential download without gossiping, the source node needs to transmit k blocks to the other $n-1$ nodes one after another, and in this scenario, the last node will finish download at time $(n-1)F/W$. The broadcast finish time reduction from $(n-1)F/W$ to F/W seems to be surprising.

Nevertheless, the above argument is incorrect in practice—if the computation time to encode and decode blocks is considered, it turns out that an infinite k will lead to infinite computation time so that no node can ever finish downloading. To verify this, I wrote a C program that can perform RLNC operations (with field size $q = 2^8$) on randomly generated real data blocks of any predefined size B. I analyze how computation overhead of RLNC can affect dissemination. I run the program to simulate the broadcast session for a given set of n, k and block size B on a single machine with a 2.6 GHz Intel Core i7 processor. The *progressive decoding* is applied, that is, whenever a node receives a new (coded) block, it will perform Gaussian elimination for this block together with all previously received blocks. The elapsed time of the entire program is recorded until broadcast finishes. Since no real network transfer happened, we can divide the total elapsed time by the number of nodes n to estimate how much time on average each node will spend on computation in a parallel cluster of n nodes.

The total computation time (including both encoding and decoding) of each node for the entire broadcast session is plotted in Fig. 3.1. I observe that the total computation time per node *1)* is linear to the block size B, and *2)* is a convex increasing function of the number of blocks k. Therefore, for a fixed n and field size q on a given processor, we can do the profiling in Fig. 3.1 and approximately model the computation time per node for the entire broadcast session as

$$T_{\mathsf{Compute}} = \beta B k^{\alpha} = \beta F k^{\alpha-1},$$

where $F = Bk$, the positive constants α, β depend on n, q and the processor, and we have $\alpha \geq 1$ due to the convexity in k as in Fig. 3.1(b).

Therefore, there is a tradeoff between throughput efficiency and coding complexity as k varies. A larger k will hamper the broadcast due to increased coding latency. On the other hand, a larger k can benefit the throughput since the overhead $\lceil \log_2 n \rceil + C$ in broadcast finish time $k + \lceil \log_2 n \rceil + C$ (rounds) becomes neglegible. The question is—how large should k be to achieve the lowest real broadcast finish time in seconds?

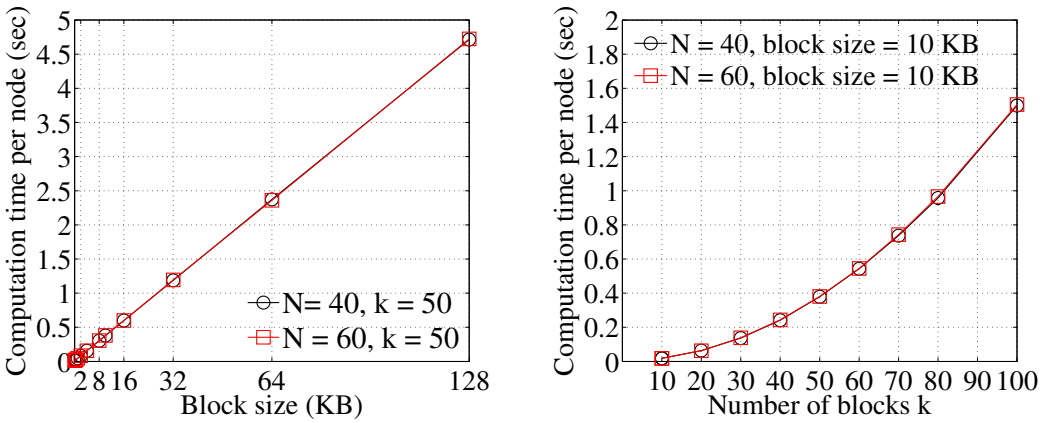

FIGURE 3.1: Mean encode + decode time per node in the entire broadcast session.

To hide the coding latency, I propose to pipeline transmission and computation: in each round, we let each node do the following 3 steps simultaneously as the resources they mainly use are different:

- receiving a new coded block;
- decoding the block received in the previous round (progressively) and encoding a new block;
- sending out a new block encoded in the previous round.

Since both transfer and computation times vary as k, B vary, we need to carefully chunk the file to leverage the throughput benefit of RLNC while hiding its coding latency. If the time spent on decoding and encoding per round by each node is smaller than the time to transfer (send or receive) a block, coding operations will not become the bottleneck of the pipelined process at each node, and the length of each round is still $T = B/W$. However, if the time spent on coding per round is greater than the time to transfer a block at a node, coding will become the bottleneck, and the length of each round will be the computation time for progressively decoding and encoding a block.

Due to the above arguments, the broadcast finish time in seconds in a pipelined implementation can be estimated by

$$T_{\mathsf{Broadcast}} = \max\left\{\left(1 + \frac{\lceil \log_2 n \rceil + C}{k}\right)\frac{F}{W},\ \beta F k^{\alpha-1}\right\}, \tag{3.1}$$

depending on whether transmission or computation constitutes the bottleneck. Clearly, *the lowest broadcast finish time is achieved by the k^* such that computation time equals to transfer time*, i.e., $T_{\mathsf{Transfer}}(k^*) = T_{\mathsf{Compute}}(k^*)$.

The reason is that if we have a $k > k^*$, then $T_{\mathsf{Compute}}(k)$ increases and $T_{\mathsf{Transfer}}(k)$ decreases, and thus $T_{\mathsf{Broadcast}}(k) = T_{\mathsf{Compute}}(k) > T_{\mathsf{Compute}}(k^*)$. If we have a $k < k^*$, then $T_{\mathsf{Transfer}}(k)$ increases and $T_{\mathsf{Compute}}(k)$ decreases, and thus $T_{\mathsf{Broadcast}}(k) = T_{\mathsf{Transfer}}(k) > T_{\mathsf{Transfer}}(k^*)$. Equivalently speaking, the lowest broadcast finish time is achieved by the k^* such that computation time roughly equals to transfer time *per round* at each node.

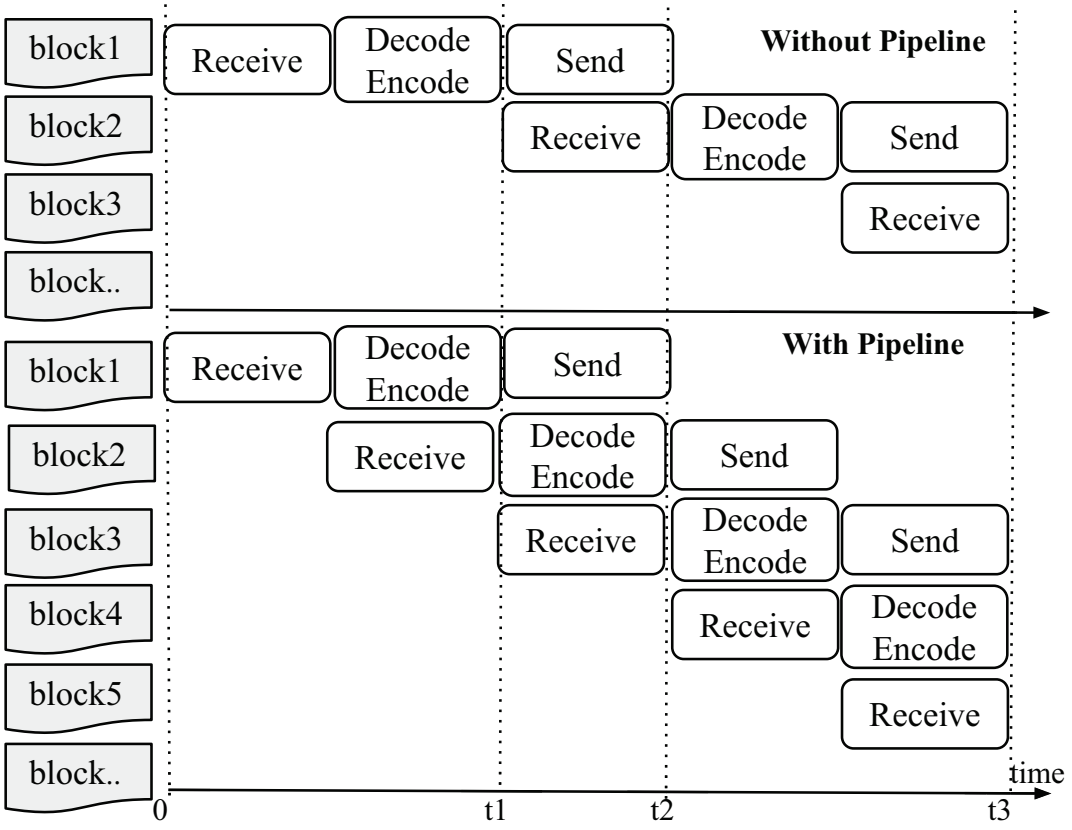

FIGURE 3.2: Processes of transmission and coding with and without pipeline at best case on each task node.

Suppose that the computation time equals to the transfer time (sending or receiving time) per round. As illustrated in Fig. 3.2 for a particular node, with pipelining at t_1, the following 3 processes can happen simultaneously: sending block 1, encoding block 2 (after decoding a previously received block) and receiving block 3, which means we can hide the encoding latency of block 2. In contrast, without pipelining, only sending block 1 and receiving block 2 happen together, and we need extra time to encode each block. In Fig. 3.2, without pipelining, only 3 blocks have been received by this node at time

t_3, whereas with pipelining, 5 blocks have been received at t_3. Note that in the actual implementation to be described in Chapter 5, pipelining is achieved *asynchronously* only by fine-tuning k to equalize independent transfer and computation processes instead of through a time-sliced fashion or synchronized barrier controls. Therefore, Fig. 3.2 is only a conceptual illustration, while in reality, computation and transfer processes may not exactly align on the timeline.

Finally, it is worth noting that the file size F does not affect the value of k^*, as shown from (3.1). The optimal k^* is determined by the bandwidth, the processor, number of nodes n and the field size q for coding, no matter what file is to be broadcasted. In fact, it is not hard to check that the critical k^* equalizing computation and transmission time increases as network bandwidth decreases and as computing power increases. For this reason, *disseminating an extremely large file in k blocks is the same as dividing the file into several smaller batches, each comprised of k blocks, and disseminating the batches sequentially.*

Chapter 4

Multi-Batch Scheduling

Multi-batch scheduling algorithms are important to study due to two reasons. *First*, for a single file, since there is a limit k^* on the number of blocks it should be divided into to avoid excessive computational time, we could divide it into multiple smaller batches, each consisting of k^* blocks, which are transferred one after another. As has been mentioned at the end of Chapter 3, if such batches are transferred sequentially, the total broadcast finish time would be the same as transferring the large file in k^* blocks directly. However, the more fine-grained batch-based approach will better utilize the pipeline: when the first batch is close to finish and few nodes can receive innovative blocks in this batch, the second batch can start to better utilize the resources. *Second*, when different files arrive at the source one after another, we need a scheduling algorithm to make sure the files are delivered to task nodes at the highest throughput. In both of the above scenarios, the key is to devise a scheduling algorithm to decide the blocks of which batch (or file) enjoys the higher priority in transmission at a given point.

4.1 Early Batch First

Two batches of blocks with random broadcast finish times of F_1 and F_2 rounds are firstly considered, respectively. If the two batches are broadcast *sequentially*, the overall expected broadcast finish time will be

$$\mathbf{E}[F_1 + F_2] = \mathbf{E}[F_1] + \mathbf{E}[F_2]$$

rounds. The question is — can we reduce this by overlapping the broadcasts of the two batches, while keeping the finish time of the first batch almost the same as before?

A simple overlapping scheme is to allow the transfers of both batch 1 and batch 2, while *giving priority* to blocks of batch 1:

Algorithm 1. **(Overlap-1)** On the random permutation topology formed in each round, each node transmits an encoded block of batch 1 to its target receiver, if the receiver has *not* decoded batch 1 yet. Otherwise, the node transmits an encoded block of batch 2 to the target receiver if it can.

When most nodes in the network have obtained an enough number of batch-1 blocks, they can start disseminating batch-2 blocks immediately without having to wait until the broadcast of batch 1 finishes in the entire network. With Overlap-1, the broadcast finish time of batch 1 will still be F_1, since it is always prioritized, while the broadcast finish time of both batches becomes less than $\mathbf{E}[F_1 + F_2]$. Nevertheless, with Overlap-1, it turns out that the overlapping phase between batch 1 and batch 2 is rather short, which implies the saving coming from overlapping is limited. In fact, it is difficult to characterize the exact saving on broadcast finish time of both batches in Overlap-1.

4.2 Temporarily Prioritize the Next Batch

In the following theorem, however, the possibility of a new overlapping scheme is pointed out, by showing that if we do not always prioritize batch 1, but instead give priority to batch 2 for $\Theta(\log n)$ rounds before batch 1 finishes broadcasting, we can achieve a saving on the total broadcast finish time by an order of $\Theta(\log n)$ for two batches.

Theorem 4.1. *Let* $m = \lfloor \frac{\log n}{2} \rfloor$. *Suppose that the source starts broadcasting batch 1 and batch 2, respectively, in round 1 and round* $F_1 - m + 1$ *(i.e., m rounds before the finish time of the first batch). Suppose that from round* $F_1 - m + 1$ *to* F_1*, called the* overlapping phase*, the priority is given to batch 2, that is if a node has blocks from both batches, it will use its outgoing link for the second batch. From round* $F_1 + 1$ *onward, the priority is given back to batch 1.*

Then, the expected broadcast finish time of both batches will be at most $\mathbf{E}[F_1] + \mathbf{E}[F_2] - m + \frac{2.7}{1-\frac{2}{\sqrt{n}}}$. *In addition, the expected broadcast finish time of batch 1 is at most* $\mathbf{E}[F_1] + \frac{2.7}{1-\frac{2}{\sqrt{n}}}$.

The proof of the above theorem can be found [16]. Admittedly, the overlapping scheme in Theorem 4.1 is only theoretical because it is hard to know the finish time of batch 1 and thus the start of the overlapping phase exactly, as F_1 is a random variable. However, Theorem 4.1 implies that by giving priority to batch 2 for a few rounds during the overlapping phase, we can save the total broadcast finish time by an order of $\Theta(\log n)$ while delaying the finish time of batch 1 by at most 2.7 rounds when n is big enough.

If there are more than two batches, we can repeat the overlapping process explained in the statement of Theorem 4.1. For $b \geq 3$ batches, let the transfer of batch b start $m = \lfloor \frac{\log n}{2} \rfloor$ rounds before the finish time of batch $b-1$, or right after the finish time of batch $b-2$ (although a rare case in practice), whichever comes last. Again, as before, batch b will have priority in the first m rounds after the start of its transmission, and then the the priority is given back to batch $b-1$ until batch $b-1$ finishes. Then, the following corollary is a straightforward result derived from Theorem 4.1, which shows that the saving for b batches is at least $\Theta(b \log n)$. A proof sketch of this corollary can be found in [16].

Corollary 4.2. *Suppose there are $b \geq 2$ batches. The expected saving in the overall finish time of b batches using the overlapping scheme explained above over the sequential broadcasts of these b batches one after another is at least*

$$(b-1)\left(\left\lfloor \frac{\log n}{2} \right\rfloor - \frac{2.7}{1-\frac{2}{\sqrt{n}}}\right).$$

Moreover, the expected delay in the finish time of batch 1 is at most $\frac{2.7}{1-\frac{2}{\sqrt{n}}}$.

Motivated by the insights offered by the above analysis, we may further speedup multi-batch transfers if we allow the blocks of the next batch to be transferred for $\Theta(\log_2 n)$ rounds with higher priority before the current batch finishes completely. Based on this observation, this thesis proposes another practical multi-batch scheduling scheme called Overlap-2:

Algorithm 2. **(Overlap-2)** Suppose batch 1 and batch 2 have $k1$ and $k2$ blocks, respectively. On the random permutation topology formed in each round, each node transmits an encoded block of batch 1 in the first $k_1 + C_1$ rounds, where C_1 is a small integer constant. In the next $\lceil \log_2 n \rceil$ rounds, batch 2 will have a higher priority, that is, batch 1 blocks should not be transmitted unless the node cannot transmit a block of batch 2

or its target receiver has already decoded batch 2. From round $k_1 + C_1 + \lceil \log_2 n \rceil + 1$ onward, the priority is given back to batch 1 until it finishes. Then, the transfer of batch 3 starts in round $k_1 + C_1 + \lceil \log_2 n \rceil + k_2 + C_2 + 1$ and the above process is repeated for batch 2 and batch 3.

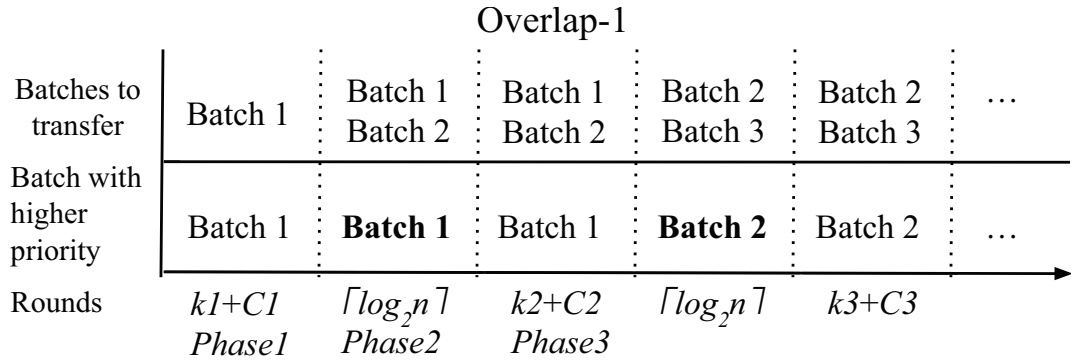

FIGURE 4.1: Priority-based multi-batch scheduling algorithm: Overlap-1.

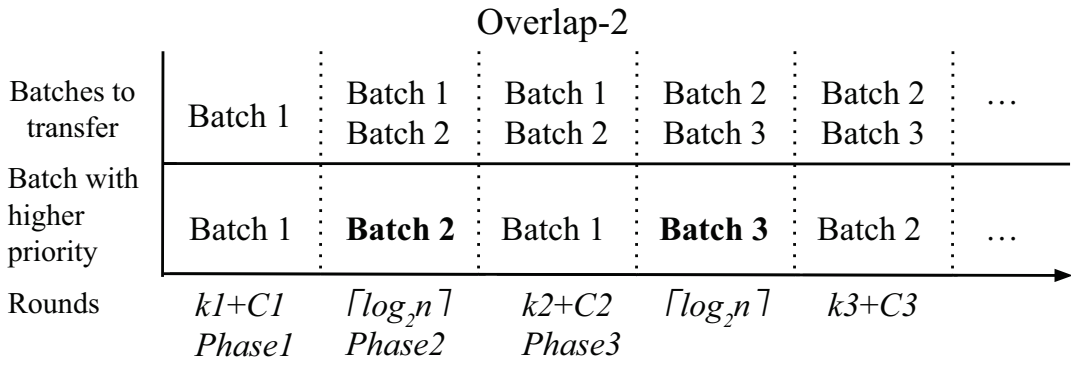

FIGURE 4.2: Priority-based multi-batch scheduling algorithm: Overlap-2.

Counterintuitively, we will show through experiments in Chapter 6 that Overlap-2 can beat Overlap-1 in terms of total broadcast finish time by squeezing in some batch-2 transfers without affecting batch 1 too much. In general, the major difference of the two schemes is what time to give which batch a higher priority for transferring.

The first 3 phases in Fig. 4.1 and Fig. 4.2 illustrate a comparison between Overlap-1 and Overlap-2. After transferring $k_1 + C_1$ rounds of batch 1, Overlap-2 gives a higher priority to batch 2 during the next $\lceil \log_2 n \rceil$ rounds. However, since few nodes have received blocks of batch 2 so far, most nodes are still sending out blocks of batch 1 during Phase 2 and the broadcast of batch 1 is generally finishing up. According to the theoretical result [8], the broadcast of batch 1 will almost finish in round $k_1 + \lceil \log_2 n \rceil + C$, if batch 2 did not affect batch 1 much in Phase 2, which means batch 1 almost finishes in the first two

phases. Therefore, in Phase 3 starting from round $k_1 + \lceil \log_2 n \rceil + C_1 + 1$, mainly batch-2 blocks are being transferred. Since we have already transferred batch 2 for $\lceil \log_2 n \rceil$ rounds in Phase 2, the broadcast of batch 2 will almost finish in another $k2+C2$ rounds. Hence, the finish time of both batches should be approximately $k_1 + \lceil \log_2 n \rceil + C + k_2$ rounds.

Consequently, Overlap-2 may outperform Overlap-1, since more blocks of batch 2 are transferred during Phase 2 of $\lceil \log_2 n \rceil$ rounds without affecting the transfers of batch 1 too much. In contrast, with Overlap-1 the cutoff of batch 1 happens at almost the same time for all nodes, making it hard to save rounds from the shorter overlapping phase.

For more than two batches, the comparison between Overlap-1 and Overlap-2 is illustrated in Fig. 4.1 and Fig. 4.2 as well, which outline the batches transferred in the network together with their priority. Note that since the time to finish broadcasting both batch 1 and batch 2 should be at least $k1 + k2 + \lceil \log_2 n \rceil + C$ rounds (achieved when cross-batch encoding is allowed) [7, 8], there is no point to start batch 3 before round $k1 + k2 + \lceil \log_2 n \rceil + C$. The same principle applies to batch $4, 5, \ldots$ and so on.

Chapter 5

Asynchronous Prototype Implementation

I have implemented an efficient *asynchronous* transmission system named *Cooper*, through Apache Thrift and Boost C++ libraries on Linux Systems. There are mainly two kinds of nodes in *Cooper*: source node and task nodes. Source node initially holds all the data blocks possibly grouped in different batches, while task nodes are target nodes to receive all the data. The entire distributed system consists of multiple processes, including send/receive agents, the encoder and decoder, running asynchronously on each node.

As shown in Fig. 5.1, on a high level, *Cooper* employs an *Inter-Transfer Controller*, located either on the source or on a separate centralized controller node, to coordinate the processes on different nodes. There are three components in the *Inter-Transfer Controller*: the *Topology Generator*, the *Status Monitor* and the *Priority Indicator*. Before starting entire transmission process begins, the *Topology Generator* generates a list of random permutations in advance based on which each node will determine its target receiver according to some protocol to be elaborated. The *Status Monitor* monitors the transferring status of each task node: once it detects that a task node has received all the blocks of a batch, the *Status Monitor* will notify other task nodes immediately, so that they will ignore this node when transferring the batch. In addition, for multiple batch transferring, the *Priority Indicator* dynamically updates the priority of the batches, based on a given multi-batch scheduling algorithm.

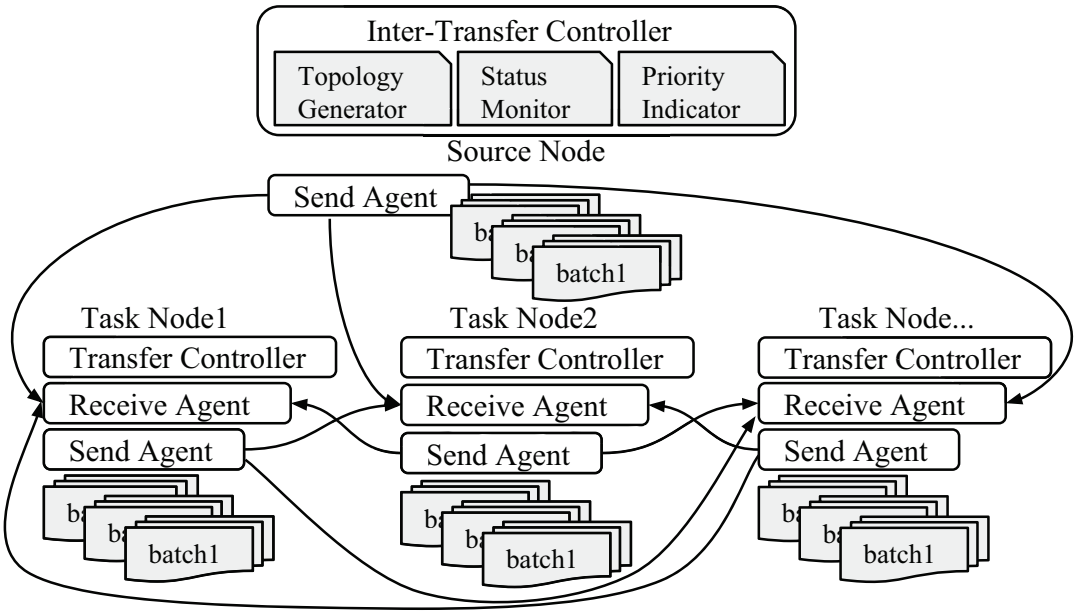

FIGURE 5.1: Architecture of Cooper transmission system.

Each task node has a *Transfer Controller*, that exchanges signals with the *Inter-Transfer Controller* to perform each block transfer. When a node finishes transmitting a block to its receiver, its *Transfer Controller* will read the permutation list generated by the *Topology Generator* on the *Inter-Transfer Controller* and take its downstream neighbour in the next permutation in the list as its designated receiver. Recall that our permutation scheme requires that each node receive from only one other node and send to only one other node at the same time to best utilize the homogeneous bandwidth capacity in the cluster. To make sure each node is receiving blocks from only one sender in this asynchronous system, the *Transfer Controller* will check whether the receiving port of its next designated receiver is occupied. If this designated receiver is busy, the *Transfer Controller* will further check its downstream neighbour in the next permutation in the list until an available node is found. In addition, maintaining only one sender for each receiver can facilitate process pipelining, which will be clear in the later part of this chapter. The transfer controller also receives signals from the *Status Monitor* and *Priority Indicator* to determine which nodes to ignore and which batch has a higher priority.

The *Transfer Controller* launches the *Send Agent* of its own node and *Receive Agent* of its receiver simultaneously to perform a block transfer. The *Send Agent* and the *Receive Agent* are implemented by the Apache Thrift framework based on TCP sockets.

Once the current block transfer finishes, the *Transfer Controller* reports to the *Status Monitor*, and then repeats the above process to transfer the next block.

5.1 Asynchronous Processes

In *Cooper*, I propose to pipeline transmission and computation to hide the coding latency. For this purpose, I launch the following 4 asynchronous processes on each node for each batch:

- **Receive Agent**: receiving a new encoded block;
- **Encoder**: encoding a new block;
- **Send Agent**: sending out a new encoded block;
- **Decoder**: decoding entire received blocks.

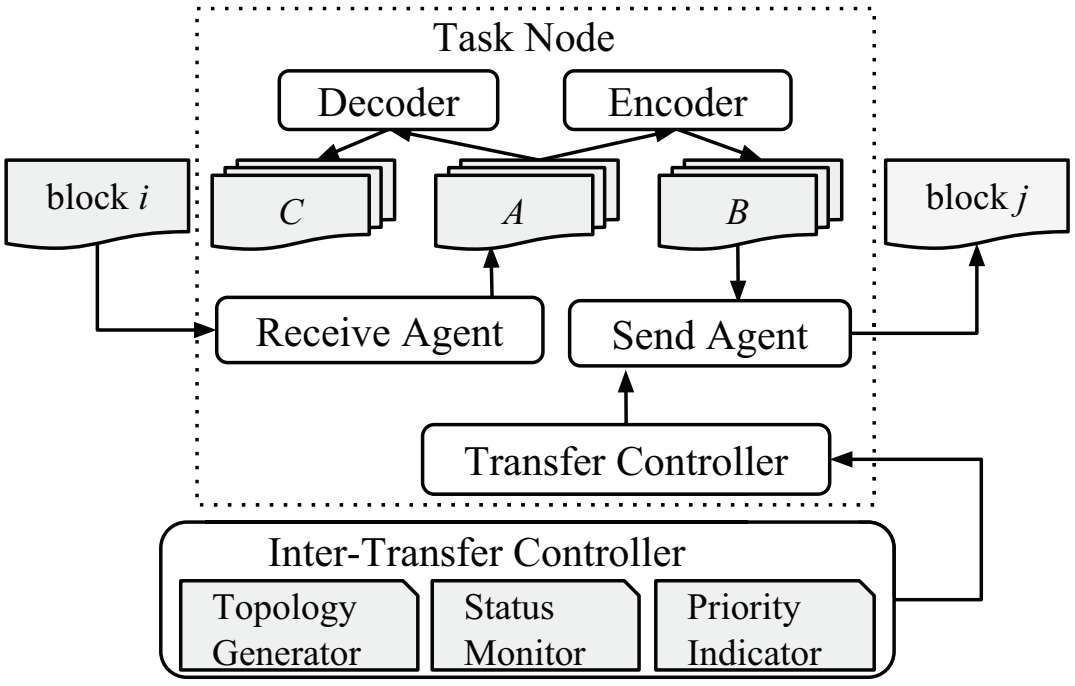

FIGURE 5.2: Asynchronous processes executed on each task node. The Encoder keeps generating encoded blocks, while the Send Agent keeps sending out the newly encoded blocks. The two processes are independent.

Note that on the source node, there is no *Decoder* or *Receive Agent*. All the above 4 processes run individually and asynchronously on each task node and are connected

with each other by reading from or writing to predefined locations on disk, as shown in Fig. 5.2. When the *Receive Agent* receives a new encoded block i, it puts this block into the file (or folder) A on disk. The *Encoder* keeps generating encoded blocks one by one based on all the received blocks in file A and saves each newly encoded block in file (or folder) B. The *Transfer Controller* monitors file B constantly. On the other hand, the *Send Agent* runs independently of the encoder and when a block transfer happens, simply sends out the most recently generated encoded block to its designated receiver determined by the *Inter-Transfer Controller*. Apparently, due to the asynchronous send and encoded processes, it is possible that a node may send out the same encoded block multiple times if the encoding process is slow, or the node might have encoded multiple blocks before a block is sent out, if the send process is slow.

The *Decoder* behaves differently from the pipeline concept described in Chapter 3. For a batch of k blocks, the *Decoder* is launched on a node only if it has received k blocks of that batch in file A. If the decode process fails, decoding will be repeated every time a new encoded block is received and added to file A until success. The reason to launch the decoder only in the end is that when k is properly chosen, the decoding time will be relatively short, as will be shown in Chapter 6, and thus there is no need to perform decoding progressively for each received block.

5.2 Process-Level Parallelism

Now I describe how to achieve the pipelining of transmission and coding on each node in this *asynchronous* distributed system. Specifically, I aim to equalize the running time of the following 3 asynchronous processes on each node (with decoding only happens in the end for all received blocks):

- receiving a new encoded block;
- encoding a new block;
- sending out a new encoded block.

Receiving and sending take roughly the same time in a homogenous network environment. Therefore, I fine tune the block number k only to equalize the time to encode a

block and the time to send one so that process-level parallelism can be roughly achieved on each node. Let us now describe what happens if encoding time and transfer time are not equal in the asynchronous system. If the encoding time is greater transfer time, coding latency will delay the entire transmission process and redundant coded blocks might be sent multiple times. On the other hand, if the transfer time is greater than encoding time, some of the encoded blocks will not be transferred, wasting resources. I will show through the experiments in Chapter 6 that perfect pipelining with equalized encoding and transfer times will indeed achieve the minimum broadcast finish time.

5.3 Inter-Batch Scheduling

I implement both Overlap-1 and Overlap-2, the two multi-batch scheduling algorithms proposed in Chapter 4 which assign different priorities to different batches of data. In an asynchronous transmission system, there is no notion of rounds. Therefore, I define each "round" as the time that the source spends to transmit each block, i.e., round i begins when the source starts to transmit the ith block (of any batch). Specifically, for Overlap-1, an earlier batch is always assigned a higher priority than a later batch. For Overlap-1, as an example, the overlapping phase of batch 1 and batch 2 starts after the source has sent out $k_1 + C_1$ blocks (of any batch) and so on.

In the implementation, I use a Priority Indicator on Intel-Transfer Controller to manage the priority assigned to each batch at a particular point and notify each Transfer Controller during execution. In addition, to avoid excessive encoding workload, I only encode for the "live" batches. For instance, with Overlap-2, after $k_1 + C_1 + \lceil \log_2 n \rceil + k_2 + C_2$ rounds, only batch 2 and batch 3 are alive. In fact, broadcast of batch 1 has already finished and its encoding processes are killed.

5.4 Multi-Encoders on Multi-Cores

Most of commodity desktop computers have multi-core processors nowadays. I further explore the benefit of additional parallelism of launching multiple encoding processes simultaneously. More specifically, I launch two encoding processes, each running a different core in parallel, while the send processes still work sequentially, as show in Fig. 5.3.

Therefore, if I increase k such that the time to transfer a block equals to $T_{\mathsf{encode}}/2$, where T_{encode} is the time to encode a block on each encoder, then during T_{encode} I will have two encoded blocks generated in parallel as well as two encoded blocks (generated during the previous T_{encode}) transmitted sequentially. This way I still achieve perfect pipelining of transfer and coding , as if the coding latency is hided.

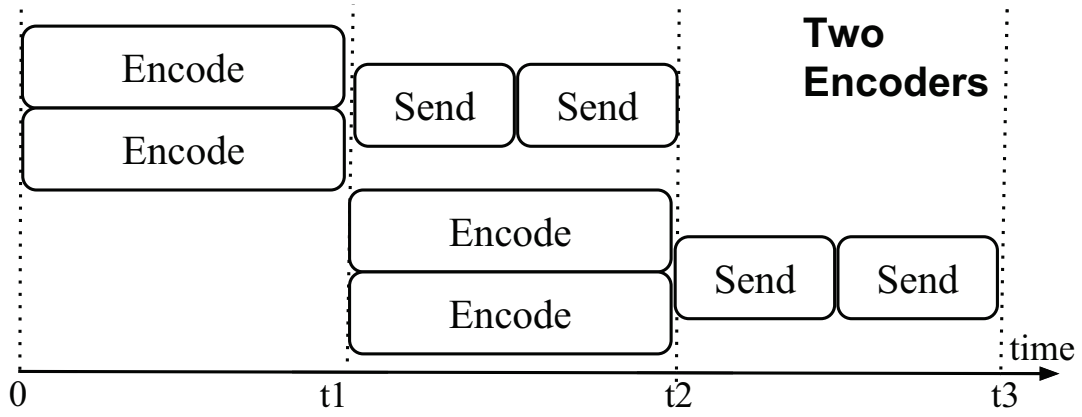

FIGURE 5.3: Processes with two parallel encoders.

However, with two parallel encoders, I end up having a larger k^* which almost doubles the k^* with one encoder (as will be shown in Chapter 6). This means that the overhead part $\lceil \log_n \rceil + C$ in the finish time becomes less significant as compared to the number of blocks k^*, increasing the throughput and further reducing broadcast finish time according to the theory. With the presence of multi-encoders, the encoded blocks are transmitted in a round-robin fashion.

Chapter 6

Performance Evaluation

In this chapter, an extensive performance evaluation of the proposed mechanisms is provided by deploying and measuring the developed asynchronous prototype system *Cooper* on Amazon EC2. I show that *Cooper* significantly outperforms state-of-the-art content distribution tools such as BitTorrent in terms of broadcast transfer speed. To ensure fair comparison in a controlled network environment, I set the upload capacity and download capacity at each node to be either all 10 Mbps or all 15 Mbps. Besides, the computation capability varies between different groups of experiments due to the workload change. However, I attempt to maintain the same experimental environment for all experiments in the same.

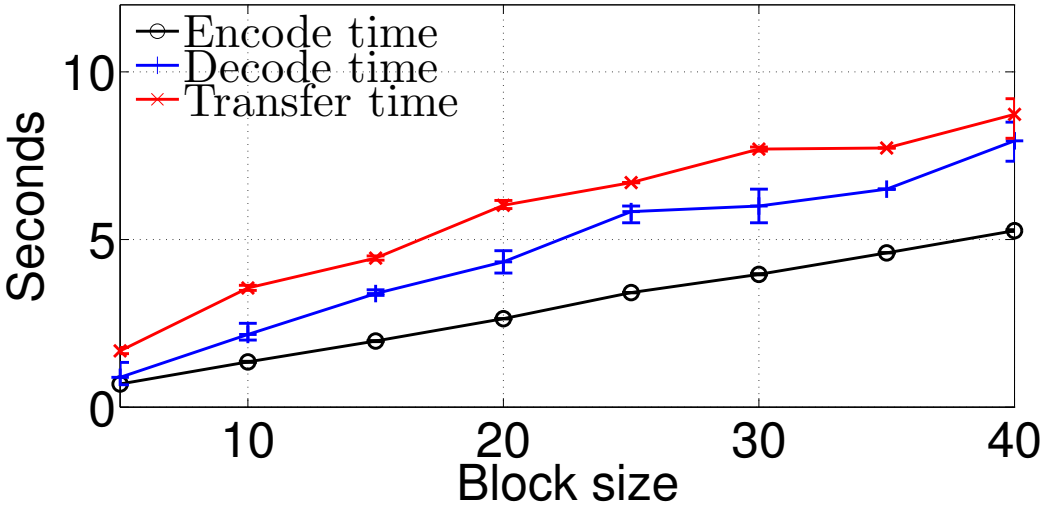

FIGURE 6.1: Average computation and transferring time per node with error bars, as block size varies.

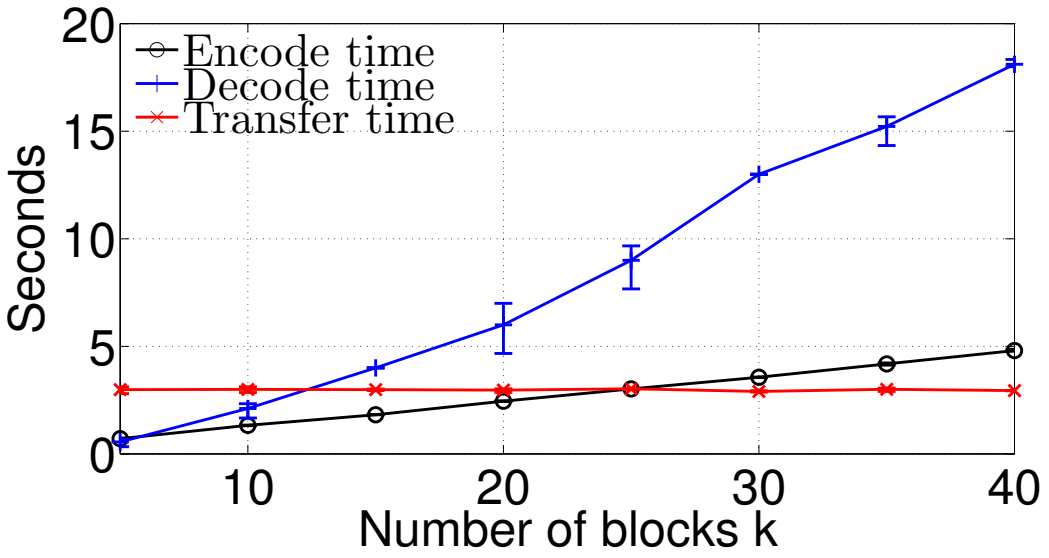

FIGURE 6.2: Average computation and transferring time per node with error bars, as block number k varies.

6.1 Evaluation for Pipelining

I first verify the effectiveness of pipelining via small-scale experiments on 4 desktop computers located in a campus environment, each with quad-core 2.6 GHz Intel Core i7. The operating system is Linux Ubuntu 12.04.

Verifying the computation time model. From simulations in Fig. 3.1), I have observed the computation time per node (with progressive decoding) $T_{\mathsf{compute}} = \beta B k^{\alpha}$, for $\alpha \geq 1$. Now I verify that T_{encode} alone (without progressive decoding as in the implementation) follows the same trend. Fig. 6.1 plots the average encoding and transferring time per block (with error bars) per node, as well as the *total* decoding time per node in the end, as the block size B varies, when the number of blocks k is fixed. A clear linearly increasing trend is observed for encoding time per block with almost constant slope.

Furthermore, I observe that the total decoding time in the end is approximately linear to the block size and is not large, justifying the reason why I only pipeline encoding alone and transfer in implementation. Similarly, Fig. 6.2 plots the same variables as the number of blocks k varies, when the block size is fixed as 10 MB, which shows that encoding time per block is convexly increasing as k increases, verifying the simulation results on computation time.

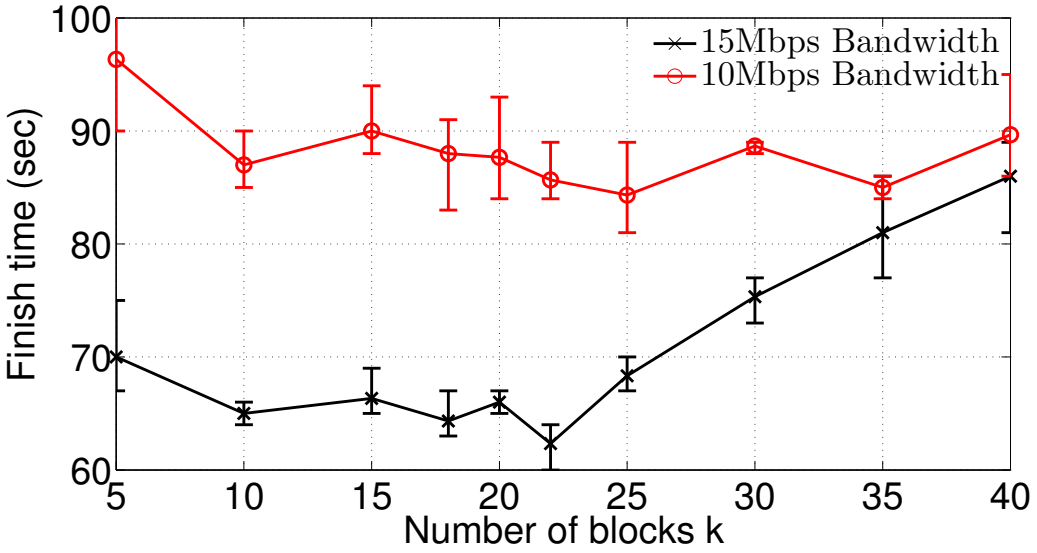

FIGURE 6.3: The finish time under different k with error bars, when the file size is fixed to 100 MB.

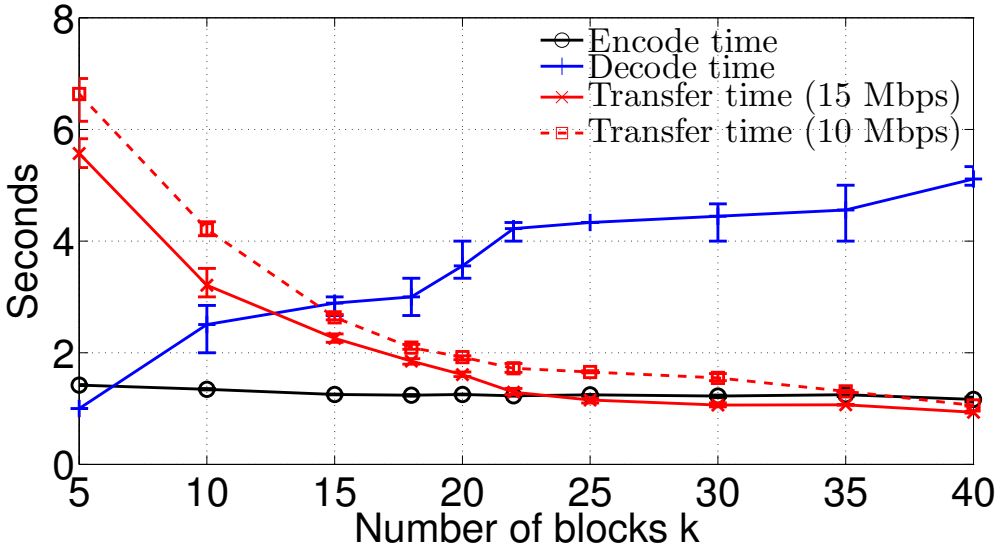

FIGURE 6.4: Average encoding and transferring time per node per block with error bars, when the file size is fixed to 100 MB.

Verifying the Optimality of Pipelining. In Chapter 3, the lowest broadcast finish time is achieved by the k^* such that computation time equals to transferring time has been argued. In practical implementation of *Cooper*, I only need to find the k^* that equalizes the encoding time and transferring time per block. I evaluate the times to broadcast a file of a fixe size (100 MB) with *Cooper*, as I chunk the file into different numbers of blocks k, as shown in Fig. 6.3. As we can see, if the network bandwidth is 15 Mbps, the lowest broadcast time is obtained at $k* = 22$, while for 10 Mbps network bandwidth, the lowest broadcast finish time is achieved at $k* = 35$. Moreover, Fig. 6.4 shows that with 15 Mbps bandwidth, the transferring time is roughly equal to the encoding time per block when k is 22, and for 10 Mbps the two quantities are roughly the same when k is 35. Therefore, I verify that the lowest broadcast finish time is achieved when the encoding time per block equals to the transferring time per block in the implementation.

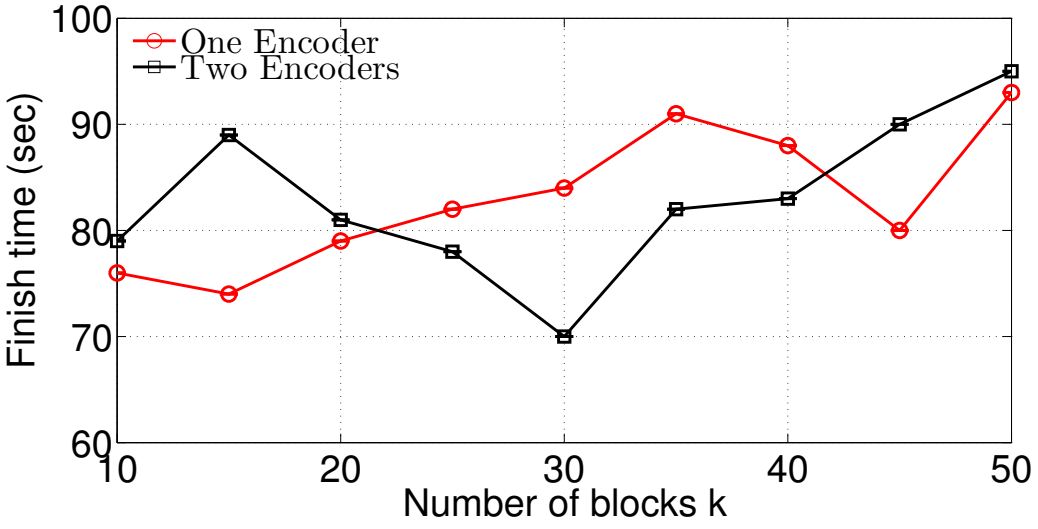

FIGURE 6.5: The finish time when setting different number of blocks with error bars. (Network bandwidth: 15 Mbps)

Pipelining with Double Encoders. I further evaluate benefit of the pipelining mechanism with two parallel encoders. It is worth noting that the computation time for this set of tests becomes longer than in previous experiments, due to the unpredictable changes in workload on the campus desktops. As a result, I obtain different computation time measurements from the previous figures.

As shown in Fig. 6.5, the lowest broadcast finish time of one encoder is 75 seconds, achieved when $k = 15$, while for two encoders, the best broadcast time is 70 seconds,

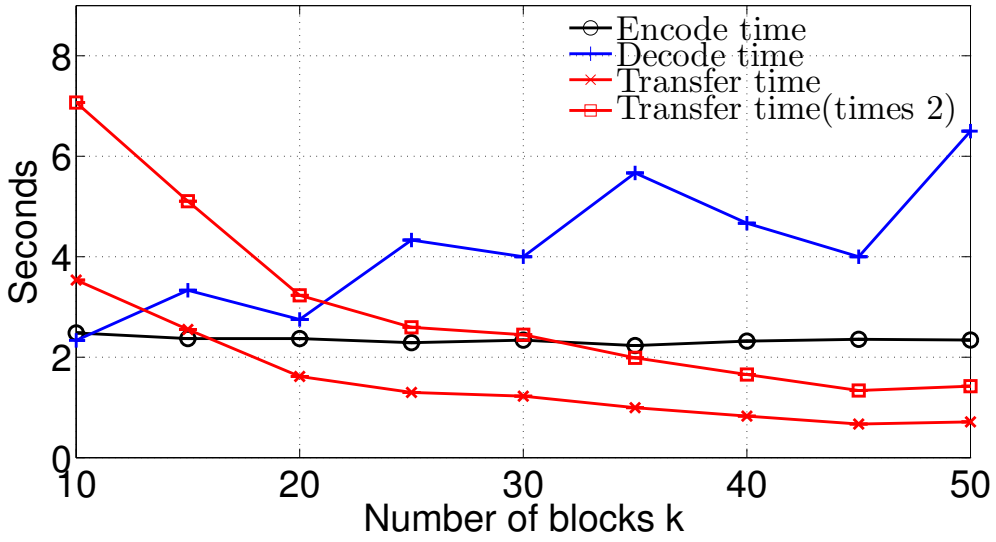

FIGURE 6.6: Average computation and transferring time per node per round with error bars, when file size is fixed as 100MB. (Network bandwidth: 15 Mbps)

achieved when $k = 30$. By further checking Fig. 6.6, I find that when $k = 15$ equalizes the encoding time and transferring time, whereas when $k = 30$, the average encoding time is same as twice the average transferring time. Thus, I have verified that two parallel encoders indeed can enhance the performance of *Cooper*, while the right k in this case roughly doubles that of using one encoder.

6.2 Comparisons with Other Systems

In this part, the performance of *Cooper* is compared with BitTorrent and a baseline rand block negotiation scheme on Amazon EC2, when *Cooper* uses one encoder, two encoders and batch-based file transfers. I launched 20 t2.medium instances from Amazon EC2 (South American region), each with 2 cores, 4 GB memory and low to moderate network performance running Linux Ubuntu 14.04. The same network environments is maintained for all experiments in each group. The file size to be broadcast is fixed to 100 MB for all systems. The network bandwidth is around 15 Mbps. I first describe the baseline systems to be compared against as well as the parameter settings in *Cooper*:

A Random Block Scheme (RB): a scheme adopting the same (asynchronous) implementation of *Cooper* except that it does not transmit coded blocks. Instead, for each node, I maintain a buffer map of received blocks, keep updating this buffer map to all

other nodes during running time once in a while. For the transmission of each block, the sender will send a random block that its target receiver does not have. If there is no such a block, the sender will proceed to check its target receiver in the next permutation generated by the topology generator. After receiving each new block, the receiver updates its buffer map and synchronize the buffer map to others periodically.

To achieve the best performance of RB, the buffer update frequency is optimized. The reason is that, if we synchronize the buffer map too often, there will be signalling overhead that downgrades the performance; if we synchronize less often, there will be redundantly transferred blocks, due to outdated buffer maps which hurts the finish time too. The best performance is achieved on Amazon EC2 if each node updates its buffer map to other nodes upon receiving *every other block*. The number of blocks k for RB is further tuned, as shown in Fig. 6.7, from which I observe that the best performance of RB is achieved when $k = 100$. Note that the optimal k for random block is much larger than in *Cooper*, because there is no coding operation.

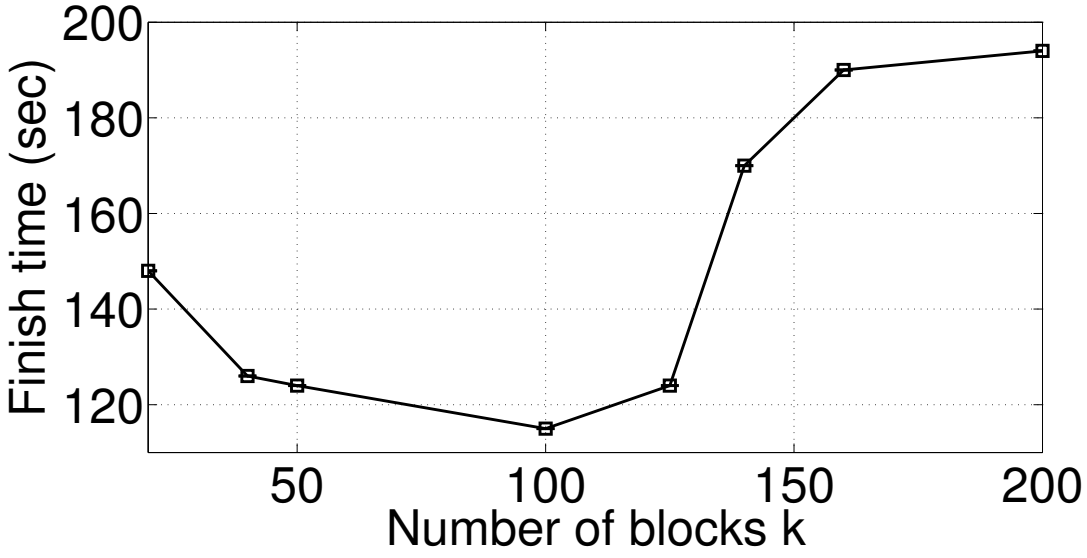

FIGURE 6.7: The finish time of RB on 20 t2.medium instances of Amazon EC2, with file size is fixed as 100 MB and k varies.

BitTorrent (BT): I deploy *rTorrent*[17], which is a command-line-based BitTorrent client written in C++ with high performance, based on the libTorrent libraries for Unix. The performance of *rTorrent* is carefully optimized on EC2 by configuring its various parameters, like number of simultaneous connections.

Tuning *Cooper*. The best broadcast performance has already been verified that it is achieved under perfect pipelining with the right k^* equalizing encoding and transferring time (in the two-encoder case, the encoding time should double the transferring time). Therefore, for *Cooper*, some profiling (that is done lightweight on a single node) is first employed to find the right k^* for both one-encode and two-encoder cases. As illustrated in Fig. 6.8 and Fig. 6.9, I observe that on t2.medium instances of Amazon EC2, k^* for pipelining with one encoder is around 16, while for pipelining with two encoders, k^* is around 30.

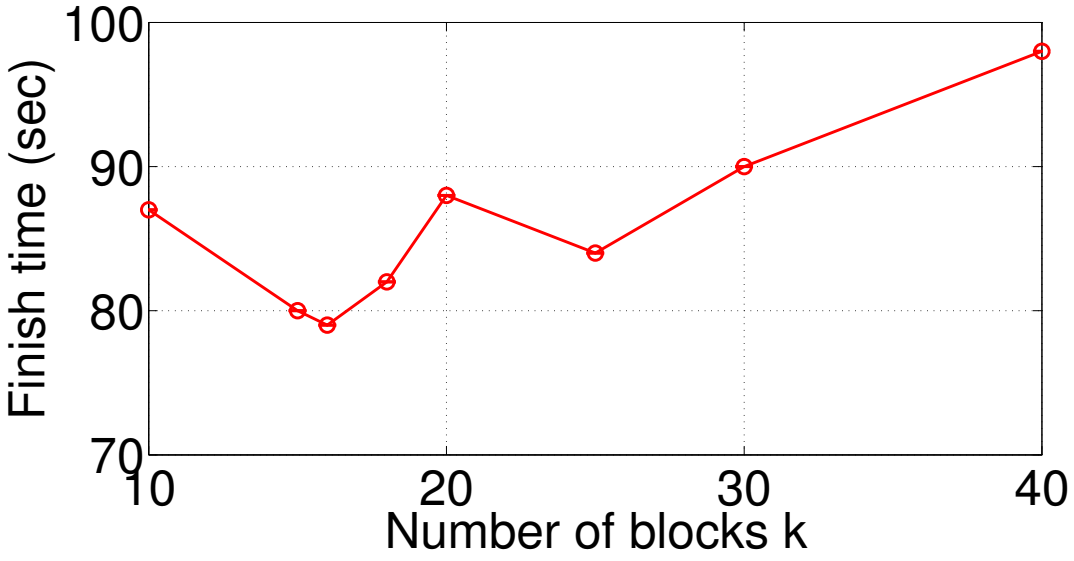

FIGURE 6.8: The finish time of Cooper on 20 t2.medium instances of Amazon EC2, with file size is fixed as 100 MB and k varies.

In Fig. 6.10, the broadcast finish times is compared for a 100 MB file under 5 different schemes: *Cooper* with a single encoder, *Cooper* with double encoders, *Cooper* with batch-based transfers, Random Block (RB) and rTorrent (BT). For *Cooper* with batch-based transfers, I divide the file into 3 batches and apply the inter-batch scheduling algorithm Overlap-2. From Fig. 6.10, we see that the broadcast finish time of 100 MB file for Random Block and BitTorrent is around 115 seconds and 106 seconds respectively, while for *Cooper* with one encoder, the finish time is around only 80 seconds, significantly outperform both Random Block and BitTorrent. Moreover, *Cooper* with double encoders further reduces the broadcast time to around 70 seconds.

The highest performance is achieved by chunking the file into 3 batches, scheduled with Overlap-2, which achieves a finish time of 60 seconds. Therefore, we conclude that,

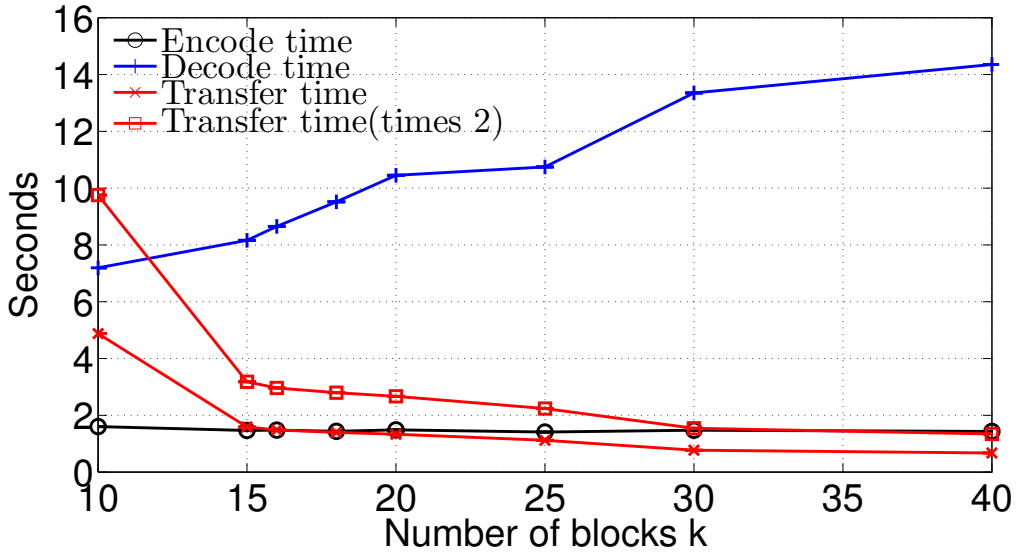

FIGURE 6.9: Average computation and transferring time per node per round on 20 t2.medium instances of Amazon EC2, when file size is fixed as 100MB.

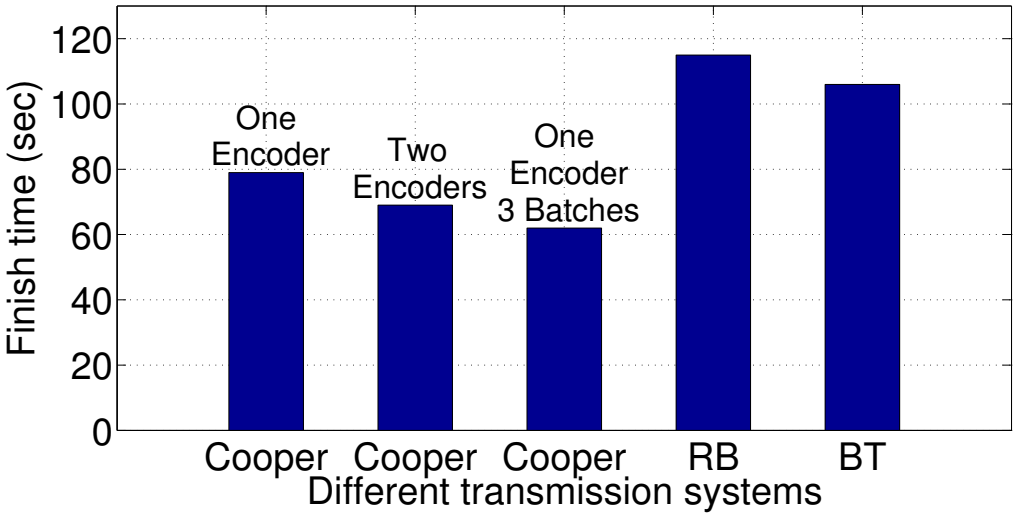

FIGURE 6.10: Comparison of finish time between Cooper and baseline algorithms, with file size fixed as 100MB.

Cooper can significantly speedup data dissemination in computer clusters, especially using a multi-batch-based scheduling algorithm.

6.3 Multi-Batch Scheduling Algorithms.

Fig. 6.10 has shown that inter-batch scheduling can reduce broadcast finish time for multi-batch transfers or when transferring a large file in smaller batches. In this part, the performance of the proposed inter-batch scheduling algorithms Overlap-1 and Overlap-2 is evaluated on Amazon EC2, as well as traditional non-overlapping sequential transfers. The experiment environment is the same as in Section 6.2. The size of *each* batch is 100 MB. And I apply *Cooper* with a single encoder in all experiments here.

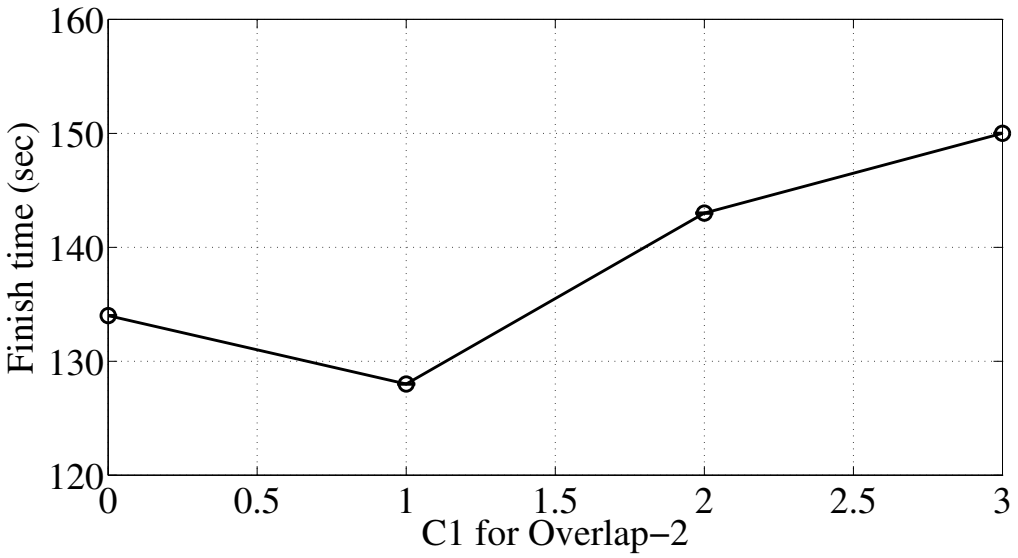

FIGURE 6.11: Finish time of transferring two batches with Overlap-2 on different C1.

The right k on t2.medium instances of Amazon EC2 is around 16. And another set of parameters I need to tune is $C1, C2, \ldots$, the small constants to determine the lengths of non-overlapping phase in Overlap-2. I tried several experiments and find that the best broadcast finish time for Overlap-2 is achieved by setting $C1 = C2 = \ldots = 1$, as shown in Fig. 6.11. To compare the performance of Overlap-1 and Overlap-2, their total broadcast finish times is evaluated when transferring 2 and 3 batches, as compared to sequential transfers of these batches one after another.

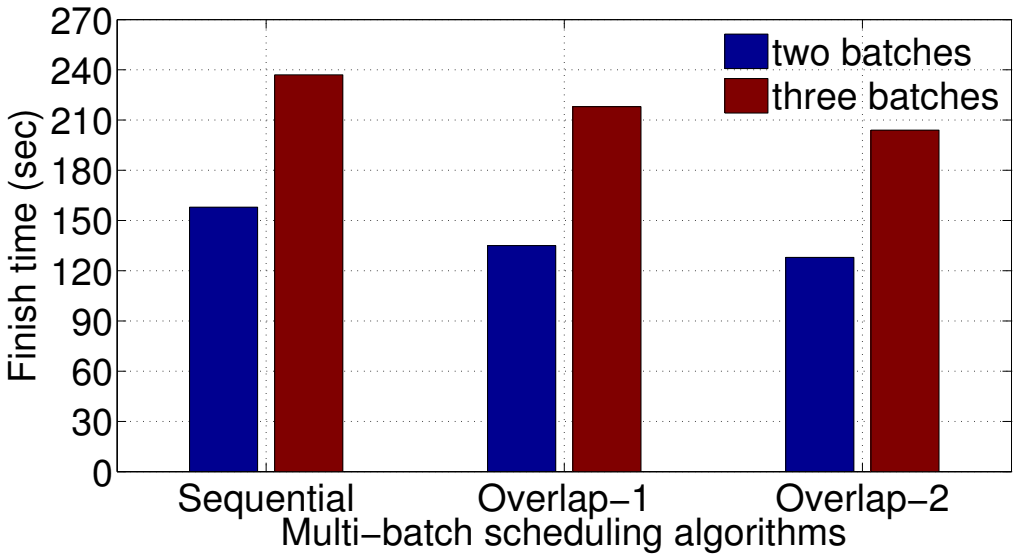

FIGURE 6.12: The finish time of transferring two and three batches with different priority-based algorithms.

As shown in Fig. 6.12, for two files (or batches), the finish time of sequential approach is around $160 = 80 \times 2$ seconds , while with Overlap-1, it is reduced to 135 seconds, which means overlapping does speedup performance. Moreover, the total broadcast finish time of Overlap-2 is only around 128 seconds, which is even better than Overlap-1. For three batches, the differences among the three approaches are more more obvious, especially the gap between Overlap-1 and Overlap-2.

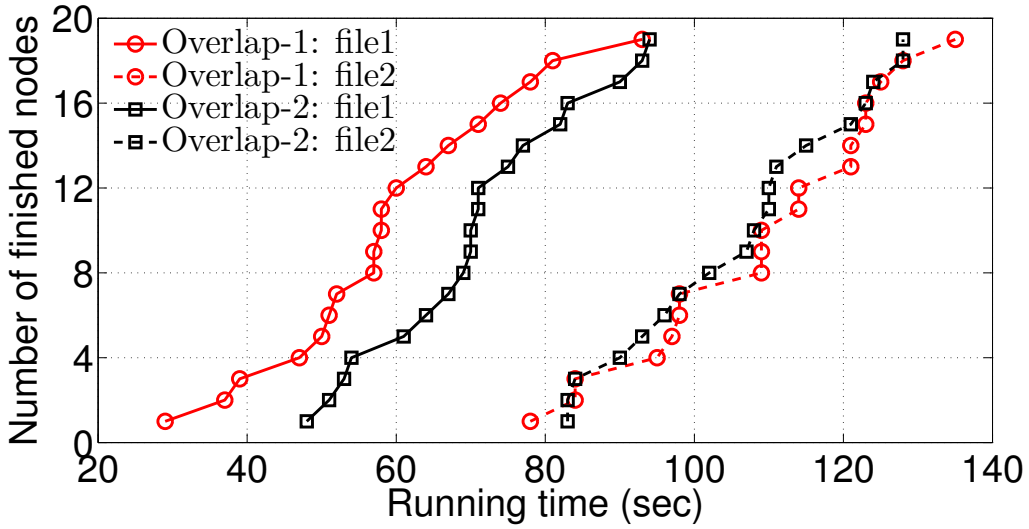

FIGURE 6.13: The number of nodes finished during running time when using different priority-based algorithms to transfer two files.

To understand the underlying mechanisms of overlapping, I collect logging data from one experiment, and illustrate the numbers of finished nodes during running time in Fig. 6.13 for both Overlap-1 and Overlap-2, and illustrate the number of received blocks at every node for Overlap-1 and Overlap-2, respectively, in Fig. 6.14 and Fig. 6.15. The total number of nodes to receive blocks is 19 (except the source). And suppose there are two files (or batches) to be broadcast.

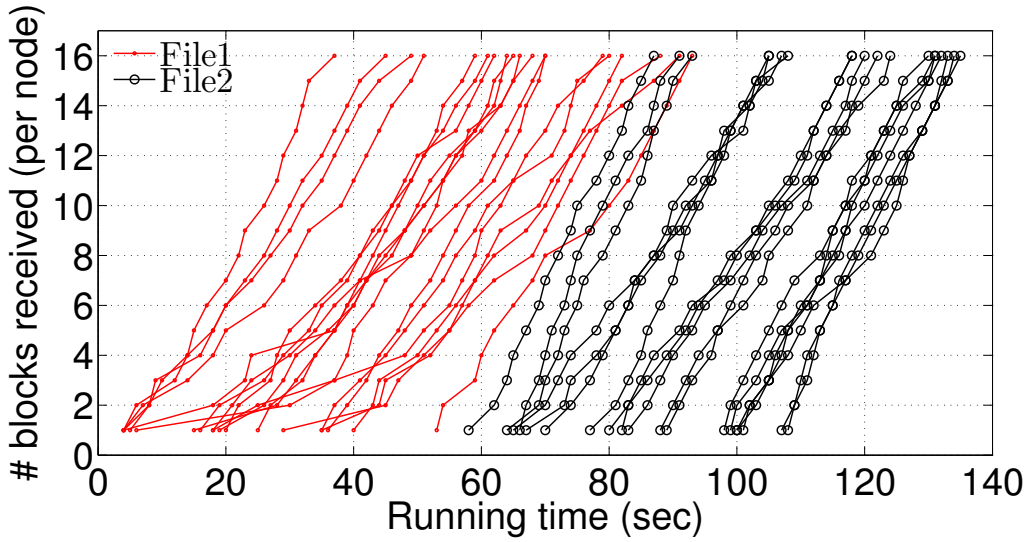

FIGURE 6.14: The number of blocks received (by each node) during running time when using algorithm Overlap-1 to transfer two batches.

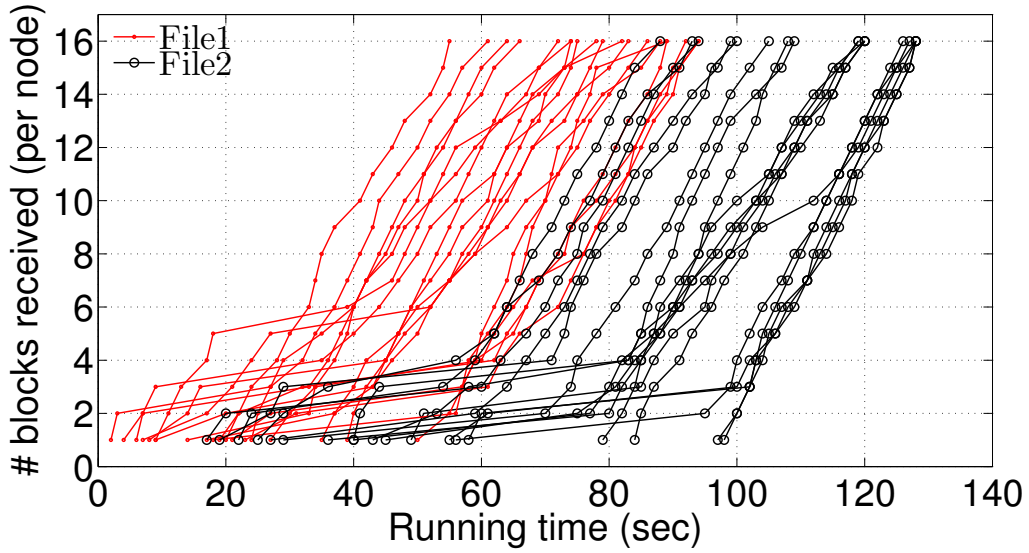

FIGURE 6.15: The number of blocks received (by each node) during running time when using algorithm Overlap-2 to transfer two files.

In Fig. 6.13, the two lines on the left represent the numbers of nodes who have finished file 1, in Overlap-1 and Overlap-2, respectively, while the two lines on the right are for file 2. As we can see, with Overlap-1, it takes around 90 seconds to finish broadcasting file 1 to all receivers and 135 seconds to finish broadcasting file 2. In contrast, it costs Overlap-2 90 seconds as well to finish broadcasting file 1 but only 128 seconds to finish file 2. In terms of the first finished node, Overlap-1 spends 30 seconds whereas Overlap-2 takes around 50 seconds.

The reason underlying the advantage of Overlap-2 is that for Overlap-1, I give priority to file 1 all the time until finishing broadcast of file 1, while for Overlap-2, after Phase 1, priority is granted to file 2 temporarily. In Fig. 6.14 and Fig. 6.15, each line is for each specific node, representing the number of blocks received by this specific node as time progresses. As we can see from the two figures, for each node, Overlap-1 only receives blocks of file 2 after finishing receiving file 1, whereas Overlap-2 squeezes in some file-2 transfers during the early stage when broadcasting file 1. However, by doing this, Overlap-2 actually does not affect the broadcast finish time of file 1 much, since after all, few nodes can transmit file-2 blocks at the beginning. Therefore, the finish time of file 1 of the two algorithms is close. Moreover, since Overlap-2 has already transferred several blocks of file 2 during the early stage, file 2 will be finished faster, reducing the total broadcast time of two files.

Chapter 7

Related Work

Generally, the objective of content dissemination protocols is to minimize the total broadcast finish time, subject to node capacity constraints. And traditional solutions for content dissemination protocols are mostly based on multiple trees. Chiu *et al.* [18] and Li *et al.* [19] show that the maximum feasible broadcast rate limited by the node download capacity constraints can be achieved by packing only depth-1 and depth-2 spanning or Steiner trees. And the maximum data streaming rate is then obtained by assigning rates to the constructed multi-trees. Chen *et al.* [20] further describe a primal-dual algorithm for the distributed implementation of tree construction and rate allocation. However, tree-based algorithms remains largely centralized with non-trivial structure maintenance overhead. Even in distributed implementation, it is not certain if the algorithm can converge with the presence of node dynamics or random failures. The frequently changing dissemination tasks in a server cluster also makes a tree-solution cumbersome to manage.

Gossip algorithms, on the other hand, enjoys the superior adaptability to arbitrary network and topology and the simplicity of implementation. Sanghavi *et al.* [21] show the order-optimality of non-coding gossiping, with the aid of node buffer state exchanges. With node state reconciliation, Massoulie *et al.* [22] prove the strict optimality of non-coding block selection protocols. However, it is theoretically shown in [23] that, a common problem with gossip protocols that require state exchanges is that their success heavily depends on the accuracy and frequency of state updates. As compared to these

works, the use of network coding completely eliminates the need of node buffer exchanges, further reducing the block scheduling overhead that escalates as k grows.

Deb *et al.* [13] show the order-optimality of network coding gossip in a random phone call model, assuming k blocks are spread across the network to start with. Haeupler [14] further proves that a finish time of $k + o(k)$ is achieved in the same model for a random pull protocol when each node holds a subset of k blocks initially. For the case that only one source node wants to broadcast its blocks to all other nodes, optimal pipelining in k is not known yet. Shifting away from the random phone call model, a class of sender-receiver pairing rules satisfying a simple permutation condition is adopted which can approach the optimal broadcast time of $k + \lceil \log_2 N \rceil$ regardless of the initial states of nodes. Such a permutation is easily implemented in any controllable server clusters. Finally, different from previous works [13, 18–22, 24, 25] that assume unbounded node download capacities, both node upload and download bandwidth constraints are considered which may both be present in reality.

In computing or server clusters, multi-node transfer operations have a significant impact on the performance of cluster applications, Orchestra [1] is proposed as an architecture that enables global control both across and within different transfers to optimize performance. For data broadcasts, Orchestra uses a BitTorrent-like scheme called *Cornet* to speed up data dissemination. [26] proposes Torchestra to reduce delays by separating interactive and bulk traffic onto two different TCP connections between each pair of nodes. A general deployment advisor named ClouDiA is proposed in [27] which selects application node deployments minimizing either the largest latency between application nodes, or the longest critical path among all application nodes. Natjam system proposed in [28] supports arbitrary job priorities, hard real-time scheduling, and efficient preemption for Mapreduce clusters that are resource-constrained. In [29], they present a system that reduces the skew impact by transparently predicting data communication volume at runtime and mapping the many end-to-end flows among the various processes to the underlying network, using emerging software-defined networking technologies to avoid hotspots in the network. And in [30], they propose a new dynamic network optimizer called OFScheduler for heterogeneous clusters to relieve the network traffic during the execution of MapReduce jobs, which focuses on reducing bandwith competition, balancing the workload of network links and increasing bandwidth utilization. In this thesis,

we show that the proposed coded permutation gossip with smart inter-batch scheduling can further save broadcast time by nearly 40% over BitTorrent-like protocols.

Chapter 8

Conclusion

In this thesis, an asynchronous and distributed prototype system is presented, named *Cooper*, to speedup data dissemination in computer clusters, in which the "permutation gossip" is leveraged to optimally utilize network bandwidth. More specifically, all nodes into time-varied random permutation topologies are arranged, on which each node transmits coded blocks using RLNC.

To boost up performance in practice, multiple inventions are incorporated in the design of *Cooper*. First, a pipelining technique is proposed to perform coding operations and data transferring in parallel on the process level, to hide coding latency when chunking the file into some fine-tuned number of blocks. Generally speaking, I aim to equalize the running time of the following asynchronous processes on each node: 1) receiving a new encoded block; 2) encoding a new block; 3) sending out a new encoded block. Second, an asynchronous transmission control model is adopted to convert the time-slotted theoretical model into real implementation. Third, for multi-batch data transfer, two priority-based scheduling algorithms are proposed to overlap the transmission of consecutive batches and reduce the total broadcast finish time: Overlap-1 and Overlap-2. The main difference of the two strategies is to decide the blocks of which batch/file enjoys the higher priority in transmission at a given point. Fourth, the feasibility of using multi-core processors is explored on a fine-tuned asynchronous coding-transfer pipeline to achieve better performance.

Cooper is deployed on Amazon EC2 and perform extensive real experiments to verify the proposed theoretically inspired algorithms, as compared to state-to-the art content

distribution tools, including BitTorrent and an optimized random block negotiation scheme. Based on results from real experiments, I find that *Cooper* (with a single encoder) can significantly reduce the time to broadcast a file by 25% over BitTorrent and 30% over the random-block scheme. Moreover, *Cooper* with parallel double encoders further increase the performance gap to 34% and 40%. Furthermore, with batch-based transfers and the proposed multi-batch scheduling, the savings reach 44% and 48%.

In addition, the performance of the proposed inter-batch scheduling algorithms Overlap-1 and Overlap-2 is evaluated on Amazon EC2, as well as traditional sequential transfers, with applying *Cooper* with a single encoder. From experiment results, we get that multi-batch-based algorithms significant reduce the total broadcast finish time, and Overlap-2 performs better than Overlap-1. Also, the underlying mechanisms of overlapping is further explored, by illustrating the numbers of finished nodes and the number of finished blocks at each node for Overlap-1 and Overlap-2 during running time.

Bibliography

[1] Mosharaf Chowdhuryand, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *Proc. of ACM SIGCOMM*, 2011.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. of the 2nd USENIX Conference on Hot topics in Cloud Computing*, 2010.

[3] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of EuroSys*, 2007.

[4] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Proc. of AAIM*, pages 337–348. Springer-Verlag, 2008.

[5] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time url spam filtering service. In *Proc. of IEEE Symposium on Security and Privacy*, 2011.

[6] Tracey Ho, Ralf Koetter, Muriel Medard, David R. Karger, and Michelle Effros. The Benefits of Coding over Routing in a Randomized Setting. In *Proc. IEEE Int'l Symp. Information Theory (ISIT)*, 2003.

[7] A. Bar-Noy and S. Kipnis. Broadcasting Multiple Messages in Simultaneous Send/Receive Systems. *Discrete Applied Mathematics*, 55:95–105, 1994.

[8] Majid Khabbazian and Di Niu. Achieving Optimal Block Pipelining in Organized Network Coded Gossip . In *Proc. of ICDCS*, 2014.

[9] *Apache Thrift: open-source cloud computing*, https://thrift.apache.org/.

[10] *Boost: a free peer-reviewed portable C++ source libraries*, http://www.boost.org/.

[11] *Amazon EC2: Amazon Elastic Compute Cloud*, http://aws.amazon.com/ec2/.

[12] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. The Bittorrent P2P File-Sharing System: Measurements and Analysis. In *The 4th International Workshop on Peer-to-Peer Systems (IPTPS'05)*, Ithaca, New York, February 2005.

[13] S. Deb, M. Médard, and C. Choute. Algebraic Gossip: A Network Coding Approach to Optimal Multiple Rumor Mongering. *IEEE Trans. Inform. Theory*, 52(6):2486–2507, June 2006.

[14] Bernhard Haeupler. Analyzing network coding gossip made easy. In *Proc. of ACM STOC*, 2011.

[15] Di Niu and Baochun Li. Asymptotic optimality of randomized peer-to-peer broadcast with network coding. In *Proc. of IEEE INFOCOM*, 2011.

[16] Yan Liu, Majid Khabbazian, and Di Niu. Cooper: Speedup batch data dissemination in computer clusters with coded permutation gossips. under submission.

[17] *rTorrent: open-source cloud computing*, http://rakshasa.github.io/rtorrent/.

[18] D. M. Chiu, R. W. Yeung, J. Huang, and B. Fan. Can Network Coding Help in P2P Networks? In *Proc. International Workshop on Network Coding*, Boston, April 2006.

[19] J. Li, P. A. Chou, and C. Zhang. Mutualcast: An Efficient Mechanism for Content Distribution in a Peer-to-Peer (P2P) Network. In *Proc. of ACM SIGCOMM Asia Workshop*, Beijing, China, April 2005.

[20] M. Chen, M. Ponec, S. Sengupta, J. Li, and P. A. Chou. Utility Maximization in Peer-to-Peer Systems. In *Proc. ACM SIGMETRICS*, Annapolis, Maryland, USA, June 2008.

[21] S. Sanghavi, B. Hajek, and L. Massoulie. Gossiping with Multiple Messages. In *Proc. IEEE INFOCOM*, Anchorage, Alaska, 2007.

[22] L. Massoulie, A. Twigg, C. Gkantsidis, and P. Rodriguez. Randomized Decentralized Broadcasting Algorithms. In *Proc. IEEE INFOCOM*, Anchorage, Alaska, USA, May 2007.

[23] Chen Feng, Baochun Li, and Bo Li. Understanding the Performance Gap between Pull-based Mesh Streaming Protocols and Fundamental Limits. In *Proc. IEEE INFOCOM*, Rio de Janeiro, Brazil, April 19-25 2009.

[24] R. Kumar, Y. Liu, and K. Ross. Stochastic Fluid Theory for P2P Streaming Systems. In *Proc. IEEE INFOCOM*, Anchorage, Alaska, USA, 2007.

[25] J. Mundinger, R. Weber, and G. Weiss. Optimal Scheduling of Peer- to-Peer File Dissemination. *Journal of Scheduling*, 2007.

[26] Deepika Gopal and Nadia Heninger. Torchestra: Reducing interactive traffic delays over tor. In *Proceedings of the 2012 ACM Workshop on Privacy in the Electronic Society*, pages 31–42. ACM, 2012.

[27] Tao Zou, Ronan Le Bras, Marcos Vaz Salles, Alan Demers, and Johannes Gehrke. Cloudia: a deployment advisor for public clouds. In *Proceedings of the VLDB Endowment*, volume 6, pages 121–132. VLDB Endowment, 2012.

[28] Brian Cho, Muntasir Rahman, Tej Chajed, Indranil Gupta, Cristina Abad, Nathan Roberts, and Philbert Lin. Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 6. ACM, 2013.

[29] Marcelo Veiga Neves, César AF De Rose, Kostas Katrinis, and Hubertus Franke. Pythia: Faster big data in motion through predictive software-defined network optimization at runtime. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 82–90. IEEE, 2014.

[30] Zhao Li, Yao Shen, Bin Yao, and Minyi Guo. Ofscheduler: a dynamic network optimizer for mapreduce in heterogeneous cluster. *International Journal of Parallel Programming*, pages 1–17, 2013.