Title: Network Coding in Live Peer-to-Peer Streaming

Authors: Mea Wang, Baochun Li
Affiliation: Department of Electrical and Computer Engineering, University of Toronto
Address: 10 King's College Road, Toronto, Ontario M5S 3G4, Canada
Email: mea@eecg.toronto.edu, bli@eecg.toronto.edu
Contacting author: Baochun Li
Phone: (416) 946-7338
Fax: (416) 978-4425

Network Coding in Live Peer-to-Peer Streaming

Mea Wang, Baochun Li, *Senior Member, IEEE*

***Abstract*— In recent literature, network coding has emerged as a promising information theoretic approach to improve the performance of both peer-to-peer and wireless networks. It has been widely accepted and acknowledged that network coding can theoretically improve network throughput of multicast sessions in directed acyclic graphs, achieving their cut-set capacity bounds. Recent studies have also supported the claim that network coding is beneficial for large-scale peer-to-peer content distribution, as it solves the problem of locating the last missing blocks to complete the download.**

We seek to perform a reality check of using network coding for *peer-to-peer live multimedia streaming*. We start with the following critical question: *How helpful is network coding in peer-to-peer streaming?* To address this question, we first implement the decoding process using Gauss-Jordan elimination, such that it can be performed *while* coded blocks are progressively received. We then implement a realistic testbed, called *Lava*, with actual network traffic to meticulously evaluate the benefits and trade-offs involved in using network coding in peer-to-peer streaming. We present the architectural design challenges in implementing network coding for the purpose of streaming, along with a pull-based peer-to-peer live streaming protocol in our comparison studies. Our experimental results show that network coding makes it possible to perform streaming with a *finer granularity*, which reduces the redundancy of bandwidth usage, improves resilience to network dynamics, and is most instrumental when the bandwidth supply barely meets the streaming demand.

I. Introduction

Network coding has been originally proposed in information theory [2], [3], [4], and has since emerged as one of the most promising information theoretic approaches to improve performance in peer-to-peer and wireless networks. The upshot of network coding is to allow coding at intermediate nodes in a directed network, assuming that links are error-free. The assumption of error-free links is made to avoid the most perplexing challenges of *interference* in the field of network information theory. It has been shown that *random linear codes* using a Galois field of a limited size are sufficient to implement network coding in a practical network setting. In some cases, even exclusive-ORs — GF(2) — can improve throughput in wireless mesh networks [5].

Avalanche [6], [7] has demonstrated — using both simulation studies and realistic experiments — that network coding may improve the overall performance of peer-to-peer content distribution, up to about a hundred peers. The intuition that supports such a claim is that, with network coding, all blocks are treated equally, without the need to distribute the "rarest block" first, or to find them in the "end game" of the downloading process. While these are noteworthy observations, we note that content distribution applications deal with *elastic* traffic: one wishes to minimize downloading times, but there are no required lower bounds with respect to the instantaneous rate of a live session.

The requirements of peer-to-peer live multimedia streaming applications, however, have marked a significant departure from traditional applications of elastic content distribution. The most critical requirement is that the *streaming rate* has to be maintained for smooth playback. Each live streaming session may involve a live media stream with a specific streaming rate, such as 800 Kbps for a typical Standard-Definition stream, generated with a modern codec such as H.264. The challenge of streaming is that the demand for bandwidth at the streaming rate (which is very similar to CBR traffic) must be satisfied at all peers, while additional bandwidth is, in general, not required.

Existing successes with peer-to-peer streaming, such as CoolStreaming [8] and PPLive, have demonstrated that peer-to-peer live streaming is not only feasible, but also practical at a large scale. Observing the recent success of using network coding in wireless mesh networks [5] and peer-to-peer content distribution [6], it is natural to ask the following interesting question: *Does network coding help in peer-to-peer live streaming?* In this paper, we endeavor to explore the benefits and trade-offs of applying network coding in peer-to-peer live streaming, *using an experimental testbed* with real traffic and a highly optimized implementation of network coding. To implement the decoding process at each peer with the highest performance possible, we propose to use Gauss-Jordan elimination (rather than the usual Gaussian elimination), which can be performed concurrently *while* coded blocks are progressively received. To further optimize our implementation, we have implemented network coding with full acceleration using the x86 SSE2 instruction set.

In building our experimental testbed, henceforth referred to as *Lava*, in a cluster of 44 dual-CPU servers, we have to address a number of significant design and implementation challenges. *First,* as live traffic is involved in all our experiments, large volumes of live TCP connections and UDP traffic need to be efficiently managed. *Second*, all experiments need to be both realistic and controllable, with upload capacities on each peer accurately emulated. *Third*, since we emulate a number of peers in each cluster node, we wish to minimize the processing and memory footprint of our implementation. *Finally,* to qualify as a "reality check," we wish to implement peer joins and departures following specific probability distributions. This brings all the challenges when *dynamics* are considered, including handling broken and orphaned network connections, exchanging availability, and maintaining updated

Both authors are affiliated with the Department of Electrical and Computer Engineering at the University of Toronto, Toronto, Ontario, Canada. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and Bell Canada through its Bell University Laboratories R&D program. A preliminary version of this work is to appear in Proc. IEEE INFOCOM 2007 [1].

peer lists.

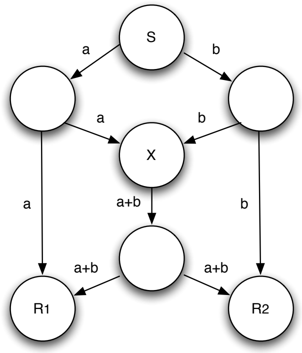
In order to compare network coding with a standard peer-to-peer live streaming protocol without coding, we have implemented a pull-based P2P live streaming protocol (henceforth codenamed *Vanilla*), which is typically used in real-world streaming applications. As an intentional design decision to guarantee fairness in our comparison studies, network coding is implemented as a *plugin* component in Vanilla, such that both share identical protocols, configuration parameters, and design choices. With our testbed, we strive to faithfully report our results from a selected set of experiments and from an unbiased point of view, as well as our empirical observations and insights from hands-on experiences of analyzing logs from a large number of experiments. Our experimental results show that network coding makes it possible to perform streaming with a *finer granularity*, which reduces the redundancy of bandwidth usage, improves resilience to network dynamics, and is most instrumental when the bandwidth supply barely meets the streaming demand.

The remainder of this paper is organized as follows. In Sec. II, we discuss related work in practical network coding. Sec. III presents the architectural design challenges in building our experimental testbed, as well as design choices we have made in our implementation. In Sec. IV, we show results from our experiences in comparing network coding with Vanilla in peer-to-peer live streaming sessions. We conclude the paper in Sec. V.

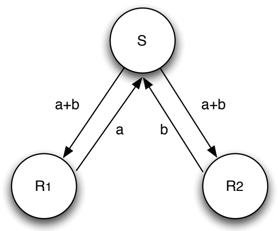
II. Related Work

The benefit of network coding with respect to improving throughput in directed graphs can be best illustrated in the "Butterfly" example, as shown in Fig. 1(a). In this example, each link in the topology has unit capacity, and the source S seeks to maximize its session throughput to both R_1 and R_2 in a multicast session. Without network coding, optimal throughput can be achieved by solving the problem of *steiner tree packing*, which leads to a flow rate of 1.875. With network coding, as shown in Fig. 1(a), node X is able to code its input messages a and b into $a + b$, with the $+$ operation defined in a finite field, leading to an effective end-to-end throughput of 2. In wireless networking scenarios requiring wireless information exchange, network coding is also shown to be helpful [5], [9]. In Fig. 1(b), if R_1 sends a to R_2, while R_2 sends b to R_1, it requires four units (*e.g.*, time slots) of transmission without coding. With network coding and the wireless broadcast advantage, the intermediate node S may simply broadcast $a + b$ to both R_1 and R_2, making it feasible to complete the exchange in three units.

Since the landmark paper on randomized network coding by Ho *et al.* [10], [11], there has been a gradual shift in research focus in the area of network coding, from theoretical studies on achievable flow rates and code assignment algorithms [12], to more practical studies on applying network coding in a practical setting. Such a shift of focus has been marked by the work of Chou *et al.* [13], which concludes that randomized network coding can be designed to be robust to random packet loss, delay, as well as any changes in network topology and capacity. Avalanche [6], [7] has further proposed that randomized



(a) The *Butterfly* example.



(b) The *wireless exchange* example.

Fig. 1. Network coding improves session throughput: well-known examples.

network coding can be used for elastic content distribution. It has been shown that the performance benefits provided by network coding in terms of throughput can be more than 2-3 times better compared to not using network coding at all. In this sense, one concludes that network coding can indeed be practically implemented, and does offer significant advantages in peer-to-peer bulk content distribution.

Our recent work [14] has investigated the practicality of randomized network coding, from the point of view of *coding complexity* and real-world coding performance in peer-to-peer content distribution. We have shown that the performance of network coding is acceptable when one uses a small number of blocks, in the order of less than a thousand. We seek to continue to explore the practicality of network coding in this paper, but in the setting of peer-to-peer live streaming, rather than bulk content distribution.

To the best of our knowledge, there has been no existing work that has systematically studied the practicality of using network coding in peer-to-peer live streaming applications, especially using a realistic testbed that involves actual network traffic and peer dynamics.

III. Lava: Experimental Testbed of Network Coding in P2P Live Streaming

The complexity and challenges of practical network coding have not been previously examined in the context of peer-to-peer live streaming. There is no doubt that an *experimental testbed* needs to be implemented to form an unbiased and meticulous evaluation of network coding in live streaming sessions. The design and implementation of such a testbed, however, proved to be both inspiring and demanding.

Ideally, the best route to evaluate a protocol may be to actually implement and deploy it across the Internet, in realistic peers with home broadband connections. We note that such an implementation, while realistic, may not be able to offer sufficiently scientific evidence from which conclusions are drawn. *First*, due to the highly dynamic nature of peers in P2P applications, experimental results in such a realistic system may not be reproduced and analyzed after parameters or designs are tuned and optimized. *Second*, existing experimental testbeds PlanetLab [15], Netbed [16], and ModelNet [17] cannot emulate certain properties of the real peer-to-peer networks, including peer dynamics, various connection types, and link delays. *Finally*, CPU and bandwidth on each peer — the most important resources that lead to the advantage of the P2P architecture — is heterogeneous and highly dynamic, as the availability of CPU cycles and bandwidth in PlanetLab are subject to the fluctuating load of concurrent tasks on the same host. For these reasons, we preclude the real-world deployment.

We insist that our experiments should not only be controllable, repeatable and configurable, but also involve a large percentage of peers with DSL Internet connections. The most accurate results are achieved by using a dedicated cluster of high-performance servers, interconnected by Gigabit Ethernet, and by correctly *emulating* upload bandwidth capacities on each peer at the application layer. Fortunately, a cluster of 44 dedicated dual-CPU servers (Pentium 4 Xeon 3.6 GHz and AMD Opteron 2.4 GHz) is at our disposal for our experiments.

To make more convincing and conclusive observations, our testbed must address the following design challenges:

- ▷ Actual network traffic needs to be involved to emulate practical streaming sessions. Hence, a potentially large number of TCP connections and UDP flows need to be efficiently managed by each peer.
- ▷ To avoid miscalculations and incorrect conclusions due to an inferior implementation of randomized network coding, a highly optimized implementation with maximum performance is a must, with attention to details.
- ▷ Network coding should be evaluated in the same context as a conventional peer-to-peer streaming protocol, with identical parameter settings and protocol design.
- ▷ Peer arrivals and departures in a particular session need to be emulated, which leads to the challenges of maintaining up-to-date peer lists and handling dynamic network connections.
- ▷ Since we wish to maximize the number of peers to be emulated on each cluster server, the processing and memory footprint of our implementation must be minimized.

In this section, we present the design choices that we have made in our testbed, *Lava*, from both the *architectural* and *algorithmic* point of views.

A. Lava: Architecture

On each peer in a peer-to-peer live streaming session, the architectural design of our experimental testbed is best illustrated in Fig. 2. The core of the Lava architecture is referred to as the *algorithm*, which includes both the standard pull-based peer-to-peer streaming protocol, and the encoding/decoding processes of randomized network coding. To feed the algorithm, the architecture allows multiple live TCP connections from multiple upstream peers. To transmit the outcomes of the algorithm (*i.e.*, results of the encoding process), multiple TCP connections to their corresponding downstream peers are established. During the lifetime of each neighboring peer, a persistent TCP connection is established to minimize overhead. A live session in Lava contains one multimedia stream with a specific streaming rate. Such a live stream is divided into *segments*, each of which has a specific duration

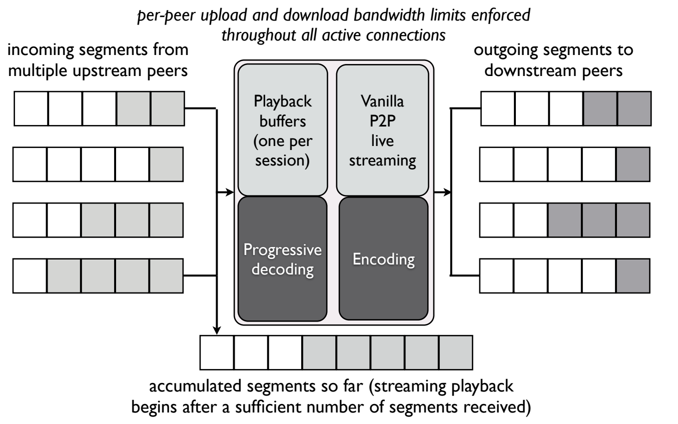

per-peer upload and download bandwidth limits enforced throughout all active connections
incoming segments from multiple upstream peers
outgoing segments to downstream peers
Playback buffers (one per session)
Vanilla P2P live streaming
Progressive decoding
Encoding
accumulated segments so far (streaming playback begins after a sufficient number of segments received)

Fig. 2. The architecture of a bandwidth-emulated peer in Lava.

(one second in our experiments). If network coding is used, each segment is further divided into blocks.

With scalability as one of the design goals, the entire Lava implementation on each peer consists of only two threads. The *network* thread has the following responsibilities: (1) It maintains all the incoming and outgoing TCP connections and UDP traffic, as well as their corresponding FIFO queues; (2) It is capable of generating data sources for a streaming session, and managing the session during its lifetime; and (3) It emulates the upload and download capacities on each peer, as well as capacities and delays on specific overlay links. To manage all TCP and UDP traffic in a single thread, they are monitored by a single `select()` call with a specific timeout value. The timeout value of the `select()` call is dynamically tuned according to the traffic volume and bandwidth settings. Such timeout-based `select()` calls are also critical for the network thread to emulate overlay link delays and bandwidth limits.

The *algorithm* thread implements the actual algorithms and protocols to support peer-to-peer live streaming, including the following duties: (1) It processes the head-of-line messages from incoming connections, and sends produced streaming segments from the algorithm to outgoing connections; (2) It maintains a local buffer that stores data segments that have been received so far, and emulates the playback of each segment; (3) It supports multiple event-driven asynchronous timeout mechanisms with different timeout periods, so that asynchronous reoccurring or one-time events can be scheduled as their respective times; and (4) It implements *Vanilla*, a standard pull-based peer-to-peer live streaming protocol, as well as a plugin component that performs randomized network coding within Vanilla. With respect to its playback buffers, the algorithm thread forms natural producer-consumer relationships with the network thread. We note that the asynchronous timeout mechanisms are implemented without any CPU usage, which is important to schedule a large number of future events (such as peer departures) at each peer.

There are no limitations in the Lava implementation that preclude running more than one peer on each server. Unlike real-world applications, they do not need to periodically contact a central server for logistics or authentication. All logs are written to local file systems, then collected and analyzed by dedicated Perl scripts after the experiments. To control all the events in an experiment, we have implemented a log-driven facility in Lava. There are two logs: *events* and *topologies*. The *events* log specifies one event per line, including the time that the event should occur in the experiment, the event type, and optional parameters associated with the event. Typical events include the beginning and end times of a session, as well as peer arrivals and departures within a session. For example, to define a session deployment event, the event in the log specifies the time, the streaming source, and the streaming rate. The *topologies* log is used to *bootstrap* peers when they first join a network. For each peer, it includes a small number of existing peers. The use of such a log-driven facility further relaxes the necessity to contact centralized servers, which may lead to a considerable amount of TCP traffic that affects the precision of the experiments..

Finally, to guarantee the most optimized binary, Lava is implemented in approximately $11,000$ lines of code in C++, and compiled with full optimization (`-O3`). The implementation of network coding is further accelerated using the x86 SSE2 instruction set. Though all our experiments are performed in our server cluster running Linux, it can be readily compiled on other UNIX variants as well.

B. Lava: Streaming with Vanilla

In order to evaluate the benefits and tradeoffs of network coding in a typical streaming system, we implemented *Vanilla*, a standard peer-to-peer streaming protocol. Our objective with Vanilla is to implement the best possible pull-based data-driven P2P live streaming protocol, achieving performance that matches real-world protocols (*e.g.,* CoolStreaming [8] and PPLive). This is necessary since we do not wish to make incorrect conclusions simply due to inferior performance when network coding is not used. As in any typical pull-based P2P streaming protocol, peers in Vanilla periodically exchange information on segment availability, which is sometimes referred to as *buffer maps* in previous literature. According to the buffer maps, those peers that have a particular segment available for transmission are referred to as the *seeds* of this segment.

For each streaming session, a peer maintains a *playback buffer* in which segments are ordered according to their playback deadlines. Segments in the buffer are grouped into *batches*, and each batch consists of a set of consecutive segments. A segment is removed from the buffer after being played. Similar to real-world streaming systems, a peer does not immediate start playback as the first data segments are received. Instead, it waits for a period of *initial buffering delay* to ensure smooth playback. During such an initial buffering process, the download capacity of the new peer is usually saturated, and the playback buffer fills up rapidly. The playback buffer is initialized to start from the segment that is scheduled for playback immediately after the initial buffering delay. To ensure that early segments receive higher priority in transmission, the peer begins with the first batch in the buffer, and moves to the next batch only if all segments in the previous batch have been scheduled. Segments from the same batch are requested in an arbitrary order, to promote bandwidth sharing among peers.

During smooth playback, all segments should be readily available before its playback deadline. For each segment that is due for playback, the peer schedule a new segment for transmission by sending a request to an arbitrarily selected seed. In the case of network coding, since segments are further divided into blocks and then coded, the peer downloads coded blocks from multiple seeds. If a requested segment is not received within the expected time — the per-segment timeout period, Vanilla requests the segment again (possibly from a different seed). In the event of peer departures, Vanilla retransmits the affected segments from other seeds.

In the unfortunate event that a segment is not successfully received in time, Vanilla *skips* the segment. The number of *playback skips* is a good indicating factor to quantitatively evaluate the quality of the streaming playback. To minimize playback skips and to fully utilize download capacity, we introduce the low and standard buffering watermarks, which mark the alert mode and normal playback mode, respectively. After the initial buffering delay, a peer enters the alert mode if there are missing segments before the low buffering watermark. In this case, it retransmits the missing segments from other seeds, and requests one additional segment for each missing segment when scheduling the next segment for transmission. Naturally, the peer returns to its normal mode after it reaches the low buffering watermark again. In the event of peer departures, Vanilla retransmits the affected segments from other seeds. Fig. 3 visually illustrates the playback buffer in Vanilla and its playback modes. The playback buffer is internally implemented as a circular queue in Vanilla for efficiency.

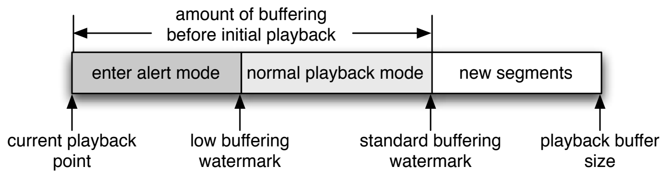


Fig. 3. The playback buffer in Vanilla.

Assuming each segment represents 1 second of playback, *i.e.*, a peer requests a new segment every second. In the worst case, when all segments in the alert zone are incomplete at time τ, two segments is scheduled for transmission every second. There will be at most $2*t$ outstanding requests at time $\tau+t$, where t is the number segments in the alert zone. At this point, if none of the segments in the alert zone is completely received in the next t seconds, the peer will have $3*t$ segments in transmission. If such scenario continues, the number of outstanding requests increases at the rate of 1 per second. Vanilla limits the buffer size to impose an upper bound for this number and to avoid excessive number of connections from seeds. Furthermore, to avoid overloading the seeds, Vanilla also limits the number of concurrent TCP connections on each peer. With appropriate settings of the buffer size and initial buffering delay, as verified in Sec. IV, the downloading rate of a peer is seldom slower than the streaming rate. Hence, the number of consecutively available segments in the buffer is always more than that buffered during the initial buffering delay.

Perfect collaboration

Network coding (both encoding and decoding processes) is implemented as a plugin component, such that it shares identical protocol design and parameter settings with Vanilla. It is intuitive to conceive at least one important benefit of using network coding in live P2P streaming: a peer may download coded blocks of a segment from multiple seeds at the same time, without requiring any protocol to coordinate the efforts of these seeds. Each seed simply starts to serve coded blocks of the segment upon receiving a request. When a peer completely decode the segment, it informs all its seeds by sending a short message to each of them to terminate the transmission of this segment. Without using coding, a peer will have to explicitly requesting particular blocks from a particular seed, and perform complex computations to estimate bandwidth availability from the seed, in order to determine the number of outstanding requests to the seed. In other words, we can easily achieve *perfect collaboration* among the seeds when serving a particular peer using network coding.

C. Lava: Progressive network coding

Aggressiveness and density in randomized network coding

We first briefly summarize the concept of randomized network coding [6], [10], [11], [13]. With randomized network coding [6], [10], [11], [13], each segment in a live stream is further divided into n *blocks* $[b_1, b_2, \ldots, b_n]$, where b_i has a fixed number of bytes k (referred to as the block size). If the number of playback seconds l represented by a segment and the streaming rate r are pre-defined, the block size $k = (r \cdot l)/n$. When encoding a new block for a downstream peer p, the peer (including the streaming source) first independently and randomly chooses a set of coding coefficients $[c_1^p, c_2^p, \cdots, c_n^p]$ in the Galois field GF(256), one for each received block (original blocks on the source) in the segment. The ratio of none-zero entries in the set of coding coefficients $d(0 < d \leq 1)$ is referred to as the *density*. It then produces one coded block x of k bytes:

$$x = \sum_{i=1}^{n} c_i^p \cdot b_i \tag{1}$$

As the session proceeds, a peer accumulates coded blocks from its seeds into the local buffer, and encodes new coded blocks to serve its downstream peers. In order to reduce the delay introduced by waiting for new coded blocks, the peer starts producing and serving new coded blocks after $a \cdot n$ $(0 < a \leq 1)$ coded blocks has been received, where a is referred to as the *aggressiveness*. A smaller a leads to a shorter waiting time and, potentially, shorter delay at downstream peers. In other words, the peer is more "aggressive."

Since each coded block is a linear combination of the original blocks, it can be uniquely identified by the set of coefficients that appeared in the linear combination. The coefficients of x can easily be computed using Eq. (1) by replacing incoming blocks b_i with the coefficients of b_i. In our implementation, a coded block x is *self-contained*, *i.e.*, the coefficients are embedded in the header of the coded block, leading to a header overhead of n bytes per coded block. When both aggressiveness a and density d are 1, the seed used to initialize a random number generator is embedded to reduce the overhead. Upon receiving a block, a peer can reproduce the coefficients using the seed.

TABLE I
MULTIPLICATION OF TWO BYTES ON GF(256) IN LAVA

```
int gf_single_multiply(int x, int y)
{
  if (x == 0 || y == 0) return 0;
  return J_TO_B[B_TO_J[x] + B_TO_J[y]];
}
```

A peer decodes a segment as soon as it has received n linearly independent coded blocks $\mathbf{x} = [x_1, x_2, \ldots, x_n]$. It first forms a $n \times n$ matrix $\mathbf{A}$, using the embedded coefficients of each block x_i. Each row in $\mathbf{A}$ corresponds to the coefficients of one coded block. It then recovers the original blocks $\mathbf{b} = [b_1, b_2, \ldots, b_n]$ as:

$$\mathbf{b} = \mathbf{A}^{-1}\mathbf{x}^T \tag{2}$$

In this equation, it first needs to compute the inverse of $\mathbf{A}$, using Gaussian elimination. It then needs to multiply $\mathbf{A}^{-1}$ and $\mathbf{x}^T$, which takes $n^2 \cdot k$ multiplications of two bytes in GF(256). The inversion of $\mathbf{A}$ is only possible when its rows are linearly independent, *i.e.,* $\mathbf{A}$ is full rank. Since addition in GF(256) is simply an XOR operation, if we wish to optimize the performance of coding, it is important to optimize the implementation of multiplication on GF(256). If x86 SSE2 acceleration is not used, this operation requires three memory reads and one addition for each coded byte, as shown in Table I.

Progressive decoding using Gauss-Jordan elimination

We note that a peer does not have to wait for all n linearly independent coded blocks before decoding a segment. In fact, it can start to decode as soon as the first coded block is received, and then *progressively* decodes each of the new coded blocks, as they are received over the network. In this process, the decoding time overlaps with the time required to receive the original block, and thus hidden from the tally of overhead caused by encoding and decoding times.

To realize such a progressive decoding process, we employ *Gauss-Jordan elimination*, rather than Gaussian elimination, in the decoding process. Gauss-Jordan elimination is a variant of Gaussian elimination, that transforms a matrix to its *reduced row-echelon form* (RREF), in which each row contains only zeros until the first nonzero element, which must be 1. As each new coded block is received, its coefficients are added to the coefficient matrix $\mathbf{A}$ of the corresponding segment. A pass of Gauss-Jordan elimination is performed on this matrix, with identical operations performed on the data portion of the blocks. At the end of this process, the data portion of each block becomes the original block.

In this progressive decoding process, once the matrix is reduced to an identity matrix, the result vector on the right of the equation constitutes the solution, without any additional needs of decoding. Moreover, if a peer received a coded block that is linearly dependent with existing blocks of the corresponding segment, the Gauss-Jordan elimination process will lead to a row of all zeros, in which case this coded block can be immediately discarded. No explicit linear dependence checks are required either during or at the end of the transmission of a segment, and the matrix $\mathbf{A}$ in Eqn. 2 is always full rank at the end of the process. Though Gauss-Jordan elimination usually leads to numerical instability, it does not affect network coding since we operate in the Galois field.

Architectural design

The architecture of the network coding implementation in Lava is summarized in Fig. 4.

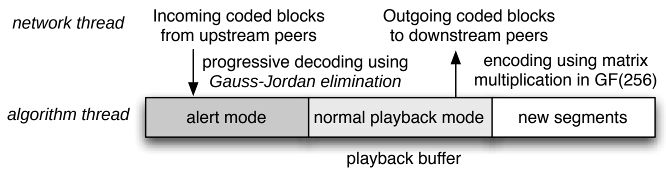

network thread
Incoming coded blocks from upstream peers
Outgoing coded blocks to downstream peers
progressive decoding using *Gauss-Jordan elimination*
encoding using matrix multiplication in GF(256)
algorithm thread
alert mode
normal playback mode
new segments
playback buffer

Fig. 4. The architectural design of network coding in Lava.

Every time a new coded block is received into a segment in the playback buffer, a peer performs progressive decoding by applying Gauss-Jordan elimination to all received blocks of this segment. As the decoding process progresses, intermediate outcomes of Gauss-Jordan elimination are stored in the playback buffer, until the entire segment is completely decoded.

Finally, an important note we wish to make is that, depending on the settings of aggressiveness, the encoding process uses either the intermediate or fully decoded blocks from the playback buffer, and progresses *concurrently* with the decoding process. This guarantees that the encoding progress may start before the decoding process is complete, such that peers are more "aggressive." We tune the number of blocks per segment so that the encoding process is much faster than the upload link capacity, and would not become the bottleneck of the streaming session.

SSE2 Acceleration

To further improve the performance of our network coding implementation on each peer, we have implemented an accelerated framework in both the encoding and progressive decoding process using x86 SSE2 instructions. To achieve this objective, we need to implement the basic GF(2^8) operations in Rijndael's finite field, employing the reducing polynomial $x^8 + x^4 + x^3 + x + 1$ for multiplication, leading to algorithm that performs a combination of bit rotations and exclusive-ORs in a loop (with 8 iterations) to produce the product of two bytes. The reason for using this algorithm is that it is relatively easy to perform "batch processing" of in special 128-bit SSE registers in x86 CPUs, making use of the SSE2 instructions to operate on these registers. We present details of our accelerated implementation in a separate paper.

When compared with a baseline implementation using C (but without SSE2 acceleration), the performance gain of employing SSE2-accelerated network coding is between 300% and 500%, depending on specific values of various parameters. We are satisfied with such results, since it paves the way towards a level ground for comparing Vanilla with the use of network coding.

IV. Evaluation of Network Coding in P2P Streaming

With Lava, we are now ready to perform an empirical "reality check" of network coding in peer-to-peer live streaming. The focus of our study is on the practicality, performance and overhead of randomized network coding, as compared to Vanilla, a standard peer-to-peer streaming protocol without using network coding. The ultimate objective of this study is to answer the question: *Should we implement network coding in peer-to-peer live streaming?*

We deploy streaming sessions in a server cluster of 44 dual-CPU cluster node. In all experiments, the uplink bandwidth on the dedicated streaming server of the session is constrained to 1 MB per second. In reality, a peer-to-peer network usually consists certain number of peers with upload capacity more than 1 MB per second. For a more challenging scenario, we emulate all peer connections as DSL uplinks, uniformly distributed between 80 and 100 KB per second. We use a streaming bit rate of 64 KB per second, which must be satisfied at all peers during their streaming playback. In our experiments, unless otherwise specified, each segment represents 1 second of playback, and is divided into 32 blocks, offering a satisfactory encoding and decoding bandwidth with our implementation of randomized network coding (around 19 MB/second). We set the buffer size to 30 seconds, the initial buffering delay to 20 seconds, and the batch size to 10 seconds. Each streaming session lasts for 10 minutes.

To compare the performance, we evaluate several important metrics: (1) *Playback skips*: measured as the percentage of segments skipped during playback. A segment is skipped during playback if it is still not completely received at the time of playback. (2) *Bandwidth redundancy:* to evaluate the level of redundancy in bandwidth usage, we measured the percentage of discarded segments (or blocks in network coding), due to obsolescence (or linear dependence, over all received segments (or blocks). (3) *Buffering levels* on each peer during a live session over time, measured as the percentage of completely received segments in the playback buffer. For all measurements, we take the average from all peers in the network.

A. *Performance of network coding*

The first important question we wish to ask is about the baseline performance of network coding. What is the raw performance of network coding, with and without our progressive decoding implementation using Gauss-Jordan elimination? With Lava, we establish a single streaming connection between one source and one receiver peer, each hosted by a dedicated server, interconnected by Gigabit Ethernet, without imposing bandwidth limits. We test network coding with live streams with an average duration of 125 seconds. In tuning network coding, we set for both density and aggressiveness to 1, since we wish to evaluate the raw coding performance. We vary the block size from 128 bytes to 256 KB, and show the average of measurements from encoding all 125 segments in the stream. For each block size, we increase the streaming rate until the CPU is 100% saturated to find the maximum sustainable streaming rate.

In Fig. 5(a), we show our results of evaluating the coding performance, with respect to the encoding and decoding bandwidth, as well as the maximum sustainable streaming rate. From these results, we have observed that, thanks to our SSE2 accelerated implementation, the absolute coding performance is quite impressive, especially when there are fewer blocks. When there are only 32 blocks, the encoding bandwidth exceeds 15 MB per second on one CPU! On the flip side, we have also observed that both encoding bandwidth and decoding bandwidth rapidly decrease as the number of blocks per segment linearly increases. We have also shown that network coding can support a wide range of streaming rates, from 100 KB per second to more than 8 MB per second, which are more than sufficient to accommodate typical streaming rates in the real-world P2P streaming.

With SSE2-accelerated network coding operations, we have observed that the decoding bandwidth decreases faster than the encoding bandwidth as the number of blocks increases. This is mostly due to the fact that the computational overhead of Gauss-Jordan elimination may not be as easily accelerated with SSE2 as straightforward vector multiplications. This phenomenon also makes the decoding process the bottleneck of network coding in the streaming process. As indicated in Fig. 5(a), the maximum sustainable streaming rate is limited by the decoding bandwidth.

To illustrate the advantage of progressive decoding using Gauss-Jordan elimination, we modified the algorithm to decode blocks only after all blocks of a segment has been received, and then run the same experiment again with progressive decoding. In this experiment, we measured the time required to completely receive a segment ("transmission time"), and the time spent by the decoding process to recover the original blocks after all blocks have been received ("recovery time"). The transmission time includes the encoding time on the source.

In Fig. 5(b), the bar on the left of each setting represents the results from using conventional decoding, and the bar on the right corresponds to progressive decoding. We note that the recovery time of conventional decoding is longer than the transmission time in most cases. In fact, the conventional decoding process consumes a remarkable amount of CPU such that most of the segments can not be played according to their deadlines. Progressive decoding significantly reduces the time required to completely receive and recover a segment, with one exception. In the $(128\text{B}, 784)$ setting, progressive decoding has a longer transmission time because the computational overhead of Gauss-Jordan elimination dominates the transmission time with a large number of blocks. In general, the decoding time spent after the last coded block is received is negligible. With progressive decoding, decoding times are almost *completely* concealed within the time required to receive the segment.

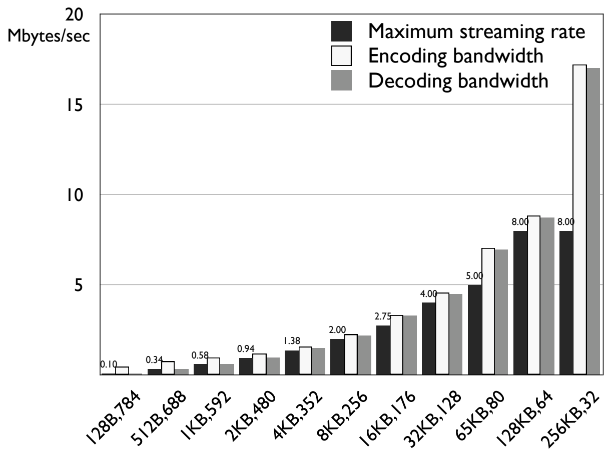
While we have previously shown the performance of network coding when CPU is intentionally 100% saturated, we would like to show the "flip side of the coin" as well. For this purpose, we deploy a streaming session with a typical streaming rate of 64 KB per second in between two peers (real-world P2P applications, such as PPLive, usually use streaming bit rates of less than 50 KB per second). Fig. 6(a) and Fig. 6(b) show that such a typical streaming rate is sustainable regardless of the number of blocks in each segment, with negligible decoding times after a segment is completely receiver. Empirically, we have also observed (not explicitly shown in the figure) that the CPU usage increases from 4% to 15% as the number of blocks increases. Beyond benefits of negligible CPU usage, the other advantage of using a smaller number of blocks, 32 in this case, is that the overhead for carrying the coding coefficients in each coded block is smaller. For instance, the coding coefficient overhead is 100% when using 256 blocks, where as it is only 1.5% when using 32 blocks.

B. Scalability

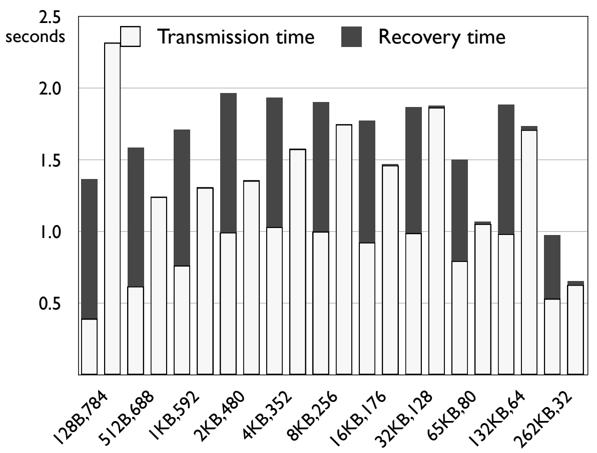
We now compare Vanilla and network coding when the number of peers in the network scales up. In this experiment, we add one peer on each server at a time, until all 44 servers are fully saturated. A 64 KB per second streaming session is deployed in the network for 10 minutes. Fig. 7(a) shows that, in networks consist of 572 peers or less, network coding is approximately as scalable as Vanilla in terms of playback quality. The CPU usage of both algorithms grows linearly with respect to the network size. Due to computational complexity, when the network size reaches 616 (13 peers on each server), network coding consumes more than 85% of on each CPU, and its performance degrades significantly. Nevertheless, we argue that such a limitation is not applicable in reality, since each peer runs only one coding instance, which consumes less than 10% of the CPU. Fig. 7(a) shows that the bandwidth redundancy introduced by Vanilla is approximately 20% of the total network traffic, in networks with more than 264 peers. Hence, network coding is more scalable in term of bandwidth redundancy.

As an important metric to gain insights on the playback quality, we examined the fluctuations in average buffering levels over the course of a streaming session. Fig. 7(b) compares the average buffering levels of network coding and Vanilla in three representative networks, from which we drew two key observations. First, the buffering level ramps up quickly and remains stable when using network coding, while Vanilla maintains a much lower level with a slight variation over time. Second, the buffering levels of both algorithms decrease as the network size increases; however, the change in the case of network coding is not significant.

It is interesting to note that network coding does not offer better playback quality than Vanilla, while maintaining a remarkably high and stable buffering level, regardless the network size. To trace the cause of such phenomenon, we

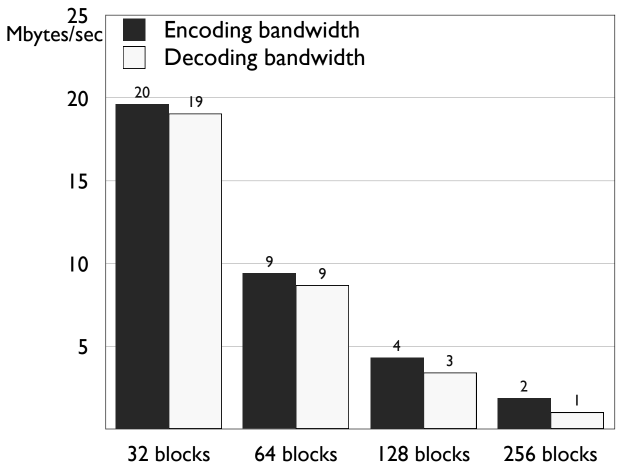


(a) The encoding bandwidth and maximum sustained streaming rates with different numbers of blocks. We have attempted streaming rates up to 8 MB per second.

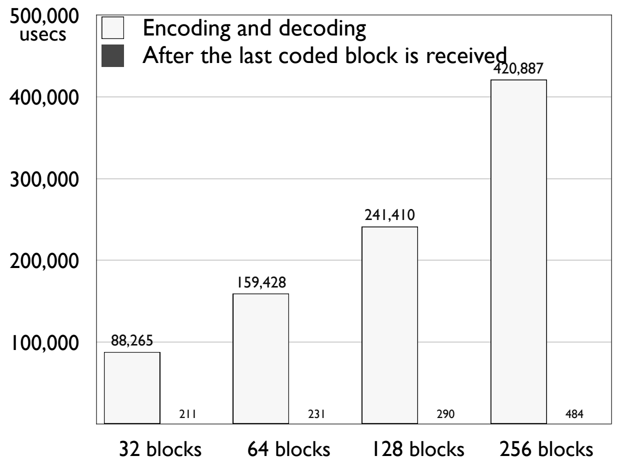


(b) The effects of progressive decoding: the time required to receive all blocks of a segment (including encoding and progressive decoding times), and the time used to recover the original blocks after the last coded block is received.

Fig. 5. The performance of network coding.

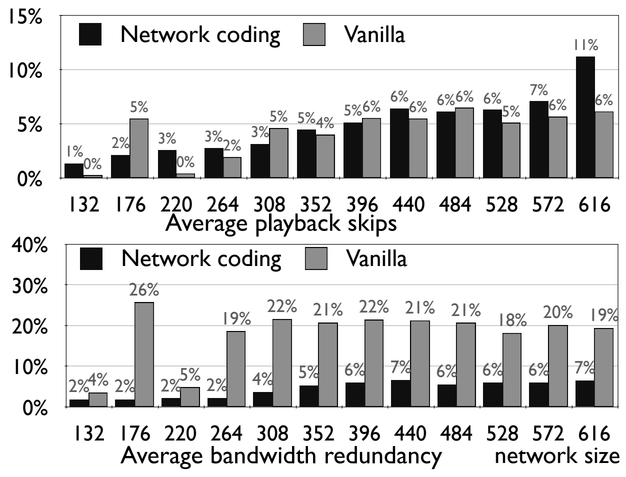


(a) Encoding and decoding bandwidth

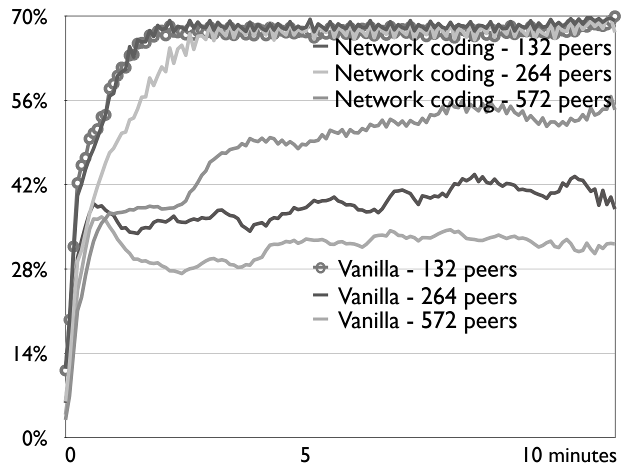


(b) Encoding and decoding overhead

Fig. 6. Coding performance for a session with a streaming rate of 64 KB per second.



(a) Average playback skips when tuning network size.



(b) Average buffering level during the sessions in a 64 KB/sec streaming session deployed to networks consist of 132, 264, and 572 peers.

Fig. 7. Scalability in terms of network size.

timed the transmission time of each segment in a 10-minute session, and took the average from 264 peers. During data streaming, when network coding is employed, peers are exchanging encoded blocks instead of segments as in Vanilla. On one extreme, if a peer discovered only one seed for a particular segment, encoded blocks are sequentially produced, leading to longer delay in transmission. On the other extreme, if a peer has more than 32 seeds for a segment, the encoding process is evenly distributed across 32 seeds, *i.e.* each seed encodes at most one block. The average transmission time depicted in Fig. 8(a) verified this conjecture. It takes network coding longer time to receive earlier segments, since fewer seeds are known by a newly joined peer. As the session progresses, more seeds are discovered; hence, the transmission time is significantly reduced.

The conclusion drawn from Fig. 8(a) is further confirmed in Fig. 8(b), which illustrates the percentage of peers in the network that successfully played each segment in the 10-minute session. We noted that network coding has more skips in the first 60 seconds of a session. However, all peers experience perfectly smooth playback after 70 into the session. We refer to the skips in the first 60 seconds as the *initial skips*.

C. Tuning density and aggressiveness

Theoretically, a lower coding density leads to a smaller number of blocks being coded, which reduces the coding complexity. In addition, a lower aggressiveness setting leads to more "supply" of coded blocks. That said, if peers become too aggressive and start producing new coded blocks too soon, it may not have a sufficient number of original blocks represented in its playback buffer, leading to a potential of linearly dependent blocks being produced. Since the transmission of such linearly dependent blocks consume bandwidth, they lead to *redundancy* in terms of bandwidth usage. Bandwidth redundancy may also be caused by blocks that are received later than the per-segment timeout or after the playback deadline, due to busy seeds and lack of bandwidth.

We are interested in the effects of tuning density and aggressiveness parameters in network coding, compared to Vanilla. For this purpose, we have established a streaming session on 264 peers. We first vary the aggressiveness in network coding, and then vary the density. In Fig. 9(a), although the percentage of playback skips remains insignificant and almost unchanged when tuning the aggressiveness, the bandwidth redundancy is minimized when the aggressiveness is 0.25. Furthermore, Fig. 9(b) shows that the average buffering level grows faster and remains at a higher level when aggressiveness is 0.25. We continue the density experiment with the aggressiveness as 0.25. Both the percentage of playback skips and bandwidth redundancy remain insignificant and almost unchanged as well when tuning the density.

Overall, variations in both aggressiveness and density do not materially affect the playback quality. The percentage of linearly dependent blocks discarded at the peers is insignificant, less than 0.2% of the network traffic. Though these results may seem counter-intuitive, we believe that it is primarily due to the nature of live streaming playback, in that there does not exist sufficient room in each segment for these coding parameter

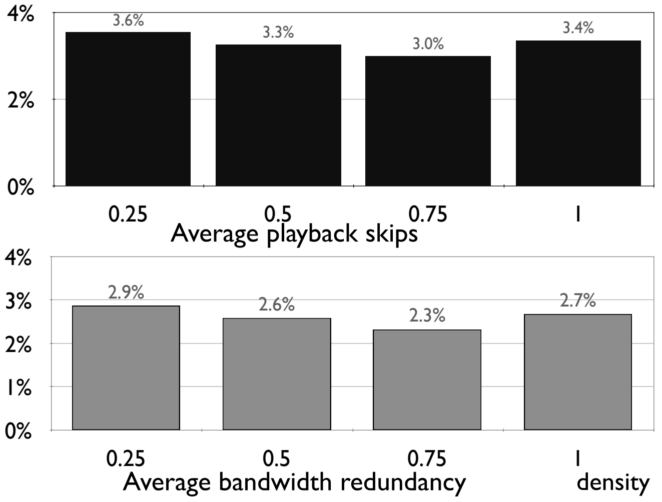


Fig. 10. Average playback skips and bandwidth redundancy when tuning density (aggressiveness set to 25%).

settings to take effect. Since tuning these parameters does not materially affect streaming quality, and the best playback and buffering level is achieved when the aggressiveness and density are 0.25 and 0.75, respectively, we use this setting in the remainder of our experiments.

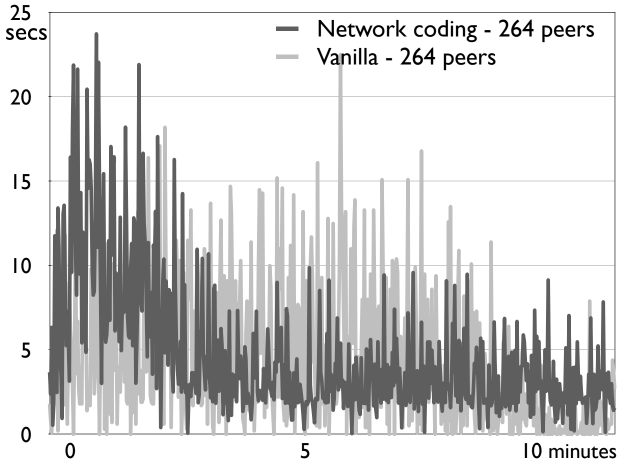
D. Tuning experimental parameters

We conducted extensive experiments to find the best possible performance of network coding in live peer-to-peer multimedia streaming. From these experiments, we discovered that network coding is very sensitive to parameter tuning. This section presents a selective set of experimental results.

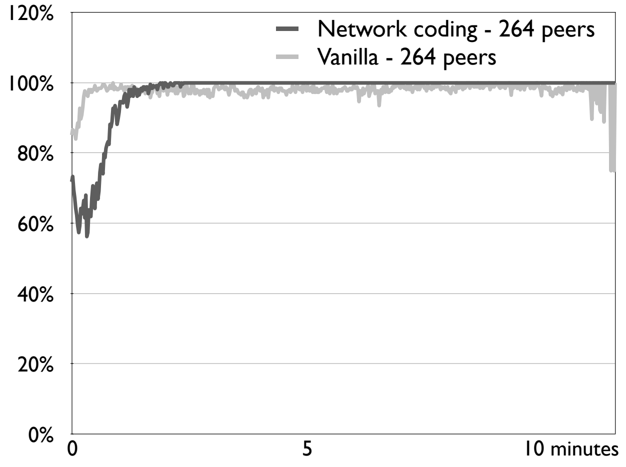
We first tuned the buffer size from 30 seconds to 60 seconds. In the steady state, the playback buffer is filled up to the standard buffering watermark, the first 20 seconds in our case; and the algorithm schedules transmissions at the rate of 1 segment per second. Hence, the buffer should be at least large enough to hold one segment in addition to the standard buffering watermark. When using a large buffer, a peer may have too many outstanding requests. Thus, the earlier segments receive less attention and are more likely to miss their deadlines. Fig. 11(a) shows that a large buffer does not necessarily offer better playback quality and buffering level. Although the buffering levels ramps up faster in larger buffers, they still converge to the same level when the session enters the steady state.

We then fixed the buffer size as 30 seconds, and adjusted the initial buffering delay from 10 seconds to 20 seconds. Intuitively, the longer initial buffering delay should offer better playback quality and higher buffering level, since it allows the buffer to accumulate more segments before the playback starts. As expected, fig. 11(b) shows that the longer delay does improve playback quality and buffering level.

With respect to the segment size, we performed two experiments to vary the time represented by a segment from 1 second to 6 seconds. When the buffer size is fixed, as a segment becomes larger the buffer holds fewer segments, leading to less randomness in scheduling segments for transmission. The result may be lower buffering levels during the initial buffering

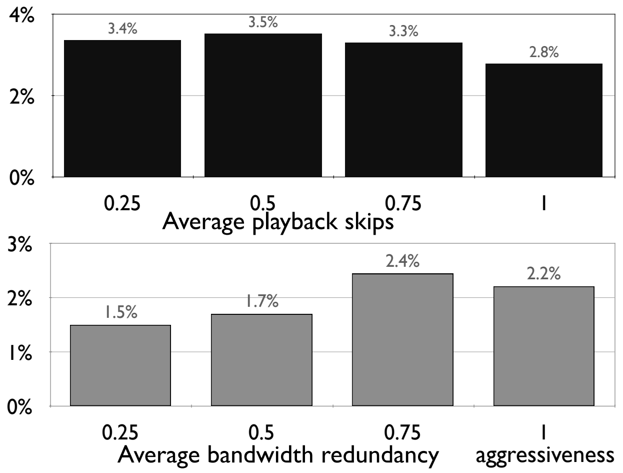


(a) The average transmission time of each segment in a 10-minute session.

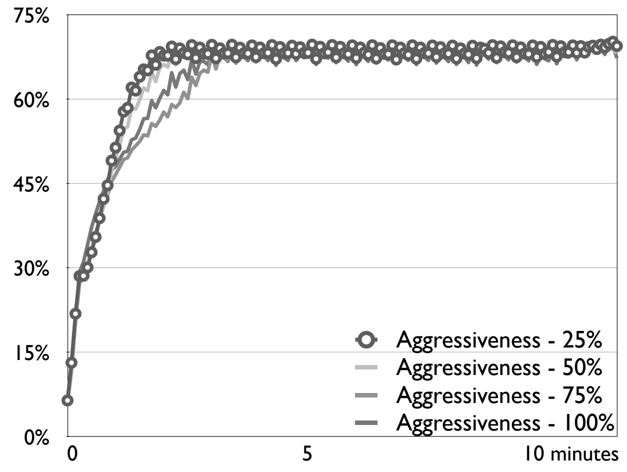


(b) Percentage of peers successfully played each segment in a 10-minute session.

Fig. 8. The transmission and playback status in a 64 KB/sec streaming session on 264 peers.

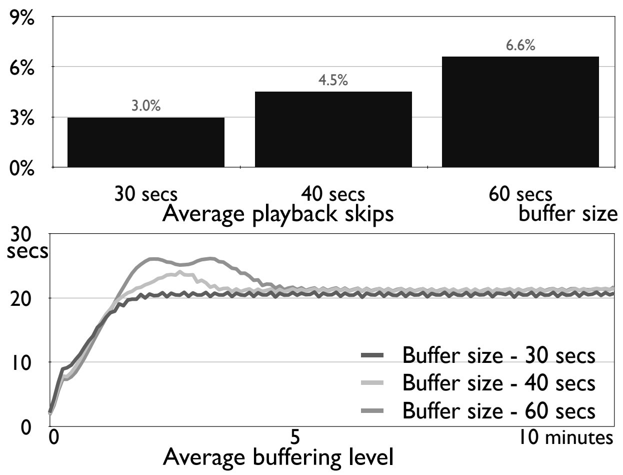


(a) Average playback skips and bandwidth redundancy when tuning aggressiveness (density set to 100%).

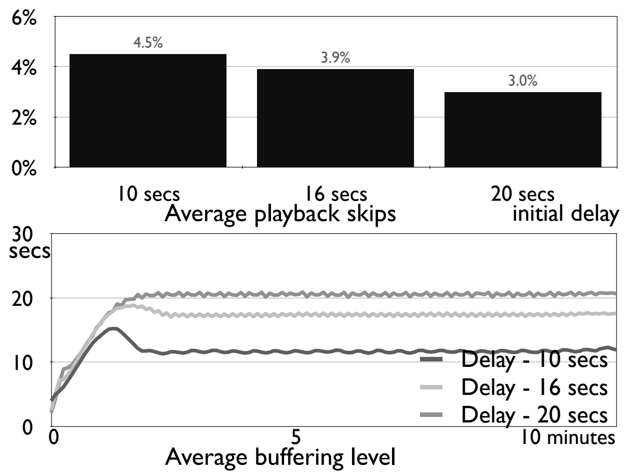


(b) The average buffering level when tuning aggressiveness (density set to 100%).

Fig. 9. Effects of the aggressiveness in a 64 KB/sec streaming session with 264 peers.

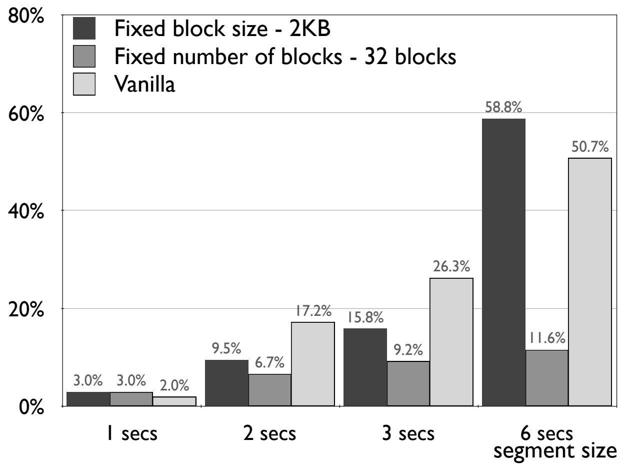


(a) The average playback skips and buffering level during the session, when tuning the buffer size.

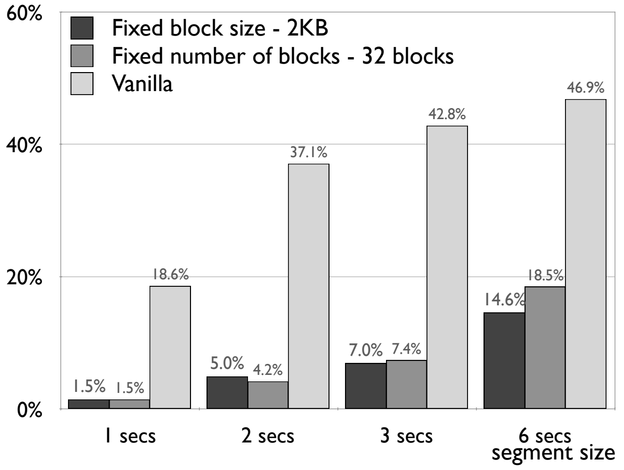


(b) The average playback skips and buffering level during the session, when tuning the initial buffering delay.

Fig. 11. Effects of the playback buffer size and the initial buffering delay in a 64 KB/sec streaming session with 264 peers.



(a) The average playback skips during the session, when tuning the segment size.



(b) The average bandwidth redundancy during the session, when tuning the segment size.

Fig. 12. Effects of the segment size in a 64 KB/sec streaming session with 264 peers.

delay, and eventually broken playback (Fig. 12(a)) and high bandwidth redundancy (Fig. 12(b)).

In the first experiment, we set the block size at 2048 bytes, and increased the number of blocks within a segment from 32 to 196 as the segment size increases. The transmission overhead, message header and coding coefficients, grow proportionally with respect to the number of blocks in a segment. Fig. 12(a) shows that both algorithms cannot maintain a satisfactory playback quality when the segment is longer than 2 seconds. In the second experiment, we fixed the number of blocks in a segment at 32, and increased the block size from 2 KB to 12 KB as the segment size increases. Fig. 12(a) shows that network coding offers much better playback than the previous experiment; thus, it is less sensitive to the block size than it does to the number of blocks.

In general, Vanilla cannot effectively utilize the bandwidth as the segment size increases, resulting in dramatic performance deterioration in playback quality. Since network coding makes it possible to perform data streaming in a *finer granularity*, the bandwidth redundancy is significantly less severe, as indicated in Fig. 12(b).

E. Balance between bandwidth supply and demand

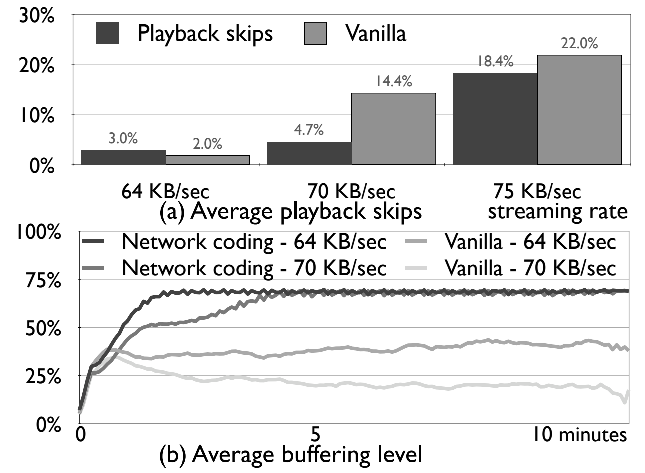
In our previous experiment, bandwidth supply outstrips demand since the peer upload bandwidth is higher than the streaming rate. It may appear that network coding does not lead to improved performance when compared to Vanilla. The question naturally becomes: is this the case when the supply-demand relationship of bandwidth changes? In a network consists of 264 peers, with all DSL-connections except the source, the average bandwidth supply in the network is 93 KB per second for each peer. We run a set of experiments to compare network coding and Vanilla, with three different streaming rates: 64 KB per second to represent the case where supply outstrips demand, 73 KB per second to represent an approximate match between supply and demand, as well as 75 KB per second, when the demand exceeds the supply of bandwidth. When the streaming rate is 75 KB per second, the average bandwidth demand by Vanilla is more than 93 KB per second from each peer, including the protocol messages and the 20% redundant traffic.

From Fig. 13(a), we observed that network coding performs significantly better than Vanilla when there is a close match between supply and demand. When supply outstrips demand or vice versa, there does not exist a significant difference between the two. We also observed that, when bandwidth supply outstrips or meets the demand, network coding is able to consistently maintain a buffering level at the standard buffering watermark, while Vanilla is striving to maintain the buffering level above the low buffering watermark. To further show the benefits of network coding, we reduced the upload capacity on the streaming source, from 1 MB per second to 250 KB per second. Fig. 13(b) illustrates the same pattern as in Fig. 13(a) as the source bandwidth supply drops. The moral of the story is that, in comparison to Vanilla, the streaming quality of network coding excels in the challenging situation when the supply of upload bandwidth only barely meets the streaming demand.

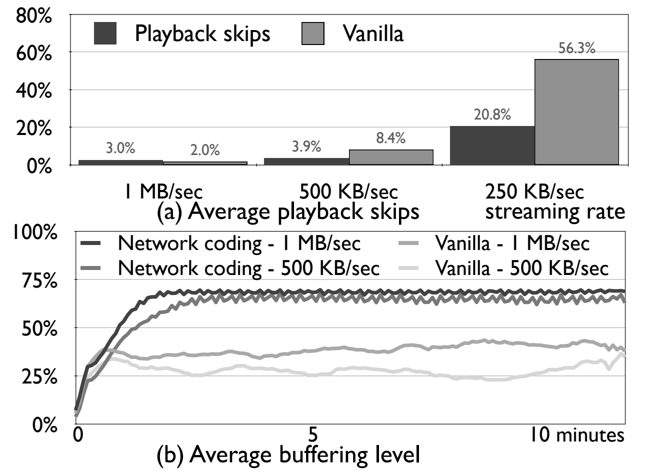
F. Peer dynamics

To investigate the effects of network coding in the case of dynamic peer arrivals and departures, we use Perl scripts to generate peer join and departure events in the *events* log. Based on Stutzbach *et al.* [18], both interarrival times of peer join events and peer lifetimes can be modeled as a Weibull distribution (k, λ), with a PDF $f(x; k, \lambda) = \frac{k}{\lambda}(\frac{x}{\lambda})^{k-1}e^{-(x/\lambda)^k}$, under various settings of the shaping parameter k and scaling parameter λ.

The first case we would like to examine is the performance under different peer join rates. According to [18], peer interarrival time in a file downloading session follows a Weibull distribution with $k = 0.79, 0.53$, or 0.62. We set k to the average value of the three, 0.65. We varied the mean value λ from 80 to 200 seconds so that the join rate reduces and network becomes more dynamic over time. When $\lambda = 80$, it is similar to the flash-crowd scenario, in which more than



(a) The average playback skips and buffering level during the sessions with different streaming rates.



(b) The average playback skips and buffering level during the sessions with different upload capacity on the source.

Fig. 13. Effects of the bandwidth supply and demand in a 64 KB/sec streaming session with 264 peers.

90% of the peers join the session in the first 60 seconds. However, [18] only studied the case of file downloading, where peers usually stay in the session until the completion of the download. In a streaming session, peers may join and leave at any time, and not necessary follow the same distribution. For this reason, we repeat the experiment with $k = 2$, in which the peer interarrival time is approaching normal distribution with different means. For clarity, the plot of each join distribution is shown in Fig. 14. The majority of the peers join the session in the first 60 seconds when $k = 0.65$, whereas peers join the session at a much more slower rate when $k = 2$. Hence, the network is more dynamic in the later case.

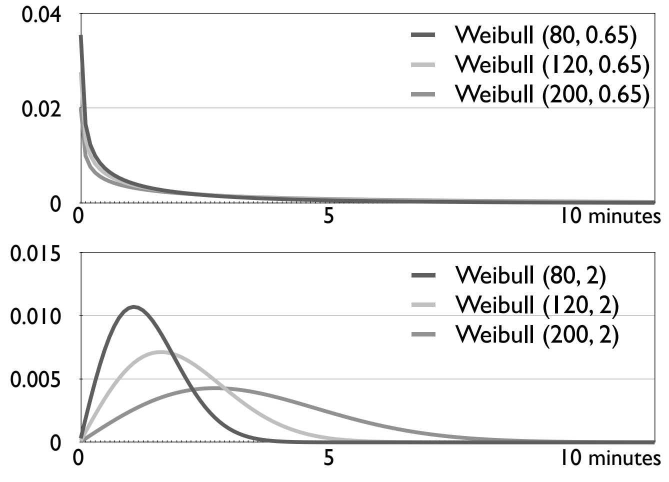


Fig. 14. The PDFs of the peer join rate distributions

As discussed in Sec. IV-B, the number of seeds has direct impact on the performance of network coding. In contrast to the flash-crowd scenario, more seeds can be quickly discovered when peers slowly join the network. For this reason, network coding outperforms Vanilla in terms of overall playback quality in Fig. 15(a). Moreover, Fig. 15(a) also indicates that network coding achieves perfect playback after 60 seconds into the session. Fig. 15(b) shows that the buffering level of Vanilla decreases as more peers joining the network, whereas that of network coding remains the same for all peer join rates.

We then switched our attention to the peer lifetime duration. In this experiment, the join events follow the Weibull distribution $(80, 0.65)$. According to [18], peer interarrival time in a file downloading session follows a Weibull distribution with $k = 0.34, 0.38$, or 0.59. We set k to the average value of the three, 0.43. We varied the mean value λ from 500 to 300 seconds so that average lifetime of a peer becomes shorter, leading to higher churn rate. For the same reason, we repeat the experiment with $k = 2$. For clarity, the plot of each join distribution is illustrated in Fig. 16.

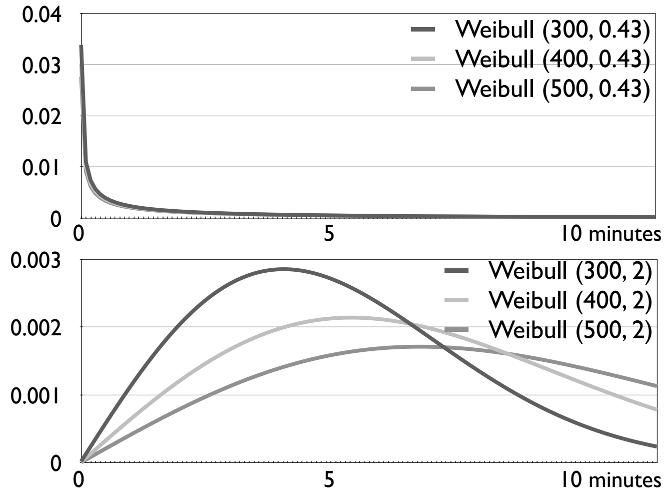
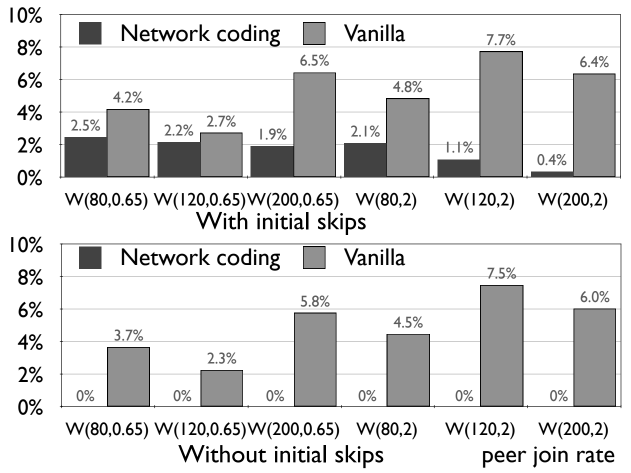
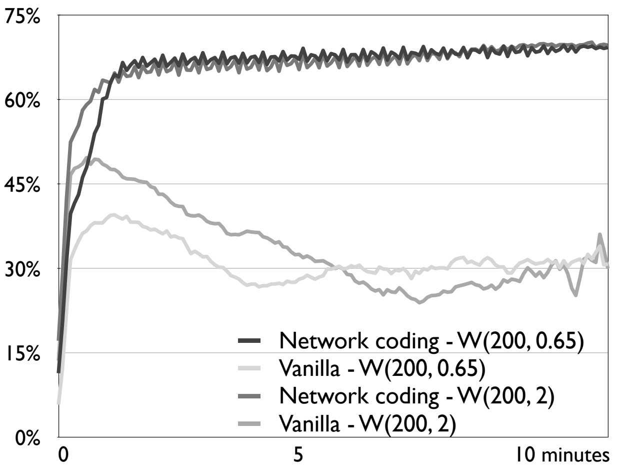


Fig. 16. The PDFs of the peer lifetime distributions

Although network coding offers slight worse overall playback quality than Vanilla, it still achieves perfect playback after 60 seconds into the session, as shown in Fig. 17(a). Vanilla is not able to maintain its performance in networks with higher churns, when $k = 2$. In Fig. 15(b), the buffering level of Vanilla is stabilized only after majority of the peers

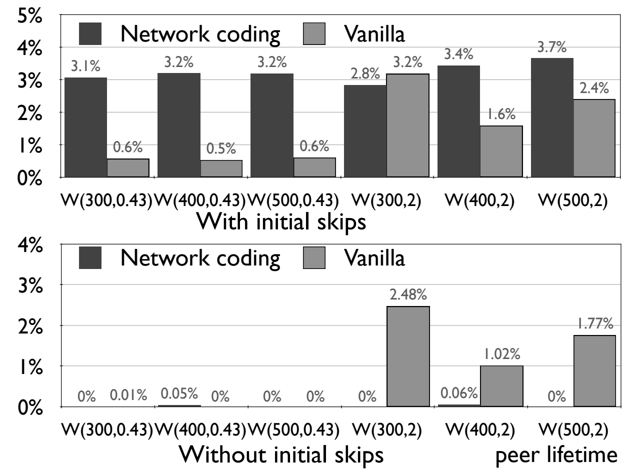


(a) The average playback skips with and without the initial skips during the session, when tuning peer join rate.

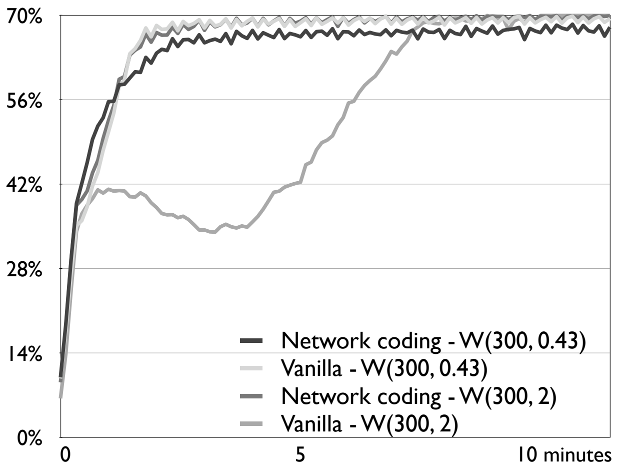


(b) The average buffering level during the session, when tuning peer join rate.

Fig. 15. Effects of the peer join rate in a 64 KB/sec streaming session with 264 peers.



(a) The average playback skips with and without the initial skips during the session, when tuning peer lifetime length.



(b) The average buffering level during the session, when tuning peer lifetime length.

Fig. 17. Effects of the peer lifetime length in a 64 KB/sec streaming session with 264 peers.

joined the session. As peers departing from the session, the ratio between bandwidth supply and demand increases. In this case, Vanilla managed to bring the buffering level up to the standard watermark. Unlike Vanilla, network coding is able to maintain the same buffering level, regardless the network churns.

To conclude, we have made the following important observations in our empirical studies. *First*, network coding offers the same playback quality as Vanilla does, with up to 24% less network traffic. *Second*, the aggressiveness and density setting do not materially change the performance of network coding. *Third*, the buffer size, initial buffering delay, and segment size may have a significant impact on the performance of network coding. *Fourth*, the network coding requires sufficient number of seeds in order to achieve its optimal performance. *Finally*, despite the initial skips due to lack of seeds, network coding is able to maintain perfectly smooth playback and stable buffering level, with the presence of peer dynamics.

V. Concluding Remarks

The objective of this paper is to evaluate the potential and trade-offs of applying network coding in peer-to-peer live streaming sessions, using an experimental testbed in a server cluster, with emulated peer upload capacities and peer dynamics. To achieve a fair comparison between using and not using network coding, we have implemented a pull-based peer-to-peer live streaming protocol in our testbed. As a result of our empirical studies, we believe that network coding has demonstrated its advantages in peer-to-peer live streaming when peers are volatile and dynamic with respect to their arrivals and departures, *i.e.,* the churn rate of the network is high, and in the case that the supply of upload bandwidth barely exceeds the bandwidth demand streaming to all the peers in the session. However, we have also observed that network coding requires precise parameter tuning to achieve its optimal performance, which could be specific to the scale of P2P networks. We do wish to note, however, that the computational *costs* introduced by network coding are very low in typical media streaming rates, especially when progressive decoding using Gauss-Jordan elimination is implemented. To the best of our knowledge, this is the first attempt to study the benefits of network coding for peer-to-peer live streaming in dynamic networks, with a highly-optimized implementation of randomized network coding, and an experimental testbed in server clusters.

References

[1] M. Wang and B. Li, "Lava: A Reality Check of Network Coding in Peer-to-Peer Live Streaming," in *Proc. of IEEE INFOCOM 2007, to appear*, May 2007.

[2] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung, "Network Information Flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, July 2000.

[3] S. Y. R. Li, R. W. Yeung, and N. Cai, "Linear Network Coding," *IEEE Transactions on Information Theory*, vol. 49, pp. 371, 2003.

[4] R. Koetter and M. Medard, "An Algebraic Approach to Network Coding," *IEEE/ACM Transactions on Networking*, vol. 11, no. 5, pp. 782–795, October 2003.

[5] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Medard, and J. Crowcroft, "XORs in The Air: Practical Wireless Network Coding," in *Proc. of ACM SIGCOMM 2006*, 2006.

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

[7] C. Gkantsidis, J. Miller, and P. Rodriguez, "Anatomy of a P2P Content Distribution System with Network Coding," in *Proc. of the 5th International Workshop on Peer-to-Peer Systems (IPTPS 2006)*, 2006.

[8] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, "Data-Driven Overlay Streaming: Design, Implementation, and Experience," in *Proc. of IEEE INFOCOM*, 2005.

[9] S. Katti, D. Katabi, W. Hu, H. Rahul, and M. Medard, "The Importance of Being Opportunistic: Practical Network Coding for Wireless Environments," in *Proc. of Allerton Conference on Communication, Control, and Computing*, 2005.

[10] T. Ho, R. Koetter, M. Medard, D. Karger, and M. Effros, "The Benefits of Coding over Routing in a Randomized Setting," in *Proc. of International Symposium on Information Theory (ISIT 2003)*, 2003.

[11] T. Ho, M. Medard, J. Shi, M. Effros, and D. Karger, "On Randomized Network Coding," in *Proc. of Allerton Conference on Communication, Control, and Computing*, October 2003.

[12] P. Sanders, S. Egner, and L. Tolhuizen, "Polynomial Time Algorithm for Network Information Flow," in *Proc. of the 15th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2003)*, June 2003.

[13] P. Chou, Y. Wu, and K. Jain, "Practical Network Coding," in *Proc. of Allerton Conference on Communication, Control, and Computing*, October 2003.

[14] M. Wang and B. Li, "How Practical is Network Coding?," in *Proc. of the Fourteenth IEEE International Workshop on Quality of Service (IWQoS 2006)*, 2006, pp. 274–278.

[15] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet," in *Proc. of the First Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[16] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), to appear*, December 2002.

[17] Jay Chen, Diwaker Gupta, Kashi V. Vishwanath, Alex C. Snoeren, and Amin Vahdat, "Routing in an Internet-Scale Network Emulator," in *Proc. of IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, October 2004, pp. 275–283.

[18] D. Stutzbach and R. Rejaie, "Understanding Churn in Peer-to-Peer Networks," in *Technical Report CIS-TR-05-03, University of Oregon*, June 2005.