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

Yan Liu, Di Niu, *Member, IEEE,* Majid Khabbazian, *Senior Member, IEEE*

Department of Electrical and Computer Engineering,
University of Alberta, Edmonton, AB Canada

Abstract—Data transfers happen frequently in server clusters for software and application deployment, and in parallel computing clusters to transmit intermediate results in batches among servers between computation stages. This paper presents *Cooper*, an optimized prototype system to speedup multi-batch data transfers among a cluster of servers, leveraging a theoretically proven optimal algorithm called "coded permutation gossip," which employs a simple random topology control scheme to best utilize bandwidth and decentralized random linear network coding to maximize the useful information transmitted. On a process-level coding-transfer pipeline, we investigate the best block division, batch division and inter-batch scheduling strategies to minimize the broadcast finish time in a realistic setting. For batch-based transfers, we propose a scheduling algorithm with low overhead that overlaps the transfers of consecutive batches and temporarily prioritizes later batches, to further reduce the broadcast finish time. We describe an asynchronous and distributed implementation of *Cooper* and have deployed it on Amazon EC2 for evaluation. Based on results from real experiments, we show that *Cooper* can almost double the speed of data transfers in computing clusters, as compared to state-of-the-art content distribution tools like BitTorrent, at a low CPU overhead.

Index Terms—content distribution; computer cluster; gossips; random linear code; parallelism; pipelining; scheduling

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 lab, educational software may need to be distributed to every computer for student lab sections, while on some other day, all the lab computers may need to be reinstalled with customized applications to run scientific experiments. As another example, in iterative big data computing clusters such as *Spark* [1] and *Dryad* [2], which are designed to solve machine learning tasks in parallel, large amounts of parameters and intermediate calculations must be transferred among nodes between computation stages, as shown in Fig. 1. Measurements [3] 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

This work was supported by NSERC Discovery Grants. Yan Liu is currently affiliated with EMC Corporation, Edmonton, AB Canada.

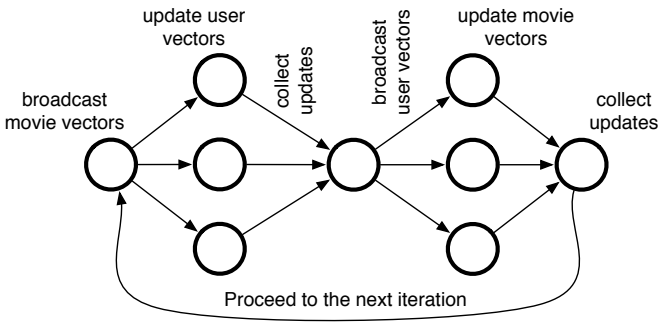


Fig. 1. Workflow of a collaborative filtering algorithm in a cluster [3]. Each circle represents a server.

[4], and spam link identification on Twitter [5]. In all these jobs, an efficient content distribution utility is needed to expedite data dissemination among a cluster of servers.

In this paper, we present *Cooper* (COding On PERmutation topologies), an optimized prototype system to speedup data dissemination in server/computing clusters. *Cooper* arranges all nodes in dynamic random permutations, or ring topologies, 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 through local area networks (LAN) with roughly homogenous network and I/O capacities, permuted connections are not only feasible, but they can also 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 has both an upload and download capacity of 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, our prior work has shown [8], [9] that 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 $k + \lceil \log_2 n \rceil + C$ [9], with $C \approx 4$, which is close to the theoretical limit of $k + \lceil \log_2 n \rceil$ rounds. Therefore, coded permutation gossip enjoys the advantages of gossiping protocols in terms of robustness and decentralization, while still achieving near-optimal utilization of network resources as in tree-packing structured multicast.

However, there exists a gap between the theoretically proved benefits mentioned above and the reality. In this paper, we take one step further to implement and optimize the coded permutation gossip in reality, while addressing the following important practical questions: 1) With encoding and decoding latencies considered, can coded transmission indeed achieve its theoretical benefit? 2) Given a file of a fixed size, how many blocks should the file be chunked into to achieve the best performance in practice? 3) Can we further reduce the broadcast finish time by dividing the file into smaller batches and distribute each batch instead? 4) If multiple files are to be disseminated, can we smartly schedule different files so that each file will finish earlier than transferred sequentially one by one?

To answer these questions, in this paper, we make multiple contributions from both systems and theoretical aspects in the design of *Cooper*. *First*, we use an asynchronous pipelining technique to perform coding operations and data transfers as parallel processes on each node, which successfully hides coding latency if the file is chunked into a proper number of blocks. *Second*, to lower coding complexity, usually a file is broken into smaller batches for dissemination. We consider multi-batch data dissemination, where coded permutation gossip is applied onto each batch. We propose a simple priority-based scheduling strategy to intelligently overlap the dissemination of consecutive batches and prove that a further saving of $\Theta(\log n)$ on finish time can be achieved as compared to non-batched file dissemination. We show the counterintuitive result that if we do not always prioritize earlier files but temporarily assign priority to a later file, the broadcast can finish faster. Similarly, if multiple files are to be broadcasted from the same source, we can reduce the finish time of each file by overlapping the transmissions of different files and temporarily prioritizing later files, at a cost of constant delay for the first file. *Third*, we design an asynchronous transfer control system to convert the theoretical model into real-world implementation. *Fourth*, we also investigate the feasibility of using off-the-shelf commodity multi-core processors on an asynchronous yet fine-tuned coding-transfer pipeline to further enhance performance.

We have prototyped *Cooper*, using *Apache Thrift* [10], an open-source software library first developed at Facebook, to expedite the implementation of efficient and scalable backend services as well as using Boost C++ libraries to handle asynchronous events. 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 with network transfers. In addition, it can also utilize the multi-core processors widely available on most modern computers to launch multiple light-weight encoders, which further expedite the broadcast process.

We have deployed *Cooper* on Amazon EC2 and performed extensive experiments to evaluate the proposed theoretically inspired algorithms. We demonstrate that *Cooper* can reduce the time to broadcast a single file by more than 40% over a state-of-the-art BitTorrent [11] system and a random block dissemination scheme with buffer negotiation in real computing clusters in the cloud.

2 System Model and Background

Consider a theoretical gossip model adopted in [7], [8], [9] for data block transfers. 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*. The upload and download bandwidth capacities of each node are both one bock per round. The reason is that nodes in a same server cluster usually have homogeneous I/O and network capacities. The objective is to disseminate all k blocks to all n nodes. Clearly, the most time-consuming and challenging case is *one-to-all broadcast*, where only a single source node holds all k blocks initially, which will be the focus of this paper as shown in Fig. 1.

The above model is more suitable to a cluster environment than to the Internet, as nodes in a cluster are often collocated geographically in datacenters and usually have homogeneous configurations of CPU and bandwidth. These facts eliminate the need for sophisticated bandwidth allocation schemes (such as tree-packing) as required in structured multicast in heterogenous networks and justifies the feasibility of simple gossip-like protocols.

2.1 Coded Permutation Gossips

It is a classical result that under the gossip model above, the best possible broadcast finish time is $k + \lceil \log_2 n \rceil$ rounds [7]. Intuitively speaking, in the extreme case of $k = 1$, a simple folklore algorithm based on recursive doubling provides an optimal solution for this problem which requires $\lceil \log_2 n \rceil$ rounds. On the other hand, when there is only one node to receive blocks, it takes k rounds to send k blocks to the node. When n and k are arbitrary positive integers, there is a fully centralized sender-receiver pairing schedule with a block selection strategy that can achieve the best broadcast finish time

of $k + \lceil \log_2 n \rceil$ rounds. Since centralized scheduling is not practical, decentralized gossip schemes have been widely studied to approach the above limit, as discussed in detail in Sec. 7.

A decentralized "coded permutation gossip" algorithm is described in our prior work [8], [9] based on a simple permutation rule. As compared to other related work on decentralized gossiping algorithms, e.g., [12], [13] to be reviewed in Sec. 7, coded permutation gossip conforms to both node upload/download capacities while achieving near-optimal broadcast finish times instead of only order-optimality as in [12].

Coded Permutation Gossip: 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 a random linear code) to node u_j where $j = i + 1 \mod n$.

The permutation rule can easily be realized in a computing cluster, where we have almost homogeneous server download/upload capacities with full control of the topology. On the other hand, with permutation-based receiver selection, data transmission is still fully distributed via coded gossip. Therefore, this semi-decentralized algorithm takes advantage of having full control over topologies, while avoiding the hassle of carefully scheduling block transfers. In other words, the coded permutation gossip enjoys the high network utilization just like structured multicast as well as the robustness and ease of implementation of decentralized gossips.

In theory, coded permutation gossip can achieve a broadcast finish time of $k + O(\log n + \log k)$ rounds (with high probability for a field size $q > n$) [9], which is close to the theoretical limit of $k + \lceil \log_2 n \rceil$ [7] previously only achievable by centralized scheduling, 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 $k + \lceil \log_2 n \rceil$.

However, all the prior results mentioned above are based on a time-slotted model for a single file. In this paper, we aim to perform a reality check of the coded permutation gossips and present a highly optimized prototype implementation in the presence of coding latency (which is widely believed to hurt throughput), taking into account the need for multi-file or multi-batch scheduling.

3 Pipelining and the Best k

In practical broadcast transfers, what is the best number of blocks that a file should be divided into? If coding latency is considered, can we still achieve the theoretically proved benefit of coded permutation gossip in practice? In this section, we show that if we properly decide the number of blocks k that a file is broken into, coding and transmission operations, although happening *asynchronously*, can almost be aligned in a *pipeline*, where the output of the coding (transmission) is the input of transmission (coding). As a result, coding and transmission can be performed in parallel, hiding the coding latency.

3.1 Modelling Computational Latency

To answer the questions above, we assign realistic parameters to the time-slotted model described in Sec. 2. Assume the file has a size of F, 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 coded permutation gossip is within $k + \lceil \log_2 n \rceil + C$ rounds, the real finish time *considering transfer latency only* will 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, the bandwidth W, and the number of nodes n are fixed, the larger the k is, the smaller the finish time. Surprisingly, when $k \to \infty$, the finish time of the last node *and every node* will be F/W, leading to a speedup of $n - 1$ times, as compared to sequentially transferring the file to $n - 1$ receiver nodes one by one. 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 quantify the computation time associated with coding operations, we 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. We analyze how computation overhead of RLNC can affect dissemination. We 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. *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. 2. We observe that the total computation time per node *1)* is linear to the block size

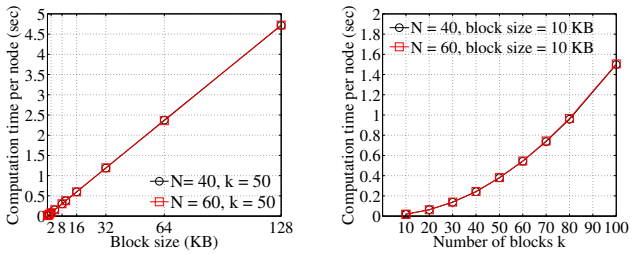


Fig. 2. Mean encode + decode time per node in the entire broadcast session.

B, and *2)* is a convexly increasing function of the number of blocks k. Therefore, for a fixed n and field size q on a given processor, we can do some profiling similar to that in Fig. 2. We can 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. 2(b).

Therefore, there is a tradeoff between throughput efficiency and coding complexity as k varies. A larger k will increase computational latency. Yet, 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 terms of seconds (instead of rounds)?

3.2 Asynchronous Pipelining

Before finding out the best k, let us describe how the transmission and coding operations happen in the system. Apparently, we do not have to wait until a previous block is sent out already to start encoding the next block to be sent. Instead, we can use pipelining, a common idea in modern computer architectures, to parallelize the transmission and computation. Specifically, as shown in Fig. 3, in each "round", we let each node do the following 3 steps simultaneously as the resources they mainly rely on 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.

However, it is worth noting that our pipelining mechanism is completely *asynchronous*, rather than relying any clocks like in a pipelined processor. *The key is to divide the file into a best number k^* of blocks, in order to exploit the theoretical throughput benefit of coded permutation gossip, while the coding latency does not grow to the point that delays broadcast finish time.* Intuitively speaking, if the time spent on encoding each block (including the progressive decoding that happens before it) by each node is smaller

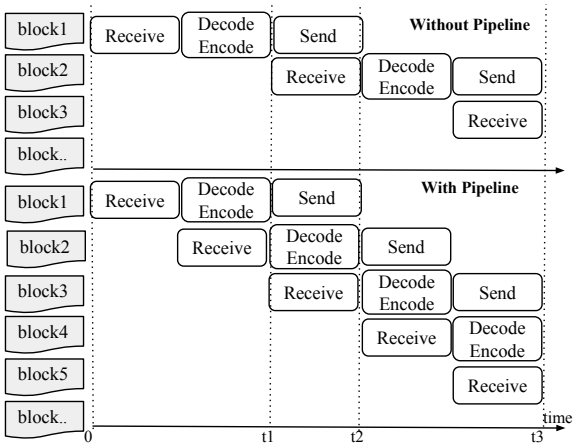


Fig. 3. The transmission and coding processes with and without pipeline on a certain node.

than the time to transfer (send or receive) this block, coding operations will not become the bottleneck of the pipelined process at each node. However, if the time spent on coding for each block is greater than the time to transfer a block, computation time associated with coding operations will become the bottleneck.

Therefore, the broadcast finish time (in terms of seconds) in a pipelined implementation is approximately given by

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

depending on whether transmission or computation constitutes the bottleneck. Thus, we have the following straightforward proposition:

Proposition 1. *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 $T_{\mathsf{Broadcast}}(k) = T_{\mathsf{Compute}}(k) > T_{\mathsf{Compute}}(k^*)$. On the other hand, if we have a $k < k^*$, then $T_{\mathsf{Transfer}}(k)$ increases and $T_{\mathsf{Compute}}(k)$ decreases, and thus $T_{\mathsf{Broadcast}}(k) = T_{\mathsf{Transfer}}(k) > T_{\mathsf{Transfer}}(k^*)$. In other words, the lowest broadcast finish time is achieved by choosing the optimal k^* to equalize the computation time and transfer time *per block* at each node, although computation and transfers happen asynchronously.

For example, as illustrated in Fig. 3, under the best k^*, computation and transfer times are *roughly* equalized. With pipelining, each particular node can perform three actions simultaneously, while without pipelining, only two actions are performed. As a result, without pipelining, only 3 blocks have been received by the node at time t_3, whereas with pipelining, 5 blocks have been received. Note that in our actual implementation to be described in Sec. 5, pipelining is achieved *asynchronously* only by fine-tuning k to equalize transfer and computation processes, instead of through a synchronized clock.

Therefore, Fig. 3 is only a conceptual illustration, while in reality, computation and transfer processes may only *roughly* align.

4 Multi-Batch Scheduling

In reality, to lower the computational complexity of network coding, a large file is usually divided into m batches or generations to be disseminated sequentially [14], [15]. The batch-based approach can significantly reduce the CPU burden, because each batch consists of k smaller blocks, each of size B/m, and thus the CPU utilization on encoding each block will be much lower.

It is worth noting that we cannot save broadcast finish time by just dividing a files into m batches for dissemination. In other words, disseminating a large file of size F in k^* blocks *takes the same time* as dividing the file into m "batches", each comprised of k^* smaller blocks, and sequentially disseminating the m batches one by one using the same pipelining scheme in Sec. 3. And according to (1), the best k^* is the same whether a batch-based approach is applied or not. The reason is that, according to (1), the best k^* is not affected by the file size F; it only depends on the bandwidth W, the processor used, the number of nodes n and the field size q for coding.

In this section, we ask the question—if a multi-batched approach is used, is it possible to further reduce the broadcast finish time as compared to broadcasting the whole file? Intuitively speaking, the answer is yes due to multi-batch *overlapping opportunities*; when the first batch is close to finish and few nodes can receive innovative blocks in this batch, we can start the dissemination of the second batch to better utilize idle links. Leveraging multi-batch overlaps, in this section, we will investigate multi-batch scheduling strategies to further beat the best broadcast finish time dictated by k^* in non-batched broadcast of a single file. In addition, if multiple files are to be broadcasted from the source, we may also take advantage of multi-file scheduling to beat the sequential transfer of these files.

In particular, we propose scheduling algorithms to decide the priorities of encoding and transmission among blocks of different batches (or files), when a node has blocks from multiple batches for transmission. We show that, for a file of size F, our multi-batch strategy can further save the broadcast finish time by $\Theta(\log n)\cdot\frac{F}{k^*\cdot W}$ in terms of seconds, as compared to non-batched broadcast with the best k^* in Sec. 3, which has a finish time of $(k^*+\lceil\log_2 n\rceil+C)\cdot\frac{F}{k^*\cdot W}$.

4.1 Early Batch First

We first consider two batches of blocks with random broadcast finish times of $\mathscr{F}_1$ and $\mathscr{F}_2$ rounds, respectively. If the two batches are broadcast *sequentially*, the overall expected broadcast finish time will be $\mathbf{E}[\mathscr{F}_1+\mathscr{F}_2]=\mathbf{E}[\mathscr{F}_1]+\mathbf{E}[\mathscr{F}_2]$ 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 $\mathscr{F}_1$, since it is always prioritized, while the broadcast finish time of both batches becomes less than $\mathbf{E}[\mathscr{F}_1+\mathscr{F}_2]$. 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, we point out the possibility of a new overlapping scheme, 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, while only delaying the finish time of batch 1 by $O(1)$.

Theorem 2. *Let $l=\lfloor\frac{\log n}{2}\rfloor$. Suppose that the source starts broadcasting batch 1 and batch 2, respectively, in round 1 and round $\mathscr{F}_1-l+1$ (i.e., l rounds before the finish time of the first batch). Suppose that from round $\mathscr{F}_1-l+1$ to $\mathscr{F}_1$, 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 $\mathscr{F}_1+1$ onward, the priority is given back to batch 1. Then, the expected broadcast finish time of both batches will be at most $\mathbf{E}[\mathscr{F}_1]+\mathbf{E}[\mathscr{F}_2]-l+\frac{2.7}{1-\frac{2}{\sqrt{n}}}$. In addition, the expected broadcast finish time of batch 1 is at most $\mathbf{E}[\mathscr{F}_1]+\frac{2.7}{1-\frac{2}{\sqrt{n}}}$.*

Please refer to the appendix for a proof of the theorem above. Theorem 2 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.

Suppose there are m batches in total. We can repeat the overlapping process explained in the statement of Theorem 2. For $b=3,\ldots,m$, let the transfer of batch b start at $l=\lfloor\frac{\log n}{2}\rfloor$ 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 later. Again, as

before, batch b will have priority in the first l rounds after the start of its transmission, and then the priority is given back to batch $b-1$ until batch $b-1$ finishes. Then, the following corollary is a result derived from Theorem 2. (Please refer to the appendix for a proof sketch.)

Corollary 3. *Suppose there are $m \geq 2$ batches. The expected saving in the finish time of each batch $b = 2, \ldots, m$ using the overlapping scheme explained above over the sequential broadcasts of batches $1, \ldots, b$ is at least*

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

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

We can now use Corollary 3 to quantify the benefit of broadcasting a file of size F in m batches, using the multi-batch overlapping scheme above. The saving on the finish time of the final batch m is at least $\Theta(m \log n)$. Recall that for a file of size F, the optimal finish time of the non-batched broadcast in terms of seconds is $(k^* + \lceil \log_2 n \rceil + C) \cdot \frac{F}{k^* \cdot W}$, which is the same as the time to broadcast m batches (each of size F/m) sequentially:

$$m(k^* + \lceil \log_2 n \rceil + C) \cdot \frac{F}{m \cdot k^* \cdot W}.$$

With overlapping, for a large n, the saving is at least $(m-1)\lfloor \frac{\log n}{2} \rfloor$ rounds with respect to the $m(k^* + \lceil \log_2 n \rceil + C)$ rounds in total for m batches. Therefore, the saving in broadcast finish time in terms of seconds using the above m-batch overlapping scheme over the non-batched broadcast is

$$T_{\text{multi-batch saving}}(m) = \frac{m-1}{m} \cdot \left\lfloor \frac{\log n}{2} \right\rfloor \cdot \frac{F}{k^* \cdot W}, \qquad (2)$$

where each batch is still divided into k^* blocks. From (2), we find that we do not have to divide the original file into too many batches to harvest the most benefit from multi-batching. As the number of batches m that the file is divided into increases, $\frac{m-1}{m}$ will quickly approach its upper bound 1.

Admittedly, the overlapping scheme in Theorem 2 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 $\mathscr{F}_1$ is a random variable. However, the insight from Theorem 2 and Corollary 3 is that 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. Based on this observation, we propose another practical multi-batch scheduling scheme called Overlap-2:

Algorithm 2. (Overlap-2) *Suppose batch 1 and batch 2 have k_1 and k_2 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.*

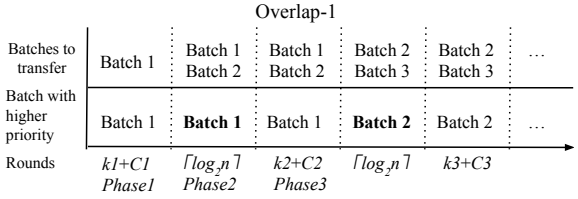


Fig. 4. Priority-based multi-batch scheduling algorithm: Overlap-1.

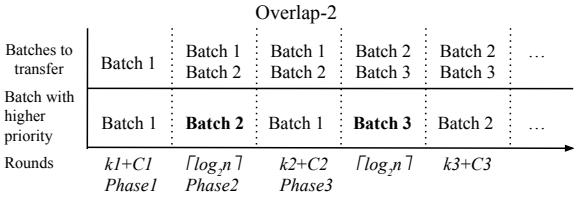


Fig. 5. Priority-based multi-batch scheduling algorithm: Overlap-2.

Counterintuitively, we show through experiments in Sec. 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.

The first 3 phases in Fig. 4 and Fig. 5 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 [9], 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 $k_2 + C_2$ 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

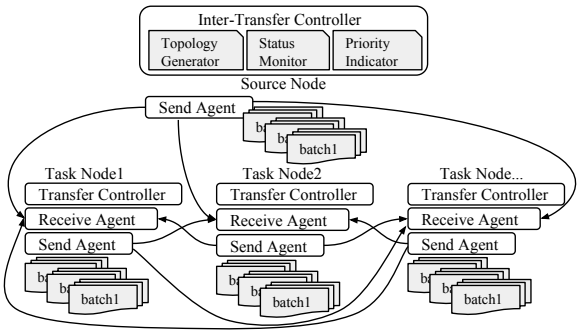


Fig. 6. Architecture of Cooper transmission system.

Overlap-1 and Overlap-2 is illustrated in Fig. 4 and Fig. 5 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 $k_1+k_2+\lceil \log_2 n \rceil + C$ rounds (achieved when cross-batch encoding is allowed) [7], [9], there is no reason to start batch 3 before round $k_1+k_2+\lceil \log_2 n \rceil + C$. The same principle applies to batch $4, 5, \ldots$ and so on.

5 Asynchronous Implementation

We have implemented an efficient *asynchronous* transmission system named *Cooper*, through Apache Thrift and Boost C++ libraries on Linux Systems. Originally developed at Facebook, Thrift allows us to generate code for inter-node TCP communications through simple service interface definition files, while we use Boost libraries to handle asynchronous multithreaded events. There are mainly two kinds of nodes in *Cooper*: the source node and task nodes. The 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. 6, 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 the 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 adjusts the priority of the batches, based on a given multi-batch scheduling algorithm.

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 section. 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*, we propose to pipeline transmission and computation to hide the coding latency. For this purpose, we launch the following four 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.

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. 7. 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

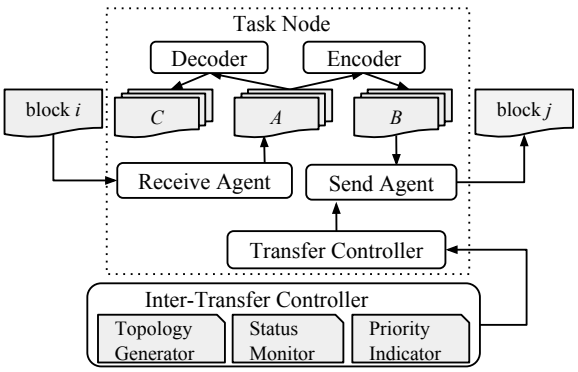


Fig. 7. Asynchronous processes executed on each task node. The Encoder keeps generating encoded blocks, while the Send Agent keeps sending out the newly encoded blocks. The two processes are independent.

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 Sec. 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 Sec. 6, and thus there is no need to perform decoding progressively for each received block.

5.2 Process-Level Parallelism

We now describe how to achieve the pipelining of transmission and coding on each node in this *asynchronous* distributed system. Specifically, we aim to equalize the running time of the following 3 asynchronous processes on each node (with decoding only happening 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, we 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. We will show through the experiments in Sec. 6 that perfect pipelining

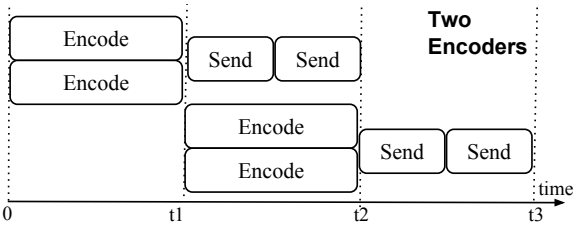


Fig. 8. Processes with two parallel encoders.

with equalized encoding and transfer times will indeed achieve the minimum broadcast finish time.

5.3 Inter-Batch Scheduling

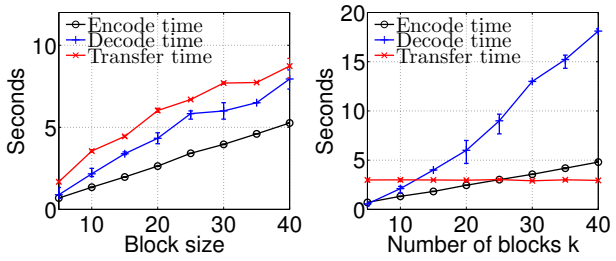
We implement both Overlap-1 and Overlap-2, the two multi-batch scheduling algorithms proposed in Sec. 4 which assign different priorities to different batches of data. In an asynchronous transmission system, there is no notion of rounds. Therefore, we define each "round" as the time that the source spends 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-2, 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, we 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, we 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. We further explore the benefit of additional parallelism of launching multiple encoding processes simultaneously. More specifically, we launch two encoding processes, each running a different core in parallel, while the send processes still work sequentially, as show in Fig. 8. Therefore, if we 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} we will have two encoded blocks generated in parallel as well as two encoded blocks (generated during the previous T_{encode}) transmitted sequentially. This way we still achieve perfect pipelining of transfer and coding, as if the coding latency is hidden.

However, with two parallel encoders, we end up having a larger k^* which almost doubles the k^* with one encoder (as will be shown in Sec. 6). This means that the overhead part $\lceil \log_n \rceil + C$ in the finish time becomes less



(a) vs. block size (b) vs. k

Fig. 9. Mean computation and transfer time per node with error bars, as block size or block number k varies.

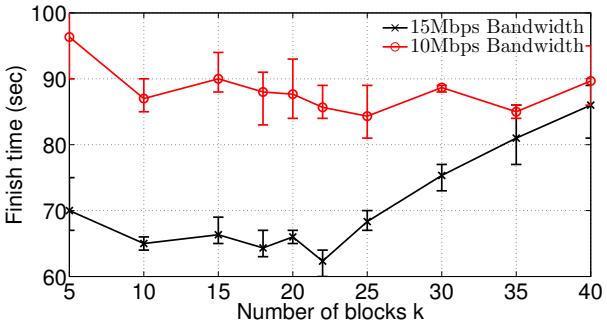


Fig. 10. The finish time under different k with error bars (when file size is fixed to 100 MB).

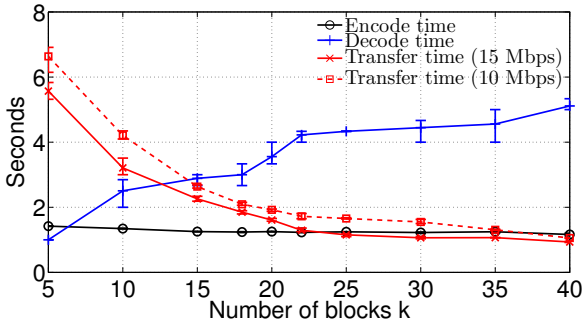


Fig. 11. Mean encode and transfer time per node per block with error bars (when file size is fixed to 100 MB).

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.

6 Performance Evaluation

In this section, we provide an extensive performance evaluation of the proposed mechanisms by deploying and measuring the developed asynchronous prototype system *Cooper* on Amazon EC2. We 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, we set the upload capacity and download capacity at each node to be either all 10 Mbps or all 15 Mbps. Although datacenters nowadays are provisioned with 10-40 Gbps networks, from our measurements on Amazon EC2, we observe that due to network sharing between tenants, the network bandwidth between two EC2 instances is always between 10 Mbps (during peak usage hours) and 15 Mbps (during trough usage hours). Furthermore, the bandwidth between two EC2 instances of the same kind measured around the same time of the day roughly remains the same.

For each data point plotted in the figures of this section, at least 10 experiments are done and the average is taken. Also, the error bars in the figures of this section indicate the *maximum* and *minimum* values from the corresponding sets of experiments. It is worth noting that in real experiments, the workload and network environment may fluctuate at any time (although to a limited extent). Therefore, completely smooth and convex curves are not always observed. We will explain the insights from the big trend in the experimental results. We also attempt to maintain the same experimental environment for all experiments in the same group. Our experimental results suggest that although the network and computing environment in EC2 is only roughly homogeneous instead of exactly uniform, Cooper can speedup both single-file and multi-batch data transfers as compared against state-of-the-art systems like BitTorrent.

6.1 Evaluation of Pipelining

We 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 running Linux Ubuntu 12.04.

Verifying the computation time model. From simulations in Fig. 2), we have observed the computation time per node (with progressive decoding) $T_{\text{compute}} = \beta B k^{\alpha}$, for $\alpha \geq 1$. Now we verify that T_{encode} alone (without progressive decoding as in the implementation) follows the same trend. Fig. 9(a) 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, we observe that the total decoding time in the end is approximately linear to the block size and is not large, justifying the reason why we only pipeline encoding alone and transfer in implementation. Similarly, Fig. 9(b) plots the same variables as the number of blocks k varies, when the block size is fixed to 10 MB, which shows that encoding time per block is convexly increasing as k increases, verifying the simulation results on computation time.

Verifying the Optimality of Pipelining. In Sec. 3, we have argued that the lowest broadcast finish time is achieved by the k^* such that computation time equals to

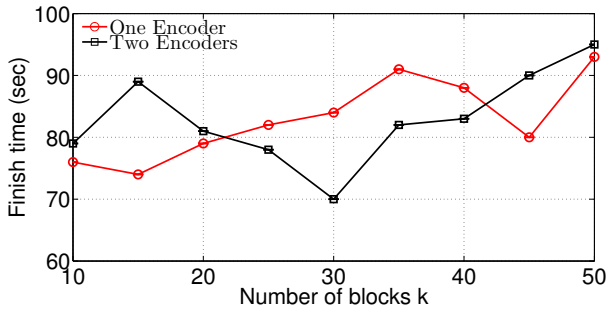


Fig. 12. The finish time comparison between one and two encoders. (Network bandwidth: 15 Mbps)

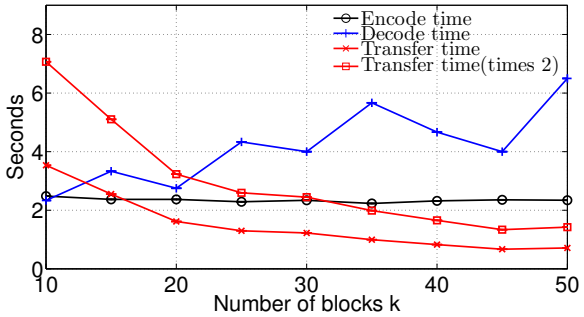
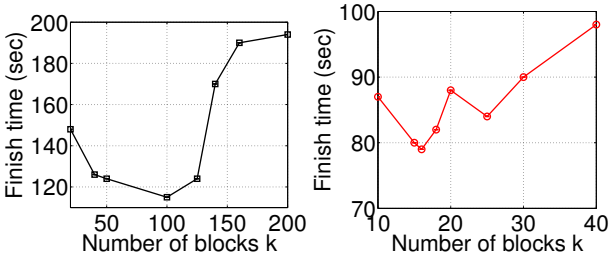


Fig. 13. Mean computation and transfer time per node per round with error bars. (File size:100MB, network bandwidth: 15 Mbps).



(a) Random Block Scheme (b) Cooper

Fig. 14. The finish times of Cooper and RB on 20 t2.medium instances of Amazon EC2. File size: 100 MB.

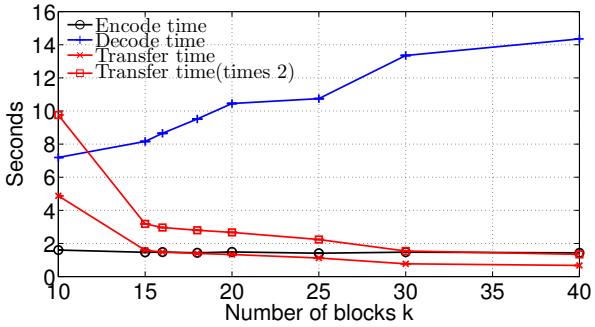


Fig. 15. Mean computation and transfer time per node per round on 20 t2.medium instances of Amazon EC2. (File size: 100 MB)

transferring time. In practical implementation of *Cooper*, we only need to find the k^* that equalizes the encoding time and transferring time per block. We evaluate the times to broadcast a file of a fixed size (100 MB) with *Cooper*, as we chunk the file into different numbers of blocks k, as shown in Fig. 10. 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. 11 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, we 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.

Pipelining with Double Encoders. We further evaluate benefit of the pipelining mechanism with two parallel encoders on two cores. 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, we obtain different computation time measurements from the previous figures.

As shown in Fig. 12, 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, achieved when $k = 30$. By further checking Fig. 13, we 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, we 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

We compare the performance of *Cooper* with BitTorrent and a baseline random block negotiation scheme on Amazon EC2, when *Cooper* uses one encoder, two encoders and batch-based file transfers. We launched 20 t2.medium instances from Amazon EC2, each with 2 cores, 4 GB memory and low to moderate network performance running Linux Ubuntu 14.04. We maintain the same network environments for all experiments in each group. The file size to be broadcast is fixed to 100 MB for all systems. We 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, we 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, we optimize the buffer update frequency. 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. We find that the best performance is achieved on EC2 if each node updates its buffer map to other nodes upon receiving *every other block*. We further tune the number of blocks k for RB, as shown in Fig. 14(a), from which we 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.

BitTorrent (BT): we deploy *rTorrent* [16], which is a command-line-based BitTorrent client written in C++ with high performance, based on the libTorrent libraries for Unix. We carefully optimize the performance of *rTorrent* on EC2 by configuring its various parameters, like number of simultaneous connections.

Tuning *Cooper*. We have already verified that the best broadcast performance 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* we first do some profiling (that is done lightweight on a single node) to find the right k^* for both one-encode and two-encoder cases. As illustrated in Fig. 14(b) and Fig. 15, we 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.

In Fig. 16, we compare the broadcast finish times of a 100 MB file under 5 different schemes: *Cooper* with one, *Cooper* with two encoders, *Cooper* (one encoder, batch-based), Random Block (RB) and BitTorrent (BT). For batch-based *Cooper*, we divide the file into 3 batches and use Overlap-2 as the inter-batch scheduling algorithm. From Fig. 16, we see that *Cooper* (one encoder) has a finish time around 80 seconds, significantly outperforming Random Block and BitTorrent, which finish around 115 seconds and 106 seconds, respectively. Moreover, *Cooper* with two encoders further reduces the broadcast finish time to around 70 seconds.

The best broadcast finish time is achieved by dividing the file into 3 batches, scheduled with Overlap-2, which achieves a finish time of 60 seconds. Therefore, we conclude that, *Cooper* can significantly speedup data dissemination, especially with batch-based dissemination and our inter-batch scheduling algorithm Overlap-2.

To understand the CPU overhead of Cooper, in Fig. 17, we plot the maximum CPU utilization of all the nodes in the network (except the source) in each second under different schemes. We use the `mpstat` command from `sysstat`, a software package that monitors system performance on Linux, to measure the percentage of time that the CPU on each node spends in the user mode (%usr) and system mode (%sys). Under each scheme, we record the %usr value at each node in each second, and get the maximum %usr among all receiver nodes in each second of the session before it finishes. On the other hand, the maximum %sys is always around 3.5%-4.5% under all schemes.

From Fig. 17, we can see that in Cooper, the CPU utilization at a node is closely related to how many blocks it is encoding. When the session first starts, all the receiver nodes have very few blocks. The CPU utilization starts to increase when the nodes receive more blocks. After the first node has finished receiving all k^* blocks, this node will always encode k^* blocks from then on, making the maximum CPU utilization in the network stay at the CPU consumption level for encoding k^* blocks. We did not plot the CPU consumption for the source node, since it always equals to the CPU utilization for encoding k^* blocks.

Comparing Cooper (one encoder) with Cooper (two encoders), we can see that there is a tradeoff between CPU overhead and broadcast finish time. Two encoders yield a shorter finish time because it enables a larger $k^* = 30$ instead of 16, according to (1). However, a larger k^* leads to a higher CPU consumption in the coding operation. Plus, there are two encoders performing the computation at the same time, leading to an overly high CPU overhead. Therefore, two encoders should not be used unless CPU overhead is not a concern in the system.

Cooper (one encoder) achieves an acceptable maximum CPU utilization of around 8%. Cooper (one encoder, 3 batches) further lowers CPU overhead to about 5%, because by dividing file into 3 batches (each composed of $k^* = 16$ blocks), the block size is only $1/3$ of the original block size. Since coding complexity depends on k^* and the block size B, batch-based dissemination can effectively reduce the CPU burden due to coding.

It is worth noting that we can always further reduce CPU utilization by dividing the file into more batches (each composed of k^* blocks). And for an arbitrarily large file (e.g., 1 GB), the CPU overhead does not increase, if we use more batches of the same size (e.g., 30 batches for 1 GB, each still of size 33.33 MB). Since at a particular time, the inter-batch scheduling only decides the priorities between 2 live batches, the scheduling overhead will not increase.

The packet overhead associated with coding is also low. For coding done in $GF(2^8)$, there are $8k$ bits or k bytes of coefficient overhead in each coded block. In our batch-based transfer, $k^* = 16$, while each block is about 2 MB, leading to an negligible coefficient overhead of $16\text{B}/2\text{MB} = 7.63 \times 10^{-4}\%$.

6.3 Multi-Batch Scheduling Algorithms

Fig. 16 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, we evaluate the performance of the proposed inter-batch scheduling algorithms Overlap-1 and Overlap-2 on Amazon EC2, as well as traditional non-overlapping sequential transfers. The experiment environment is the same as in Sec. 6.2. The size of *each* batch is 100 MB. And we apply *Cooper* with a single encoder in all experiments here.

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

As shown in Fig. 18, 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 obvious, especially the gap between Overlap-1 and Overlap-2.

To understand the underlying mechanisms of overlapping, we collect logging data from one experiment, and illustrate the numbers of finished nodes during running time in Fig. 19 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. 20 and Fig. 21. The total number of nodes to receive blocks is 19 (except the source). And suppose there are two files (or batches) to be broadcast.

In Fig. 19, 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, we 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. 20 and Fig. 21, 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.

7 Related Work

Traditional solutions to content dissemination are mostly based on multiple tree construction. Chiu *et al.* [17] and Li *et al.* [18] show that packing Steiner trees of a depth of one or two can achieve the maximum feasible broadcast rate limited by the node download capacity. The rates are then assigned to the constructed multi-trees to maximize the data streaming rate. Chen *et al.* [19] further describe a primal-dual algorithm for the distributed implementation of tree construction and rate allocation. However, tree-based algorithms remain largely centralized with large structure maintenance overhead, making it harder to manage in dynamic environments.

In contrast, gossip algorithms are adaptable to arbitrary network topology and are easy to implement. The random phone call (or random contact) is the most popular model in gossiping literature [12], where in each round, each node and push (or pull) a block to (or from) another node at random. Under this model, it is shown [12] that 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 (c being $3 - 6$), which is order-optimal. This bound is further tightened in [13], which proves a finish time of $k + o(k)$ for the pull version. However, these results do not apply to the broadcast case, where a single source holds all k blocks initially.

More importantly, the success of these algorithms [12], [13] critically rely on the assumption that each node's download capacity (or upload capacity in the pull case) must be *unbounded*, to allow multiple nodes to push to (or pull from) a node at the same time. This assumption is perhaps acceptable for Internet end-users with asymmetric downlink and uplink capacities, but is hard to justify for a homogeneous cluster with roughly uniform and symmetric capacities. Moreover, with the random phone call model, some nodes may not be chosen by any senders (or receivers) in a round, causing link underutilization. Different from previous work [12], [15], [17]–[23] that assume *unbounded* node download capacities, we consider both node upload and download bandwidth constraints which may both be present in real computer clusters.

Shifting away from the random phone call model, our previous works [8], [9] have theoretically shown that simple permutation (ring) topologies can approach the optimal broadcast time of $k + \lceil \log_2 N \rceil$ (not only order-optimal), whether the data blocks are in one node or

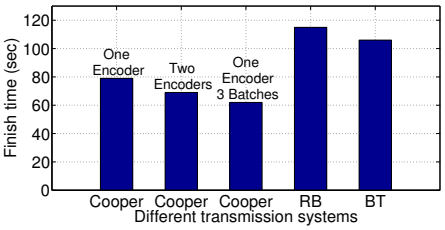


Fig. 16. Comparison of finish times between Cooper and baseline algorithms. (File size: 100 MB)

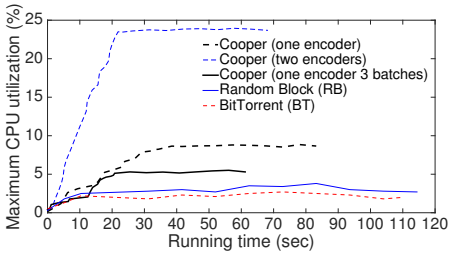


Fig. 17. The maximum CPU utilization of all the nodes (except the source) in the network under different schemes. (File size: 100 MB)

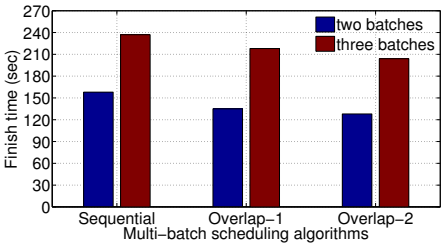


Fig. 18. The finish times of transferring 2 and 3 batches using different priority-based algorithms.

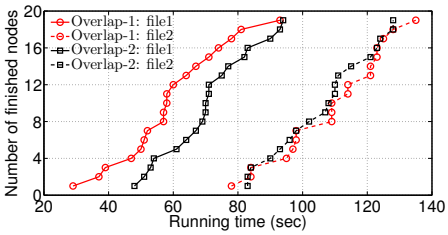


Fig. 19. The number of finished nodes as time goes when using different priority-based algorithms to transfer 2 files.

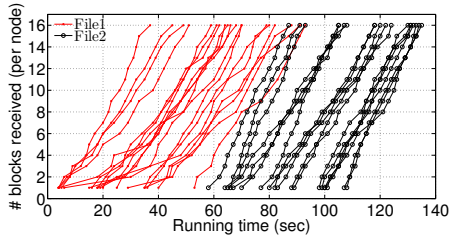


Fig. 20. The number of blocks received by each node as time goes, when using Overlap-1 to transfer 2 files.

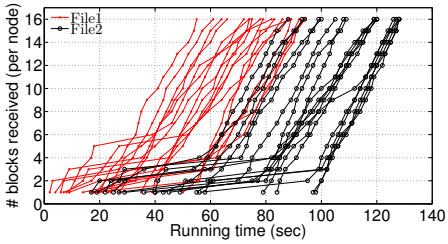


Fig. 21. The number of blocks received by each node as time goes, when using Overlap-2 to transfer 2 files.

spread across the network initially. Such a permutation topology can easily be implemented in any controllable server clusters. From a theoretical point of view, in this paper, we further consider multi-batch data dissemination, and propose a priority-based scheduling strategy to smartly overlap the distribution of consecutive batches. We show the counterintuitive result that a further saving of $\Theta(\log n)$ rounds on finish time is possible as compared to non-batched dissemination, if we temporarily assign priority to a later batch.

In computing clusters, multi-node transfer operations have a significant impact on the performance of cluster applications. Orchestra [3] is proposed as an architecture that enables global control both across and within different transfer tasks to optimize the transfer performance in clusters. For data broadcasts, Orchestra uses a BitTorrent-like scheme called *Cornet* to speed up data dissemination. [24] proposes Torchestra to reduce transfer delays by separating interactive and bulk traffic onto two different TCP connections between each pair of nodes. A general deployment advisor called ClouDiA is proposed in [25] which selects application node deployments, minimizing either the largest latency between application nodes, or the longest critical path among all application nodes. In [26], a system is presented to reduce the skewness 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 software-defined networking to avoid hotspots in the network. And in [27], a dynamic network optimizer called OFScheduler is proposed for heterogeneous clusters to relieve the network traffic during the execution of MapReduce jobs, which focuses on reducing bandwidth competition, balancing the workload of network links and increasing bandwidth utilization.

A piece of closely related work is using network coding for large scale content distribution [15] in heterogeneous wide-area P2P networks. However, having adopted random topology control and receiver selection just like in BitTorrent (while transferring coded blocks), [15] cannot guarantee to minimize the broadcast finish time or maximize network utilization. Nor did it provide a rigorous study on the choice of block division k, batch (or generation in their terms) division, inter-batch coordination, or topology management. Different from [15], Cooper uses a much smaller k to hide coding latency via pipelining, and smart inter-batch scheduling policies to further decrease broadcast finish time.

8 Conclusion

In this paper, we present the design of *Cooper*, a highly optimized, asynchronous and distributed prototype system to speedup data dissemination in computer clusters, in which we leverage the coded permutation gossips to approach the optimal broadcast finish time. In an asynchronous process-level coding-transfer pipeline, we investigate the best block division strategy, as well as the benefit of batch-based transfers. We propose a smart multi-batch scheduling algorithm with theoretical performance guarantees to further enhance performance.

We deploy *Cooper* on Amazon EC2 and perform extensive real-world experiments to verify the proposed theoretically inspired algorithms, as compared to state-to-the art content distribution utilities, including BitTorrent and an optimized random block negotiation scheme. Based on results from real experiments, we 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. Furthermore, with batch-based transfers and the proposed multi-batch scheduling, such savings reach 44% and 48%, with a controllable and low CPU overhead.

REFERENCES

[1] 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.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. of EuroSys*, 2007.

[3] M. Chowdhuryand, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *Proc. of ACM SIGCOMM*, 2011.

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

[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] T. Ho, R. Koetter, M. Medard, D. R. Karger, and M. 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*, vol. 55, pp. 95–105, 1994.

[8] D. Niu and B. Li, "Asymptotic optimality of randomized peer-to-peer broadcast with network coding," in *Proc. of IEEE INFOCOM*, 2011.

[9] M. Khabbazian and D. Niu, "Achieving Optimal Block Pipelining in Organized Network Coded Gossip ," in *Proc. of ICDCS*, 2014.

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

[11] 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.

[12] S. Deb, M. Médard, and C. Choute, "Algebraic Gossip: A Network Coding Approach to Optimal Multiple Rumor Mongering," *IEEE Trans. Inform. Theory*, vol. 52, no. 6, pp. 2486–2507, June 2006.

[13] B. Haeupler, "Analyzing network coding gossip made easy," in *Proc. of ACM STOC*, 2011.

[14] P. A. Chou, Y. Wu, and K. Jain, "Practical Network Coding," in *Proc. of the 41st Annual Allerton Conference on Communication, Control and Computing*, October 2003.

[15] C. Gkantsidis and P. Rodriguez, "Network Coding for Large Scale Content Distribution," in *Proc. of IEEE INFOCOM*, 2005.

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

[17] 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.

[18] 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.

[19] 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.

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

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

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

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

[24] D. Gopal and N. Heninger, "Torchestra: Reducing interactive traffic delays over tor," in *Proceedings of the 2012 ACM Workshop on Privacy in the Electronic Society*. ACM, 2012, pp. 31–42.

[25] T. Zou, R. Le Bras, M. V. Salles, A. Demers, and J. Gehrke, "Cloudia: a deployment advisor for public clouds," in *Proceedings of the VLDB Endowment*, vol. 6, no. 2. VLDB Endowment, 2012, pp. 121–132.

[26] M. V. Neves, C. A. De Rose, K. Katrinis, and H. 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*. IEEE, 2014, pp. 82–90.

[27] Z. Li, Y. Shen, B. Yao, and M. Guo, "Ofscheduler: a dynamic network optimizer for mapreduce in heterogeneous cluster," *International Journal of Parallel Programming*, pp. 1–17, 2013.

Yan Liu. Yan Liu received the B.Engr. degree in Software Engineering from the School of Software, Northeast Normal University, China, in 2008 and the M.Sc. degree from the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Alberta, Canada, in 2015. Since January 2015, she has been with EMC Corporation, Edmonton, Alberta, Canada, where she is currently a software engineer in the storage team.

Di Niu. Di Niu received the B.Engr. degree from the Department of Electronics and Communications Engineering, Sun Yat-sen University, China, in 2005 and the M.A.Sc. and Ph.D. degrees from the Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, in 2009 and 2013. Since September, 2012, he has been with the Department of Electrical and Computer Engineering at the University of Alberta, where he is currently an Assistant Professor. His research interests span the areas of cloud computing and storage, distributed systems, data mining, and statistical machine learning. Dr. Niu is a member of IEEE and ACM.

Majid Khabbazian. Majid Khabbazian received his undergraduate degree in computer engineering from Sharif University of Technology, Iran, and his Masters and Ph.D. degrees both in electrical and computer engineering from the University of Victoria and the University of British Columbia, Canada, respectively. From 2009 to 2010, he was a postdoctoral fellow at the Computer Science And Artificial Intelligence Lab, MIT. Currently, Dr. Khabbazian is an assistant professor in the Department of Electrical and Computer Engineering, University of Alberta, Canada. His research interest is mainly in designing efficient, and reliable algorithms for distributed systems. Dr. Khabbazian is a senior member of IEEE, and a member of ACM.

APPENDIX

Proof of Theorem 1:

Lemma 4. *Let $\mathcal{U} = \{u_1, \ldots, u_n\}$ be a set with $n \geq 2$ nodes, and $\mathcal{S} \subseteq \mathcal{U}$ be a subset of it. Assume that $u_1 \in \mathcal{S}$. Let Π be a permutation of nodes in $\mathcal{U}$ selected uniformly at random from the set of all permutations that start with node u_1. Then, the probability that there two adjacent nodes u and v in Π such that $\{u, v\} \subseteq \mathcal{S}$ is at most $\frac{l^2}{n}$, where $l = |\mathcal{S}|$ is the cardinality of $\mathcal{S}$.*

Proof: This is clearly true for $l = 1$. So let us assume $l \geq 2$. The total number of permutations starting with u_1 is $(n-1)!$. The number of permutations starting with u_1 that include at least two adjacent nodes in $\mathcal{S}$ is at most

$$\underbrace{(l-1)(n-2)!}_{\substack{\text{the 2nd node of } \Pi \text{ is in } \mathcal{S} \\ \text{other nodes are permuted} \\ \text{randomly}}} + \underbrace{(n-l)\left\{(l-1)(l-2)\right\}(n-3)!}_{\substack{\text{the 2nd node of } \Pi \text{ is not in } \mathcal{S}; \\ \text{there are, however, two adjacent nodes} \\ \text{of } \Pi \text{ that are in } \mathcal{S}}}, \quad (3)$$

where the term inside the curly brackets is the number of combinations of gluing a pair of remaining nodes in $\mathcal{S}$. By (3), the probability that there two adjacent nodes u and v in Π such that $\{u, v\} \subseteq \mathcal{S}$ is at most

$$\begin{aligned}
&\frac{(l-1)(n-2)! + (n-l)\left\{(l-1)(l-2)\right\}(n-3)!}{(n-1)!} \\
\leq &\frac{(l-1)(n-2)! + (n-2)\left\{(l-1)(l-2)\right\}(n-3)!}{(n-1)!} \\
= &\frac{(l-1) + (l-1)(l-2)}{n-1} = \frac{(l-1)^2}{n-1} \leq \frac{l^2}{n}.
\end{aligned}$$

□

We call a node *interrupted* in a round of the *overlapping phase*, if it receives a (coded) block of the second batch in that round. To prove the theorem, we first show that uninterrupted nodes (those that are never interrupted) receive the first batch by the end of round $\mathscr{F}_1$ (as if there is no overlapping phase). Then, we prove that each interrupted node misses, on average, $O(1)$ coded blocks of the first batch in the *overlapping phase*. Also, we show that, on average, $O(1)$ rounds after the overlapping phase, interrupted nodes will receive enough coded blocks to decode the first batch. Therefore, the expected finish time of the first batch will be $\mathbf{E}[\mathscr{F}_1] + O(1)$. Since the second batch has priority in the overlapping phase (which has l rounds), after the finish of the first batch, the expected number of rounds the second batch needs to finish is at most $\mathbf{E}[\mathscr{F}_2] - l$. Consecutively, the expected finish time of both batches will be at most

$$\left(\mathbf{E}[\mathscr{F}_1] + O(1)\right) + \left(\mathbf{E}[\mathscr{F}_2] - l\right).$$

Note that, due to overlapping, the random finish times of both batches will not be independent. However, this does not effect the calculation of the overall expected finish time, as the expected value of sum of dependent random variables is still equal to the sum of their expected values.

Let $\mathcal{B}_{u,i}$ be the set of blocks of the first batch received by node u by the end of round $i \leq \mathscr{F}_1$, when there is no overlapping phase. Also, let $\mathcal{B}'_{u,i}$ be the set of the coded blocks of the first batch received by u by the end of round $i \leq \mathscr{F}_1$, when there is the overlapping phase. By induction, we show that $\mathcal{B}_{u,i} = \mathcal{B}'_{u,i}$ if u has not been interrupted by the end of round i. Clearly, the above statement holds for the first round (in fact, it holds for any round before the overlapping phase, that is rounds i, $1 \leq i \leq \mathscr{F}_1 - l$). Suppose that the induction hypothesis holds for some round i, $1 \leq i < \mathscr{F}_1$. We show that it also holds for round $i+1$. Let u be any node that has not been interrupted by the end of round $i+1$. Suppose u receives a block from v in round $i+1$. Since u was not interrupted in round $i+1$, node v must have not been interrupted by the end of round i, as, otherwise, node v will transmit a block of the second batch in round $i+1$ to v, and v becomes interrupted in round $i+1$. By induction hypothesis, $\mathcal{B}_{v,i} = \mathcal{B}'_{v,i}$, and $\mathcal{B}_{u,i} = \mathcal{B}'_{u,i}$. Therefore, we must have $\mathcal{B}_{u,i+1} = \mathcal{B}'_{u,i+1}$.

In the jth round of the overlapping phase, the number of interrupted nodes is at most $2^j - 1$. Since the overlapping phase has l rounds, the total number of interrupted nodes is at most $2^l - 1 < \sqrt{n}$. Therefore, by the end of the overlapping phase, the total number of nodes (including the source) that have a block of the second batch is at most $\sqrt{n}$. Note that a node u can be interrupted by a node v in a round i only if v has been interrupted in a round $j < i$.

Let $\mathcal{S}_i$ be the set of nodes that has been interrupted by the end of round $\mathscr{F}_1 - l + i$ (i.e., by the end of ith round of the overlapping phase). Let ζ_i be a random variable equal to the maximum number of times any single node has been interrupted by the end of round $\mathscr{F}_1 - l + i$. Therefore, every node $u \in \mathcal{S}_i$ has been interrupted at most ζ_i times by the end of round $\mathscr{F}_1 - l + i$. Note that $\zeta_1 = 1$, and $\zeta_{i+1} \leq \zeta_i + 1$. The number of nodes (including the source) that have at least a block of the second batch by the end of round $\mathscr{F}_1 - l + i$ is $|\mathcal{S}_i| + 1 \leq 2^i$. Therefore, by lemma 4, for $i \geq 1$, the probability that $\zeta_{i+1} = \zeta_i + 1$ is at most $\frac{(|\mathcal{S}_i|+1)^2}{n} = \frac{(2^i)^2}{n}$. Thus, we have

$$\begin{aligned}
\mathbf{E}[\zeta_l] &\leq 1 + \sum_{i=1}^{l-1} \frac{\left(|\mathcal{S}_i| + 1\right)^2}{n} \leq 1 + \sum_{i=1}^{l-1} \frac{\left(2^i\right)^2}{n} \\
&< \sum_{j=0}^{\infty} \left(\frac{1}{4}\right)^2 = 1 + \frac{1}{3}.
\end{aligned} \quad (4)$$

Let $\mathbf{M}_{i,j}^{B_1}$ denote the matrix corresponding to the coded blocks of the first batch that node u_i has received by the end of round j. For an uninterrupted node u_i, the matrix $\mathbf{M}_{i,\mathscr{F}_1}^{B_1}$ has full rank, that is, its rank is equal to k, where k is the number of uncoded blocks of the first batch. By (4), the *expected* rank of $\mathbf{M}_{i,\mathscr{F}_1}^{B_1}$ for a node u_i that has been interrupted is at least $k - \left(1 + \frac{1}{3}\right)$.

Let r_1 and r_2 be two consecutive rounds after the overlapping phase. Suppose $u \in \mathcal{S}_{\mathscr{F}_1}$, that is u is a node that has been interrupted in the overlapping phase. The maximum number of nodes that are ever interrupted is $\sqrt{n}$. Therefore, the probability that, in round r_1, u

receives a (coded) block of the first batch from a node v that has not been interrupted before is at least $1-\frac{1}{\sqrt{n}}$. Thus, the probability that, in either round r_1 or r_2, u receives a (coded) block of the first batch from a node v that has not been interrupted before is at least

$$1-(\frac{1}{\sqrt{n}})^2 = 1-\frac{1}{n}.$$

Therefore, assuming that the size of finite field $q \geq n$, by a union bound, the probability that rank of u's matrix is incremented (if it is not full rank) in either round r_1 is at least

$$1-\left(\frac{1}{n}+\frac{1}{n}\right) = 1-\frac{2}{n}.$$

Hence, by a union bound, the probability that the rank of non-full-rank matrices of nodes $u \in \mathcal{S}_{\mathscr{F}_1}$ is incremented in either round r_1 or r_2 is at least

$$1-\sqrt{n}\cdot\frac{2}{n} = 1-\frac{2}{\sqrt{n}}.$$

Therefore, on average, in at most $\frac{2}{1-\frac{2}{\sqrt{n}}}$ rounds after the overlapping phase, the rank of all non-full-rank matrices is incremented. Let γ be a random variable equal to the number of rounds required after the overlapping phase to have all matrices full rank. Since the minimum rank of any matrix after the overlapping phase is $k-\zeta_m$, we get

$$\mathbf{E}[\gamma|\zeta_l=\zeta] \leq \frac{2}{1-\frac{2}{\sqrt{n}}}\zeta$$

Thus, the expected number of rounds needed after the overlapping phase such that every matrix becomes full rank is at most

$$\begin{aligned}\mathbf{E}[\gamma] &= \mathbf{E}_\zeta[\mathbf{E}[\gamma|\zeta_l=\zeta]] \leq \mathbf{E}[\frac{2}{1-\frac{2}{\sqrt{n}}}\zeta] = \frac{2}{1-\frac{2}{\sqrt{n}}}\mathbf{E}[\zeta] \\ &\leq \frac{2}{1-\frac{2}{\sqrt{n}}}\left(1+\frac{1}{3}\right) \leq \frac{2.7}{1-\frac{2}{\sqrt{n}}}\end{aligned}$$

□

Proof Sketch of Corollary 2: Without loss of generality, we assume that there are three batches. We can argue similarly when the number of batches is more than three. Suppose that the second batch finishes at least l rounds after the finish time of the first batch. In this case, the third batch starts l rounds before the finish time of the second batch, and that would be after the finish time of the first batch. Therefore, the transmission processes of the first and third batches will not overlap. In this case, the proof that we can save another $\Theta(\log n)$ rounds by overlapping transmissions of the second and third batches will be identical to the proof of Theorem 1.

The second case is that the finish time of the first batch is more than the finish time of the second batch minus l. Since the broadcast of every batch requires at least $\log n \geq 2m$ rounds, and the length of the overlapping phase (during which the second batch can progress) is l rounds, the second case happens only when the second batch has progressed in the "recovery phase" of the first batch, that is in the rounds after the overlapping phase, and before the finish time of the first batch. In the proof of Theorem 1, we have implicitly assumed the worst case in which the second batch does not even progress during the "recovery phase" of the first batch. This is a worst-case scenario as any progress of the second batch in the "recovery phase" of the first batch will increase the overall savings. Here, we can also assume the worst case (i.e., no progress of the second batch during the "recovery phase" of the first batch) which implies that the second case does not occur. In other words, since the second case is not worse than the first case, we can only consider the first case in the analysis, which is a direct result of Theorem 1.