Lava: A Reality Check of Network Coding in Peer-to-Peer Live Streaming

Mea Wang, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto
{*mea, bli*}*@eecg.toronto.edu*

***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 [1], [2], [3], 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 [4].

Avalanche [5], [6] 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 [7] 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 [4] and peer-to-peer content distribution [5], 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.

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 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 acyclic graphs has been studied extensively in previous literature (*e.g.*, [3]). In recent studies, network coding has also been shown to be helpful with respect to throughput in wireless networks [4], [8].

Since the landmark paper on randomized network coding by Ho *et al.* [9], [10], 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 [11], 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.* [12], 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 [5], [6] has further proposed that randomized 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 [13] 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 Peer-to-Peer 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.

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. For these reasons, we first preclude a real-world deployment due to its unpredictability with respect to both bandwidth and CPU availability. We believe that 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.

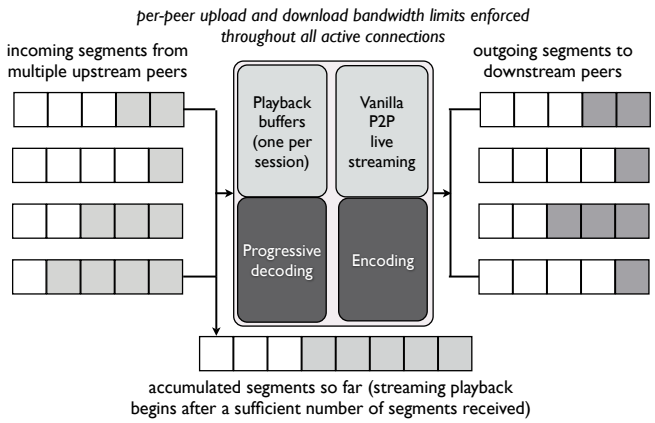

Fig. 1. The architecture of a bandwidth-emulated peer in Lava.

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. 1. 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 (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 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 time; 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, when a session begins, the event specifies the time, the streaming source, and the streaming rate. The *topologies* log is used to *bootstrap* the peer when it first joins 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. 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. In this section, we briefly review our design choices in Vanilla.

Similar to real-world peer-to-peer live streaming systems (*e.g.,* CoolStreaming [7] and PPLive), Vanilla employs a *data-driven* pull-based peer-to-peer streaming protocol. As in any pull-based peer-to-peer streaming protocol, peers 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. 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. At any time, a peer makes concurrent requests for missing segments in its playback buffer, from an arbitrarily selected seed of each segment. The number of concurrent requests is upper bounded. In the case of network coding, since segments are further divided into blocks and then coded, the peer downloads coded blocks from multiple seeds. To avoid overloading the seeds, Vanilla limits the number of concurrent TCP connections on each peer. When a requested segment is not received in time, Vanilla requests the segment again (possibly from a different seed) after a per-segment timeout. In the event of peer departures, Vanilla re-transmits the affected segments from other seeds.

During smooth playback, all segments should be readily available before its playback deadline. 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 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 in the next period to increase the transmission rate. Naturally, the peer returns to its normal mode after it reaches the low buffering watermark again. Fig. 2 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.

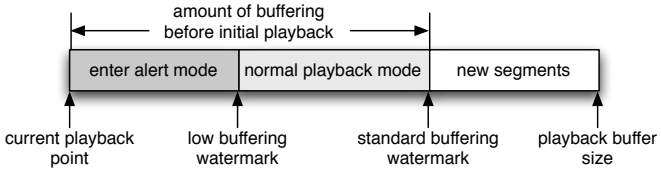

Fig. 2. The playback buffer in Vanilla.

C. Lava: Progressive network coding

Aggressiveness and density in randomized network coding

With randomized network coding [5], [9], [10], [12], 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(2^8), 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 the seeds into its 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 *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*, in that 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.

A peer decodes the 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 b_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.

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. The benefit of the reduced row-echelon form is that, 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.

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. Though Gauss-Jordan elimination usually leads to numerical instability, it does not affect network coding since we operate in the Galois field.

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.

x86 SSE2 acceleration

To further optimize the network coding implementation, 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 an algorithm that performs a combination of bit rotations and exclusive-ORs in a loop (with 8 iterations) to produce the product of two bytes. This algorithm is relatively easy to perform "batch processing" of special 128-bit SSE registers in x86 CPUs, making use of the SSE2 instructions to operate on these registers. 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. The implementation details of our accelerated framework are omitted due to space constraints, and will be presented in a separate paper.

Architectural design

The architecture of the network coding implementation in Lava is summarized in Fig. 3. Every time a new coded block is received into a segment in the playback buffer, progressive decoding is performed by applying Gauss-Jordan elimination to all received blocks of this segment. As decoding progresses, intermediate outcomes of Gauss-Jordan elimination are stored in the playback buffer, until the entire segment is completely decoded.

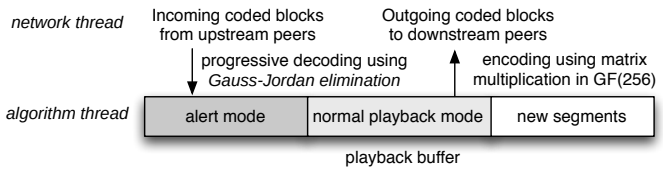

Fig. 3. The architectural design of network coding in Lava.

Finally, an important note we wish to make is that, depending on the settings of aggressiveness, the encoding process in network coding uses either the intermediate or fully decoded blocks from the playback buffer, and progresses *concurrently* with the decoding process. Thus, the encoding progress may start before the decoding process is complete, such that peers may appear more "aggressive."

IV. Network Coding in P2P Streaming: A Reality Check

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?* In all experiments, we set each segment to represent 1 second of playback, and the playback buffer to contain 30 segments. The low and standard buffering watermarks are 10 and 20 seconds, respectively. The initial buffering delay is set to 20 seconds, and the maximum number of concurrent requests for missing segments is 10.

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 dual-CPU 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 use 100% for both density and aggressiveness, since we wish to evaluate the raw coding performance.

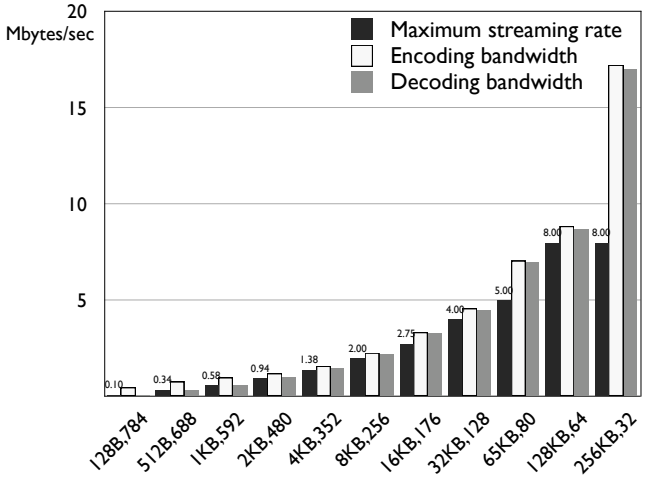

Fig. 4. The encoding bandwidth and maximum sustained streaming rates with different numbers of blocks. We have attempted streaming rates up to 8 MB per second.

In our experiments, we activate all optimization switches, including x86 SSE2 acceleration. 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. 4, 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 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. 4, 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. This time, we measure the time required to completely receive a segment ("transmission time"), and the time spent in 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, 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. 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.

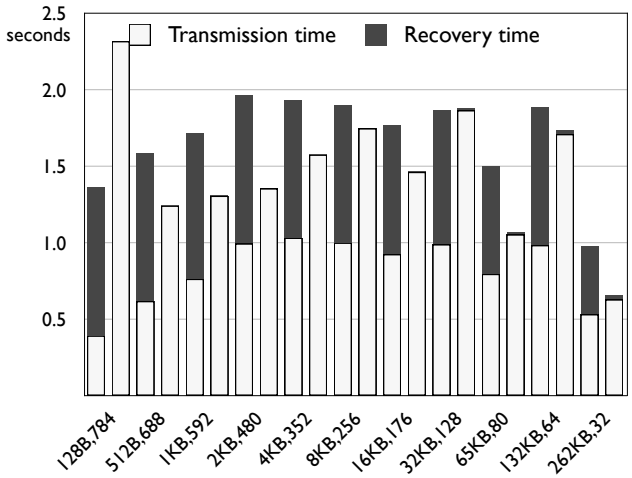

Fig. 5. The effects of progressive decoding: the time required to receive all blocks of a segment (including encoding and progressive decoding times), and the time used to recover the original blocks after the last coded block is received.

B. 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, 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 with 88 peers in a server cluster of 44 dual-CPU cluster node, *i.e.*, each peer has a dedicated CPU. With respect to the "supply" of upload bandwidth, the streaming source of the session is constrained to 1 MB per second, and all peer connections are emulated as DSL uplinks, uniformly distributed between 80 and 100 KB per second. With respect to the "demand" of live streaming, we use a streaming rate of 64 KB per second, which should be satisfied at all peers during their streaming playback. Each segment is divided into 32 blocks since it offers a high encoding and decoding bandwidth (19.3 MB per second), and introduces little header overhead (32 bytes). The streaming session lasts for 10 minutes. To evaluate playback quality, we measure the percentage of playback skips during streaming playback, caused by a missing segment when it is due for playback. To evaluate the level of redundancy when using bandwidth, we measure the percentage of discarded blocks (due to linear dependence or obsolescence) over all received blocks. For all measurements, we take the average from all 88 peers.

We first vary aggressiveness in network coding, and then vary density. In Fig. 6(a), we show that both the percentage of playback skips and bandwidth redundancy remain insignificant and almost unchanged when tuning aggressiveness. The same

can be observed when tuning density as well, as shown in Fig. 6(b). We have also observed that the percentage of linearly dependent blocks discarded at the peers before the segment is completely decoded is insignificant, at less than 0.03% of the network traffic.

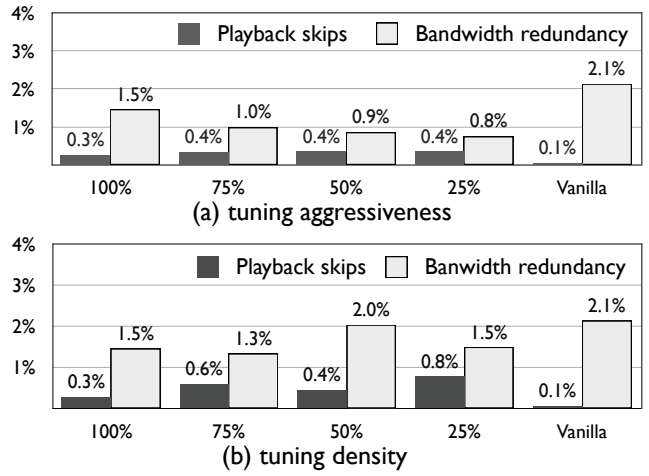

Fig. 6. Average number of playback skips and average number of discarded blocks due to either linear dependence or obsolescence, when tuning (a) aggressiveness (density set to 100%); and (b) density (aggressiveness set to 100%).

Though these results may seem counter-intuitive, we believe that it is primarily due to the nature of live streaming playback, in that there are not sufficient room in each segment for these coding parameter settings to take effect. In addition, it is also because of the relatively small number of blocks in each segment when network coding is used. Since tuning these parameters does not materially affect streaming quality, and the best playback is achieved when both aggressiveness and density are 100%, we use this setting in the remainder of our experiments.

C. 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? We run another 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 78 KB per second, when the demand exceeds the supply of bandwidth. Fig. 7 shows the results of our comparison study.

From Fig. 7(a), we have 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. As shown in Fig. 7(b), the bandwidth redundancy of Vanilla increases remarkably faster than that of network coding, as the streaming rate increases. This is

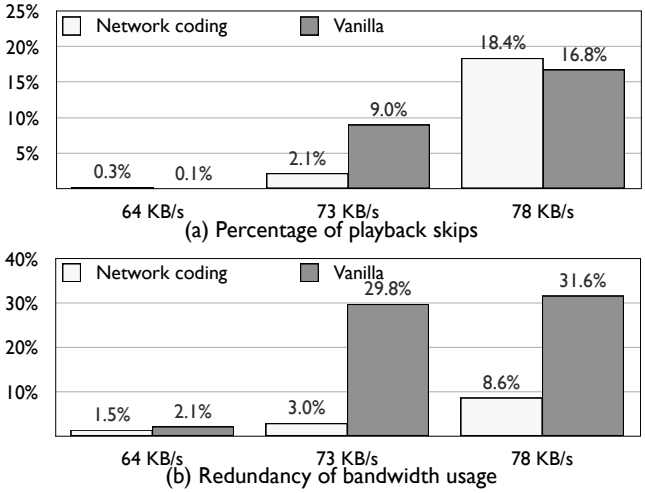

Fig. 7. A comparison between network coding and Vanilla with respect to (1) average percentage of playback skips; and (2) redundancy of bandwidth usage, when tuning the streaming rate.

because that, with network coding, peers may be served by multiple randomly selected upstream peers that have coded blocks of the requested segment, leading to fewer redundant transmissions due to requests timing out. The key insight is that network coding makes it possible to perform data streaming in a *finer granularity*, so that the impact of a bandwidth supply shortage is significantly less severe.

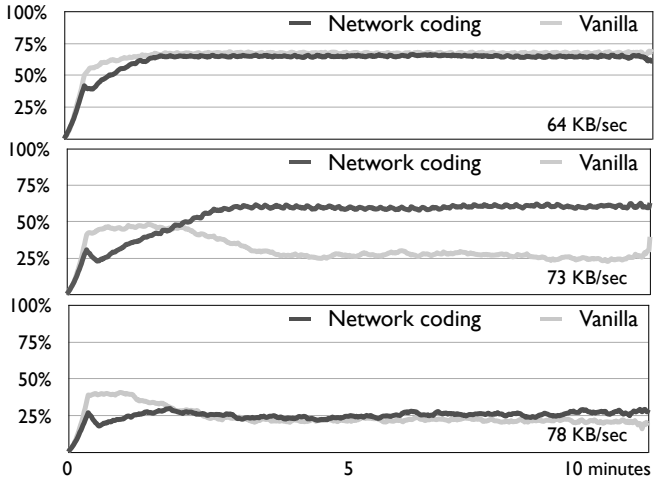

Fig. 8. A comparison of the average peer buffering levels between network coding and Vanilla, over the session lifetime. Three different streaming rates represent different supply-demand relationships with respect to bandwidth.

To further show the benefits of network coding, we show the average buffering levels of peers over the session lifetime in Fig. 8, for each of the three streaming rates. We observed that, when bandwidth supply outstrips demand, both network coding and Vanilla are able to easily locate available segments from neighboring peers, and to increase the buffering levels to the standard buffering watermark. In the case when the demand and supply closely matches each other, 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. When

the bandwidth demand exceeds the supply, neither is able to maintain satisfactory buffering levels. 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.

Finally, we note that the initial buffering level of network coding increases less aggressively than that of Vanilla, in Fig. 8. This is due to the higher processing overhead that network coding introduces, when it works with blocks, rather than segments. This is especially true at the beginning of a session when a peer only knows one or two seeds for each segment. As more seeds are discovered, the overhead is balanced across seeds. For this reason, there are more skips in the first 30 seconds of a session. We refer to these skips as the *initial skips*. This also explains why network coding always has higher playback skips than Vanilla in Fig. 6.

D. Scalability

We now compare Vanilla and network coding when the number of peers in the network scales up. In our previous experiments, each server is 12% and 7% loaded by network coding and Vanilla, respectively. 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. As shown in Fig. 9(a), the CPU usage of both algorithms grows linearly with respect to the network size. When the network size reaches 308 (7 peers on each server), network coding consumes more than 60% of CPU, and its performance degrades significantly. Due to its computational complexity, network coding is not as scalable as Vanilla in our emulation environment. 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.

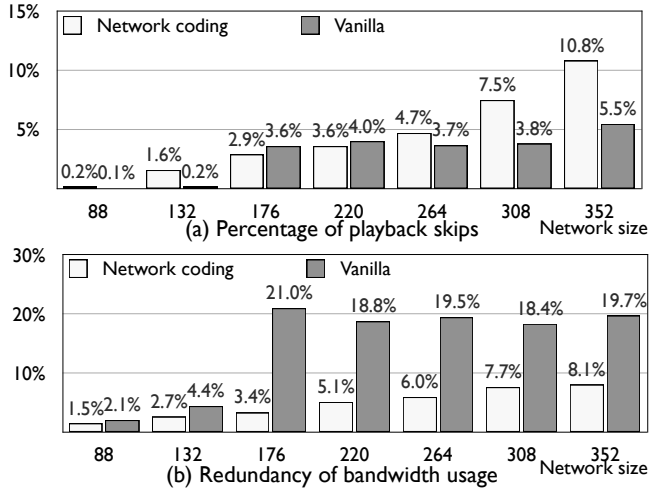

Fig. 9. A comparison of the scalability of network coding and Vanilla: (1) average percentage of playback skips; and (2) redundancy of bandwidth usage.

Fig. 9(b) shows that the bandwidth redundancy introduced by Vanilla is approximately 20% of the total network traffic, when the network consists of more than 176 peers. Network coding is more scalable in term of bandwidth redundancy. Since both algorithms have similar playback quality with 264 or fewer peers, we use this setting in the remainder of our experiments.

E. 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.* [14], both interarrival times of peer join events and peer lifetimes are 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 shape parameter k and scale parameter λ.

The first case we would like to examine is the flash crowd scenario. We set the join events to follow the Weibull distribution $(60, 10)$. The first peer joins the network at the 35th second starting from the beginning of the session, and all other peers join the network in the subsequent 35-second period. As expected, network coding has higher percentage of playback skips (5.7%) than that of Vanilla (3.4%), due to the slow increase of buffering levels at the beginning of a session. Without the initial skips, both algorithms have 3% average playback skips.

We then switch our attention to the peer lifetime duration. In this experiment, join events follow the Weibull distribution $(60, 2)$, *i.e.,* peers gradually join the session from the beginning of a session. This ensures all peers can successfully join the session since there are always a few peers already engaged in the normal playback mode. We vary parameters k and λ to adjust the length of peer lifetimes. With Weibull distribution $(450, 10)$, peers leave the session gradually starting from the 293th second into the session. With Weibull distribution $(450, 50)$, all peers leave the session in the last two minutes. With Weibull distribution $(500, 200)$, all peers leave the session in the last minute.

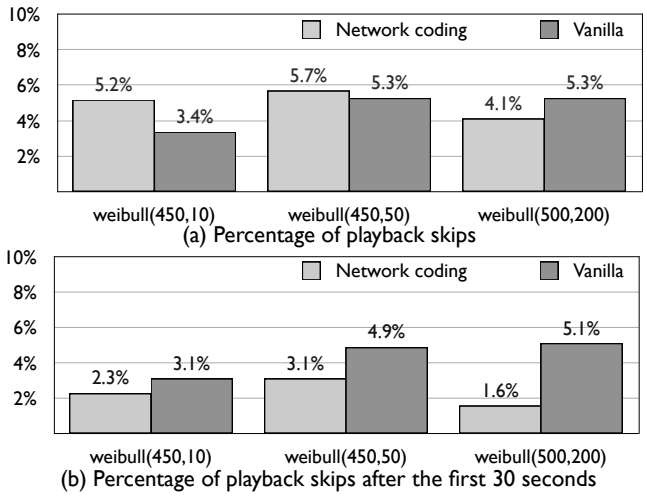

Fig. 10. A comparison of (a) the percentage of playback skips; and (b) the redundancy of bandwidth usage between network coding and Vanilla, under different settings of peer lifetime durations.

Fig. 10 presents a comparison of the average percentage of playback skips with and without the initial skips between network coding and Vanilla, under the three typical settings of peer lifetime durations. Despite the initial skips, network

coding has better performance than Vanilla, especially when peers depart at a faster rate.

We now design more experiments to evaluate the cases of high churn rates. In the first scenario, we used a join Weibull distribution of $(100, 0.5)$ and a lifetime duration Weibull distribution of $(350, 1)$, from which we have the join period overlaps with the departure period. The average playback skips with and without the initial skips of network coding and Vanilla are $(3.8\%, 0.8\%)$ and $(1.3\%, 0.9\%)$, respectively. Fig. 11(a) presents the buffering level of a long-lived peer over the session lifetime, after the initial skips — 50 seconds into the session. Although Vanilla enjoys a better overall playback, its buffering level fluctuates significantly as the network evolves.

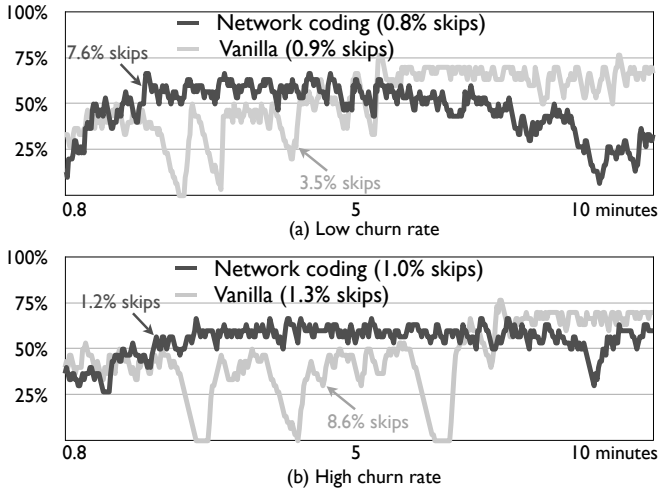

Fig. 11. A comparison of average peer buffering levels between network coding and Vanilla, under (a) join Weibull distribution $(100, 0.5)$ and duration Weibull distribution $(350, 1)$, and (b) join Weibull distribution $(150, 0.5)$ and duration Weibull distribution $(300, 1)$.

In the second scenario, we prolong both join and leave periods so that more peers will join and leave at the same time, leading to higher churn rates. The average playback skips with and without the initial skips of network coding and Vanilla are $(4.4\%, 1.0\%)$ and $(1.8\%, 1.3\%)$, respectively. Fig. 11(b), again, shows that the buffering level after the initial skips is more stable when using network coding, leading to better performance in the long-lived node. Regardless of the join and duration distribution settings, the bandwidth redundancy of both algorithm remains approximately the same as in Fig. 9.

To conclude, we have made the following important observations in our empirical studies. *First*, for typical streaming rates (*e.g.,* 64 KB per second), the aggressiveness and density settings do not have significant effect on the playback quality and bandwidth redundancy. *Second*, network coding makes it possible to perform data streaming with *finer granularity*, so that the impact of a bandwidth supply shortage is significantly less severe. *Third*, the buffering levels with network coding increases slowly at the beginning of a session, due to processing overhead of coded blocks. However, we believe that this phenomenon can be avoided by increasing the number of initial peers at bootstrapping. *Fourth,* though network coding does not improve the playback quality in static sessions, it reduces the amount of redundancy with respect to bandwidth usage, when compared to Vanilla. *Finally,* despite initial skips, network coding demonstrates its resilience to network dynamics, without incurring any additional bandwidth.

V. Concluding Remarks

The objective of this paper is to evaluate the potential and tradeoffs of applying network coding in peer-to-peer live streaming, 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 does have its advantages in peer-to-peer live streaming when the supply of upload bandwidth barely exceeds the bandwidth demand in the session. In addition, network coding maintains stable buffering levels when peers are volatile with respect to their arrivals and departures, and it leads to less redundancy in terms of bandwidth usage. Finally, 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 systems paper to investigate the effects of network coding in peer-to-peer live streaming with an experimental testbed.

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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*, 2003.

[11] 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)*, 2003.

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

[13] 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.

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