R^2: Random Push with Random Network Coding in Live Peer-to-Peer Streaming

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

***Abstract*— In information theory, it has been shown that network coding can effectively improve the throughput of multicast communication sessions in directed acyclic graphs. More practically, random network coding is also instrumental towards improving the downloading performance in BitTorrent-like peer-to-peer content distribution sessions. Live peer-to-peer streaming, however, poses unique challenges to the use of network coding, due to its strict timing and bandwidth constraints. In this paper, we revisit the complete spectrum in the design space of live peer-to-peer streaming protocols, with a sole objective of taking full advantage of random network coding. We present R^2, our new streaming algorithm designed from scratch to incorporate random network coding with a randomized push algorithm. R^2 is designed to improve the performance of live streaming in terms of initial buffering delays, resilience to peer dynamics, as well as reduced bandwidth costs on dedicated streaming servers, all of which are beyond the basic requirement of stable streaming playback. On an experimental testbed consisting of dozens of dual-CPU cluster servers, we thoroughly evaluate R^2 with an actual implementation, real network traffic, and emulated peer upload capacities, in comparisons with a typical live streaming protocol (both without and with network coding), representing the current state-of-the-art in real-world streaming applications.**

I. Introduction

The peer-to-peer communication paradigm has been successfully used in live multimedia streaming applications over the Internet [1]. The essential advantage of live peer-to-peer streaming is to dramatically increase the number of peers a streaming session may sustain with several dedicated streaming servers. Intuitively, as participating peers contribute their upload bandwidth capacities to serve other peers in the same streaming session, the load on dedicated streaming servers is significantly mitigated. Therefore, as one of the most significant benefits, peer-to-peer streaming enjoys the salient advantage of *scalability* in live sessions, where upload capacities on streaming servers are no longer the bottleneck.

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 networks. The upshot of network coding is to allow coding at intermediate nodes in information flows. It has been shown that *random linear codes* using Galois fields of a limited size are sufficient to implement network coding in a practical network setting [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. The intuition is that, with network coding, all pieces of information are treated equally, without the need to identify and distribute the "rarest piece" first.

The requirements of peer-to-peer live multimedia streaming applications, however, have marked a significant departure from applications in content distribution. Since live content becomes available as time progresses, it is delivered to each peer in a roughly sequential order. The most critical requirement is that the *streaming rate* has to be maintained for smooth playback. 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.

Due to such strict timing and bandwidth requirements in live streaming, the advantages of network coding is less obvious, and would certainly justify an in-depth study. Experimentally, we have recently performed a fair comparison between using and not using random network coding in a typical live streaming protocol [8]. We have found that network coding may offer some advantages when peers are volatile and dynamic with respect to their arrivals and departures. To maintain fairness in our comparisons, our previous study have not attempted a redesign of the P2P streaming protocol, and as such may not have taken full advantage of random network coding.

Convinced that random network coding is beneficial, in this paper, we believe a complete redesign of the P2P streaming algorithm is necessary to take full advantage of network coding. We present R^2, our new streaming algorithm designed from scratch to incorporate random network coding with a randomized push algorithm. Though the first and most important requirement of R^2 is to achieve perfect playback, R^2 is nevertheless designed to improve the overall performance, in terms of initial buffering delays, resilience to peer dynamics, as well as reduced bandwidth costs on dedicated streaming servers. On an experimental testbed consisting of dozens of dual-CPU cluster servers, we thoroughly evaluate R^2 with an actual implementation, real network traffic, and emulated peer upload capacities, in comparison with a typical P2P live streaming protocol (both without and with network coding), representing the current state-of-the-art in real-world streaming applications.

The remainder of this paper is organized as follows. Sec. II discusses related work in P2P streaming and practical network coding. The design and implementation of R^2 is presented in Sec. III and IV. We present our experimental experiences with R^2 in live peer-to-peer streaming sessions in Sec. V, and conclude the paper in Sec. VI.

II. Related Work

To alleviate bandwidth demand at dedicated streaming servers, tree-based streaming topologies [9], [10], [11], [12] have been proposed, which organize peers into one or more multicast trees rooted at the streaming servers. The original streaming content is decomposed into sub-streams that are pushed through respective trees from the servers to all peers. Although such push-based tree topologies are beneficial in reducing the delays of distributing live content, they are generally not deployed in real-world streaming systems, mainly due to the complexity involved in maintaining such tree structures in dynamic P2P environments.

The design of CoolStreaming [1] employs a gossip-like protocol to discover content availability among peers, eliminating the need for trees. The streaming content is presented as a series of *segments*, each represents a short duration of playback. A peer in CoolStreaming maintains not only a list of neighboring peers, but also a summary of available segments on these neighbors. Based on such information, segments are *pulled* from appropriate neighbors, in order to meet their playback deadlines and to accommodate heterogeneous streaming bandwidth from the peers. Although such a "pull-based" design with "mesh" topologies is more robust to peer dynamics than push-based tree topologies, it inevitably increases the delay of distributing live content from servers to all participating peers, due to delays caused by periodic exchanges of segment availability. Zhang *et al.* [13] has proposed to combine pull-based and push-based protocols, in order to take advantage of better resilience to dynamics with a pull-based design, and better delay and stability with push-based protocols. Essentially, its push-based design divides the stream into multiple sub-streams, each pushed down a different tree structure.

Network coding has been initially shown to improve information flow rates in multicast sessions in directed acyclic graphs [2], [3], [4], while random network coding has been shown by Ho *et al.* [5], [14] to be feasible in a more practical setting, without deterministic code assignment algorithms. Avalanche [6], [7] has proposed that randomized network coding can be used for elastic content distribution to reduce downloading times. Our previous work [15] has concluded that random network coding would only be computationally feasible if the number of blocks to be coded is fewer than a thousand.

While advantages of network coding have been better understood and tested in scenarios of elastic P2P content distribution, our previous work, *Lava* [8], represents the first fair evaluation on the feasibility and effectiveness of random network coding in live P2P streaming sessions, with strict timing and bandwidth requirements. While a traditional P2P streaming protocol sends original segments, the random network coding "plug-in" component makes it possible to send coded blocks of each segment, such that receiving peers may decode them on-the-fly. With Lava, we have discovered that network coding provides some marginal benefits when peers are volatile with their arrivals and departures, and when the overall bandwidth supply barely exceeds the demand.

While Lava has focused on a *fair* comparison study without any changes of the P2P streaming protocol, we believe the advantages of network coding have *not* been fully explored with a traditional pull-based protocol. In this paper, we are determined to redesign the P2P streaming protocol to take full advantage of random network coding by revisiting entire spectrum of design choices, and by introducing randomizing elements into the algorithm. Unlike the rigid and more "static" push design using predetermined trees in [13], a peer in R^2 randomly chooses a segment to push whenever a coded block (of a very small size) is sent. Peers in R^2 proactively sends segments that are missing from their downstream peers — no "pull" is ever required — and there is never a need to time the switch between pull and push mechanisms on-the-fly.

The idea of random push is partly inspired by Chunked Codes [16]. With Chunked Codes, the message to be communicated is logically partitioned into disjoint "chunks" of contiguous symbols. To encode at the source node, it randomly and uniformly chooses a chunk, and performs a dense linear combination of input symbols from this chunk. The intermediate node, again, randomly and uniformly chooses a chunk, and then performs a dense linear combination of so-far received coded symbols within this chunk. Decoding at the receiver is performed as a regular Gaussian elimination to solve the symbol diagonal matrix. Chunked Codes are designed for the network erasure channel, which is applicable to the P2P streaming case, due to peer dynamics. However, the design of Chunked Codes have been purely analytical and specifically excluded support for the streaming case, due to its strict timing requirements. With R^2, we focus on the design of a practical streaming solution based on random push of coded blocks of each segment, as well as an experimental evaluation of its effectiveness. Since R^2 only encodes within a particular segment, it enjoys similar advantages in terms of coding complexity as Chunked Codes.

III. The Design of R^2

In this paper, we consider a typical live peer-to-peer session (also referred to as a *channel*), with a number of dedicated streaming servers (usually under the administrative control of a service provider), and a large number of peers. Peers participate in and depart from a session in unpredictable ways. The live stream to be served is coded into a constant bit rate, usually in the range of $38 - 50$ KB per second in real-world streaming applications. In this section, we present R^2, our attempt at a complete redesign of the live P2P streaming protocol to take full advantage of random network coding.

A. Traditional Pull-based P2P Streaming

A traditional live P2P streaming protocol — one similar to CoolStreaming and PPLive — utilizes a pull-based mesh streaming mechanism (sometimes referred to as a "data-driven" approach in previous literature). In such a pull-based P2P streaming protocol, peers periodically exchange information on segment availability with active neighboring peers, which is commonly referred to as *buffer maps*. According to the buffer maps, those peers that have a particular segment

available for transmission are referred to as *seeds* of this segment. Each peer maintains a *playback buffer* that consists of segments to be played in the immediate future. For any segment that is missing in the playback buffer, a peer sends an explicit request to a chosen seed of this segment. Segments that are not retrieved in time for playback are skipped during playback, leading to degraded quality.

When a new peer participates in the session, in order to ensure smooth streaming playback, it does not immediate start playback as the first data segments are received; rather, it waits for a period of *initial buffering delay*. During such an initial buffering process, the new peer attempts to download the initial segments from as many other peers as possible, subject to availability.

To best saturate peer download capacities, a typical P2P streaming protocol usually allows a peer to concurrently "pull" multiple segments from respective seeds. Each outstanding "pull" request is an active *task*. The protocol usually imposes an upper bound on the number of tasks, in order to reduce context switching overhead among tasks (which may corresponds to OS-level threads). To avoid overloading the seeds, the number of concurrent outgoing connections on a seed is also bounded, which may be computed based on the upload capacity and the streaming rate. In the event of peer departures, the affected segments are usually "pulled" again from other seeds, with the hope that they will arrive in time for playback.

B. R^2: Design Objectives

Traditional pull-based P2P streaming protocols are simple to implement (possibly within a few thousand lines of Python code), and are robust to peer arrivals and departures. Current-generation real-world protocols, such as PPLive and UUSee, have used such pull-based strategies, and have been able to serve tens of thousands of users in a session with smooth playback. However, we believe that the user experience in P2P streaming sessions can be further enhanced by considering the following quality metrics that are beyond basic playback quality:

- ▷ *Shorter initial buffering delays*: Since the initial buffering delay must be experienced by a user when it switches to a new channel, a shorter delay dramatically improves the user experience.
- ▷ *Reduced server bandwidth costs*: Since peer upload capacities may not be sufficient to sustain the entire streaming session for all participating peers, dedicated streaming servers provide additional "supply" of bandwidth. Since operational costs of these dedicated servers depend on the bandwidth consumed, we believe that it is critical to minimize server bandwidth costs. Even if monetary operational costs are not a concern, minimizing server bandwidth usage allows sufficient unused server capacity to cater to the demand spike in "flash crowd" scenarios.
- ▷ *Better resilience to extremely volatile peers*: When peers join and leave in an extremely volatile fashion, we wish to maintain smooth playback as much as possible.
- ▷ *Smoother playback when bandwidth supply barely exceeds the demand*: With an increasing number of peers and a fixed number of dedicated streaming servers, the saturation point will eventually be reached, where the bandwidth supply barely exceeds the demand. We wish to maintain smooth playback as much as possible in such challenging situations. This is especially the case in unpopular sessions, when a very small number of servers (usually just one or two) and a few hundred peers are engaged. The playback quality in current pull-based protocols is usually unsatisfactory in such cases, due to the lack of seeds and unpredictable bandwidth among peers.

C. The Essence of R^2: Random Push with Random Network Coding

Random Network Coding. Random network coding serves as the cornerstone of R^2, and is instrumental towards most of the advantages of R^2. In traditional P2P streaming protocols, the live stream to be served is divided into *segments*, such that they can be better exchanged among peers. In R^2, each segment is further divided into n *blocks* $[b_1, b_2, \ldots, b_n]$, each b_i has a fixed number of bytes k (referred to as the block size). If the segment duration (for example, four seconds) and the streaming rate is predetermined, the block size k can be directly computed from n.

When network coding is used on a seed (a serving peer or streaming server), we do not propose to code across different segments. This is similar to the approach taken by Chunked Codes, which only codes within a chunk. This is primarily for the purpose of reducing the number of blocks to code, leading to much reduced coding complexity of dense linear codes, as has been well established in the literature [16].

When a seed encodes for a downstream peer p, as a critical aspect of the algorithm design, we need to address the question: *Which segment should the seed select in which to code and send a coded block?* Trivially, segments that p has already received should be excluded, and the decision would be made among the remaining segments that p has not completely received so far. In a traditional pull-based protocol, such a decision is made on the downstream peer p, and explicit requests are made to the seed. The seed then honors the request by sending the segment. To better take advantage of random network coding in R^2, we instead use *random push*, in which the seed *randomly chooses* a segment whenever it produces *one* coded block, among all remaining segments that p has not completely received. The coded block is then sent to p without the need of any requests. Since all coded blocks are equally innovative, all seeds of p cooperatively serve the missing segments on p, without any explicit exchanges of protocol messages.

When a segment is selected by a seed to code for its downstream peer p, the seed independently and randomly chooses a set of coding coefficients $[c_1^p, c_2^p, \cdots, c_m^p] (m \leq n)$ in GF(2^8) for each coded block to be sent to p. It then randomly chooses m blocks — $[b_1^p, b_2^p, \ldots, b_m^p]$ — out of all the blocks in this segment that it has received so far (all the *original* blocks in the segment if the seed is a streaming server), and

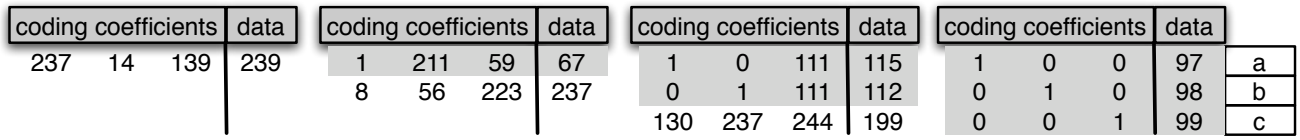


Fig. 1. An example of progressive decoding with Gauss-Jordan elimination.

produces one coded block x of k bytes:

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

The ratio m/n is referred to as *density*, and a low ratio leads to sparse decoding matrices. Our previous work [15] has experimentally shown that the density can be as low as 6% without leading to linear dependence among coded blocks, which has been analytically corroborated in Ma *et al.* [17].

The coding coefficients used to encode *original blocks* to x are typically embedded in the header of the coded block [18] for transmission. We thus need a total of n coefficients, leading to an overhead of n bytes per coded block. These n coding coefficients to be embedded can easily be computed by multiplying $[c_1^p, \cdots, c_m^p]$ with the $m \times n$ matrix of coding coefficients embedded in the incoming blocks $[b_1^p, b_2^p, \ldots, b_m^p]$. We note that an overhead of n bytes may still be considered substantial when n is large (*e.g.,* 128) and the block size k is small (*e.g.,* 1 KB). If $m/n = 1$ and the seed has all n blocks of a segment when producing a coded block, we just need to embed the random seed used to produce the series of random coefficients with a known pseudo-random number generator. This effectively reduces the overhead to just 4 bytes, regardless of n and k.

As the session proceeds, a peer accumulates coded blocks into its playback 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 produces a new coded block upon receiving $\alpha \cdot n$ coded blocks ($0 < \alpha \leq 1$), in which the tunable parameter α is referred to as *aggressiveness*. A smaller α implies that downstream peers can be served sooner. In other words, a peer is more "aggressive" to become a seed.

Coupled with a choice of flow control algorithm (such as TCP-friendly flow control), a coded block is best sent over the UDP transport protocol, and uses TCP only when UDP is not available (*e.g.,* due to firewalls blocking UDP traffic). This is due to the inherent error resilience of random network coding, in that should a particular coded block be lost, subsequent coded blocks received are equally innovative and useful. This is in sharp contrast to traditional pull-based streaming, in which TCP is preferred to send segments, due to its built-in reliability.

With Gauss-Jordan elimination implemented in the decoding process [8], a peer starts to progressively decode a segment, as soon as it receives the first coded block of this segment. As a total of n coded blocks $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ has been received, the original blocks can be immediately recovered as Gauss-Jordan elimination computes:

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

where A is the matrix formed by coding coefficients of $\mathbf{x}$. The use of progressive decoding with Gauss-Jordan elimination implies that the start of the decoding process does not have to wait for all n coded blocks. It also enjoys an additional benefit: if a peer has received a coded block that is linearly dependent on existing blocks, the elimination process will lead to a row of all zeros, so that this coded block can be immediately discarded. This eliminates explicit linear dependence checks when all n blocks are received.

We illustrate progressive decoding in Fig. 1, with 3 blocks in a segment, and one byte of data in each block. The coding coefficients and the actual data (in their ASCII values) is separated by a vertical bar. For each newly received coded block, Gauss-Jordan elimination is applied to the coding coefficient matrix A on the left. The same operation is carried out in $\mathbf{x}$ on the right. Each iteration partially decodes the data by reducing A to the Row Reduced Echelon Form (RREF). Once the last block is received, the segment is recovered with a final iteration.

Random Push. We have shown that, in R^2, whenever a seed sends a coded block to a downstream peer, it needs to use the "random push" mechanism by randomly selecting a segment to code, among segments that the downstream peer has not yet completely received. How does a seed randomize such a segment selection process for each outgoing coded block? The answers to this question constitutes an important design choice in R^2.

Naturally, an important concern at the downstream peer is that it should expedite the downloading process of "urgent" but missing segments, *i.e.,* those missing segments that are close to their playback time. This range of urgent segments may be τ seconds after the playback point, and is referred to as the *priority region*, as shown in Fig. 2. Since there are no explicit requests made by the downstream peer (no "pull" required), seeds should give strict priority to the segments within the priority region. In our randomized segment selection, we stipulate that a seed should randomize *within* the priority region using an uniform distribution, whenever segments in this region are still missing in the downstream peer. From the viewpoint of a receiving peer, as playback progresses, if a few missing segments eventually fall into the priority region, their urgency guarantees that *all* of its seeds will serve these segments. If the receiving peer has sufficient download bandwidth, it should be able to completely receive these missing segments before playback.

If there are no missing segments in the priority region at the downstream peer, the seed will choose missing segments

from outside of the priority region in the playback buffer. Such a randomized choice is subject to a certain probability distribution with a PDF that gives preference to segments that are earlier in time. In our experiments (Sec. V), we use a Weibull distribution with a PDF $f(x;k,\lambda) = \frac{k}{\lambda}(\frac{x}{\lambda})^{k-1}e^{-(x/\lambda)^k}$, such that different shapes of the PDF may be obtained by simply tuning the shape parameter k and scale parameter λ. An alternative distribution is acceptable as well, as long as it prefers earlier segments.

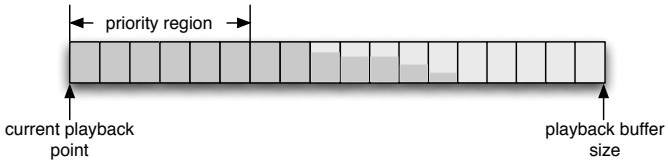


Fig. 2. The playback buffer in R^2.

To summarize, as long as the receiving rate at a peer exceeds the streaming rate, the priority region of the playback buffer should be always filled. The peer concurrently transmits segments in the remainder of the playback buffer, where earlier segments take precedence over the later ones. The dark shade in Fig. 2 indicates the receiving status of each segment in the playback buffer on a typical peer.

D. Timely Feedback from Downstream Peers

One outstanding but important question from the previous discussion is: *How does the seed obtain precise knowledge of the missing segments on its downstream peers at any time?* In traditional pull-based protocols, a *buffer map* is exchanged among peers periodically, which is a bitmap that represents segment availability in the playback buffer. The period of such an exchange cannot be too short, as a typical playback buffer in traditional pull-based protocols usually contains hundreds of segments. We perform the following back-of-the-envelope calculation: With 480 segments, a buffer map needs 60 bytes. With dozens (if not hundreds) of neighboring peers, if we exchange buffer maps every second, it amounts to 6 KB/sec on-the-wire overhead from exchanging buffer maps alone (out of around 40 KB/sec streaming bit rate)! For this reason, most real-world protocols exchange buffer maps less frequently.

Even with the level of overhead in the unrealistic case of exchanging every second, a seed may still be sending segments that are no longer missing in the downstream peers, since its knowledge may be up to a second *delayed*. In the traditional pull-based protocol, such delayed knowledge is less of a concern. Since the seed will not send the segment until it is explicitly requested, such delayed knowledge only leads to delayed requests. In R^2, such delayed knowledge of missing segments is no less than catastrophic: it will lead to redundant coded blocks being sent to and discarded by a downstream peer, that are no longer useful, but consume bandwidth nevertheless.

The design of R^2 stipulates that buffer maps be exchanged with much higher frequency. As a matter of fact, the buffer maps are no longer sent *periodically*. Instead, a downstream peer sends its buffer map whenever the buffer status changes — when it has played back a segment, or when it has completed the downloading of a segment. Whenever possible, the buffer map is embedded in outgoing coded blocks. Otherwise, it is separately sent to the neighboring peers. With such a design, R^2 guarantees that the delay of obtaining precise buffer maps from downstream peers is never higher than the network transmission delay on the overlay links, which is in the range between a few to a few hundred milliseconds. We further note that, as an arbitrary pair of peers will be likely to serve as seeds for each other, such explicit transmission of buffer maps may rarely be needed.

The buffer maps are also used as a signal for the seed to *stop* a segment transmission, once the segment has been completely received (likely with the assistance from other seeds). Since buffer maps are sent in the most timely fashion, such a "negative" signal is received as soon as the network allows. In fact, there is nothing in the design of R^2 that prevents downstream peer to send the negative signal even *before* it has completely received the segment, in order to *prematurely* stop a subset of seeds for this segment, usually when segment downloading is almost completed. Such a *premature braking* algorithm may be designed to favor seeds with better bandwidth to complete the download, and stop those seeds with lesser inter-peer bandwidth. The design of such algorithms may be quite elaborate, gradually stopping more seeds based on precise completion timing estimates of the downloading process. In our experiments, we do not include this feature, and leave its design to our future work.

How does R^2 manage the excessive overhead of exchanging buffer maps, then? Let us revisit the example discussed earlier, in which a playback buffer has 480 segments representing 160 seconds of playback, around 15 KB per segment with a streaming rate of 45 KB/second. R^2, instead, divides the buffer into 40 segments of 4 seconds each, and further divides each 180 KB segment into 180 blocks of 1 KB each. This leads to just 5 bytes to represent each buffer, which can be easily embedded in a 1 KB coded block with a 4% overhead, when required. Moreover, a segment is removed from the buffer every 4 seconds, and a segment is completely received every 4 seconds in a steady state. Hence, a peer sends at most 2 buffer maps to each neighboring peer every 4 seconds on average. It amounts to approximately 200 bytes/sec on-the-wire overhead to exchange buffer maps among dozens of peers, a significant improvement in comparison to the 6 KB/sec overhead offered by a traditional protocol.

Finally, why is R^2 able to use much larger segments? In traditional pull-based protocols, we observe that a missing segment on a downstream peer can only be served by one seed at a time (with the possibility of switching to a different seed if the transmission fails due to low bandwidth or peer dynamics). With random push coupled with random network coding, a segment can be served by *multiple* seeds, as each seed uses its randomized selection algorithm to select a segment to send coded blocks. We refer to such a phenomenon as *perfect collaboration*, since seeds collaborate with each other without any protocol messages. Such an excellent property is due to the fundamental characteristic of dense random linear codes, in that any coded block is as good as any other, regardless

of the seed that produces them. The sharp contrast between a traditional P2P streaming protocol and R^2 is shown in Fig. 3.

E. Synchronized Playback

As we have much fewer segments in the playback buffer in R^2, we prefer to recruit as many seeds as possible for each segment. To achieve this, we wish to make sure that the playback buffers overlap as much as possible among peers. R^2 features *synchronized playback* as follows. In a live streaming session, the playback buffers on all peers are synchronized so that all peers plays the same segment at approximately the same time. When a peer joins a streaming session, it first retrieves buffer maps from its initial seeds (usually assigned by tracking servers), along with the current segment being played back. To synchronize the playback buffer, the new peer *only* retrieves segments that are δ seconds after the current point of playback, while δ corresponds to the *initial buffering delay*. The peer starts playback after precisely δ seconds elapsed in real time, regardless of the current status of the playback buffer. This allows the peer δ seconds to fill as many segments as possible in the priority region of the playback buffer before the playback starts. Naturally, both the priority region and the initial buffering delay can be tuned, and they can be equal to each other. Regardless of the state of the playback buffer, a new peer starts playback after exactly δ seconds have elapsed.

Recall that a seed only selects segments within the priority region if they are still missing in a downstream peer. This is exactly the case when a new peer joins, with an empty playback buffer. If the priority region is the same as the initial buffering delay, within δ seconds of initial buffering delay, *all* the seeds of a new peer start to serve segments within the priority region. Akin to a "flash crowd" of seeds, such a phenomenon in R^2 easily saturates the download bandwidth of the new peer, and if it exceeds the streaming rate, R^2 guarantees smooth playback during the priority region. In practice, this ensures that R^2 does not need to employ an exceeding large initial buffering delay (in the order of one minute in PPLive, for example), and can use as small as $10-20$ seconds.

A possible drawback of synchronized playback is that, the time between the occurrence of a live event in the media stream and its playback is the same across the board in all peers in the entire session. Though seemingly harmful, this may even be an advantage when live interaction is involved (such as live voting with SMS): *all* peers will view the same content at the same time, such that interactive behavior starts to occur at the same time as well.

F. Random Selection of Downstream Peers

To make sure that coded blocks from one segment is not "spread too thin" in all the peers, a seed only sends a segment to a limited number of downstream peers at any given time, subject to an upper bound. To select such limit, the seed can randomly select from all its downstream peers, or select those that have historically had the highest flow rate with the seed. The maximum number of downstream receivers should be linearly related to the upload capacity of a seed: the lower the upload capacity, the smaller number of active downstream receivers it should maintain. This design choice in R^2 places "emphasis" on a small number of receiving peers for a particular segment, which accelerates the rate of initial propagation of a segment that has just been made available on dedicated servers. For such a segment, as the number of peers who have already received it exceeds a threshold, the remaining peers will be able to download smoothly — leading to exponential propagation behavior.

When a seed randomly chooses downstream peers for a segment, each segment should have a different and randomly generated set of seeds. This randomizes the data dissemination process since a seed serves different segments to different sets of peers. The randomized selection of both downstream peers and segments (for a particular peer) in R^2 is perfectly resilient to peer departures and network losses.

G. R^2: Design Objectives Revisited

Let us now revisit the original design objectives that we have outlined, and note how R^2 fulfills these requirements.

- ▷ *Shorter initial buffering delays:* Peers in R^2 enjoy shorter initial buffering delays, as smooth playback is guaranteed if sufficient seeds are used to saturate the peer download capacity. This is due to our design of synchronized playback, as well as perfect collaboration among seeds due to random network coding.
- ▷ *Reduced server bandwidth costs:* With network coding, every coded block being transmitted is equally useful to the receiving peer. With multiple seeds serving any segment, and without any overhead incurred by explicit requests, the probability of saturating both peer upload and peer download bandwidth capacities is much higher than traditional pull-based protocols. Both of these factors contribute to reducing server bandwidth costs, since peers are able to serve more useful bits to one another.
- ▷ *Better resilience to peer dynamics:* In R^2, resilience to peer dynamics has been significantly improved. Since multiple seeds are used to serve each segment, the departure of one or a few of them does not constitute a challenge.
- ▷ *Smooth playback when bandwidth supply is tight:* Since R^2 utilizes bandwidth as efficiently as possible with UDP-based transmission of coded blocks under flow control, and since R^2 gives strict priority to urgent missing segments, playback quality will be much improved as compared to traditional pull-based protocols, especially in challenging scenarios when overall bandwidth supply barely exceeds the overall demand in the entire channel.

IV. R^2: Implementation

As we prepare to evaluate the performance of R^2, we believe that the most insightful results are achieved with an actual implementation. We utilize a cluster of 48 dedicated dual-CPU servers (Pentium 4 Xeon 3.6 GHz and AMD Opteron 2.4 GHz), interconnected by Gigabit Ethernet. In order to obtain accurate results, R^2 should be evaluated with network settings that are as close to the reality as possible, with real TCP

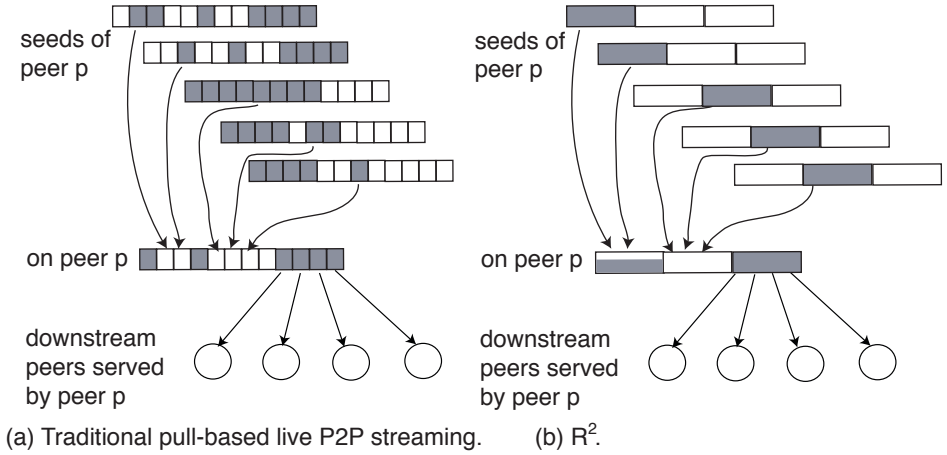


Fig. 3. An illustrative comparison between a traditional P2P streaming protocol and R^2. While traditional P2P streaming protocols have smaller segments, and each segment is served by one seed, R^2 can afford to have larger segments (that are further divided into blocks), and each segment is served by multiple seeds.

and UDP flows. In addition, since we wish to maximize the number of peers to be emulated on any server in the cluster, the processing and memory footprint of our implementation must be minimized.

We start our implementation of R^2 from basic building blocks: multithreading, data block processing and forwarding, timed and periodic event scheduling, network socket programming, as well as exception handling. In addition, we have implemented a lightweight emulation of arbitrary peer upload and download capacities, end-to-end delays, as well as peer arrivals and departures. We also developed a set of facilities to automate the deployment, troubleshooting, and data collection.

Each peer in the R^2 implementation consists of only two threads. The *network* thread is in charge of maintaining all the incoming and outgoing TCP connections or UDP flows, their corresponding FIFO queues, as well as bandwidth and delay emulation. It also emulates the dedicated streaming servers by producing data at the streaming rate. Incoming and outgoing network traffic are monitored by a single *select()* call with a specific timeout value. The timeout value of the *select()* call is tuned on-the-fly, and is critical to the scalable implementation of bandwidth emulation. The *engine* thread, on the other hand, processes incoming blocks, and sends coded blocks to outgoing connections. The engine thread also implements random network coding, using random linear codes over GF(2^8). To optimize the computational efficiency of encoding and decoding processes, we take full advantage of hardware acceleration by using SSE2 SIMD instruction sets, supported by both AMD and Intel processors.

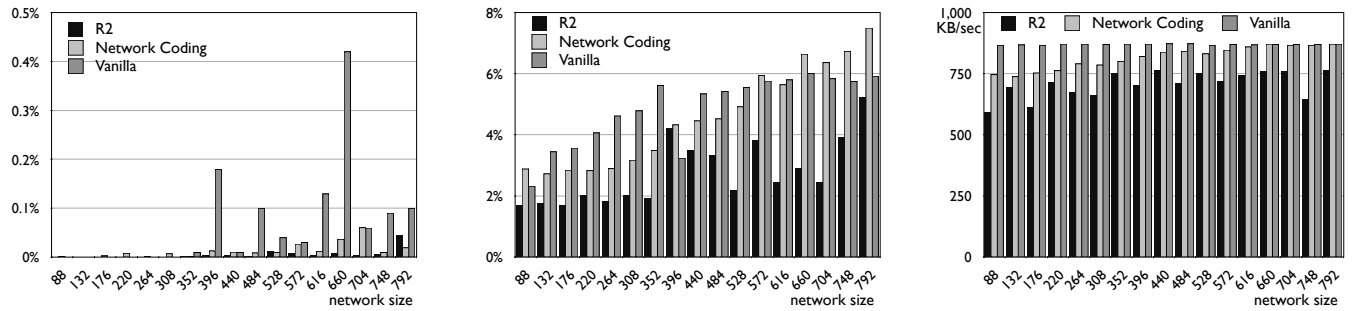
V. R^2: Performance Evaluation

The focus of our experiments is to examine the effectiveness of R^2 with respect to our design objectives. For the sake of comparison, we implemented a conventional streaming protocol, henceforth referred to as *Vanilla*. Similar to existing protocols such as CoolStreaming [1] and PPLive, Vanilla employs a data-driven pull-based design philosophy, as have been described in Sec. III-A. Moreover, we also implemented *Vanilla with network coding* [8] as a comparison (later referred to as *network coding*), which uses network coding as a "plugin" component in Vanilla, without making changes to the pull-based streaming protocol.

A large real-world streaming channel involves tens of thousands of peers, supported by dozens of high performance servers with at least 10 MB per second upload capacity each. It is impossible for us to emulate such a large streaming population with actual traffic using only 48 clustered servers. With network coding, each emulated peer consumes roughly $5 - 7\%$ of the CPU; hence, we are approaching 100% CPU usage with 800 peers. In order to obtain reasonable observations from such a relatively small network, we use only one streaming server with 1 MB per second upload capacity, and limit all other peer connections to DSL grade, with upload capacities uniformly distributed between 80 and 100 KB per second. We use a streaming rate of 64 KB per second, a typical rate in real-world streaming protocols. We believe that such a scenario may be experienced in real-world channels that are relatively unpopular, when very few streaming servers have been deployed. In these cases, the real-world user experience has often been quite poor as well, corroborating our observations in these sets of experiments. By emulating a streaming channel using such extreme settings, our results represent a baseline level of performance, and we fully expect R^2 to perform better in reality with more popular channels.

In all experiments, unless specified otherwise, each segment represents 4 seconds of the playback, and is divided into 128 blocks, offering a satisfactory encoding and decoding bandwidth with our implementation of random linear codes (around 4 MB per second). Each streaming session lasts for 10 minutes. For a more challenging scenario, we set the buffer size to 32 seconds, the initial buffering delay to 16 seconds, and the priority region to 8 seconds.

To evaluate the performance of R^2, 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 playback time. (2) *Bandwidth redundancy:* To evaluate



(a) Average playback skips.

(b) Average bandwidth redundancy.

(c) Uplink bandwidth consumption on the dedicated streaming server.

Fig. 4. Average playback and bandwidth status on each peer in a 64 KB/sec streaming session deployed to P2P sessions of various sizes, ranging from 88 to 792.

the level of redundancy when using bandwidth, we measured the percentage of discarded segments or blocks (due to linear dependence or obsolescence) over all received segments or blocks. (3) *Buffering levels* on each peer during a live session over time, measured as the percentage of received blocks or segments in the playback buffer. (4) The *uplink bandwidth consumption* on the dedicated streaming server. All measurements are averaged over all peers in the session.

A. Scalability

We first evaluate the scalability of R^2, by varying the number of peers in the live streaming session from 88 to 792 peers. In this experiment, peers join the streaming session as soon as there is sufficient bandwidth supply in the session, *i.e.* the bandwidth demand closely matches the bandwidth supply, especially at the beginning of the session. Fig. 4(a) shows that R^2 offers steady playback quality, with less than 0.02% of playback skips, whereas Vanilla and network coding have an increasing percentage of playback skips as the number of peers increases. The improvements in R^2 is due to its effective use of bandwidth, as shown in Fig. 4(b), which is brought by network coding and the use of smaller blocks rather than larger segments. For the benefit of the service provider that hosts dedicated streaming servers, R^2 saves almost 15% of the upload bandwidth on the streaming server, as shown in Fig. 4(c). Subject to exactly the same scenarios, however, R^2 delivers robust and convincing performance in terms of both playback quality and bandwidth costs on the servers.

B. Buffering Levels

As an important metric to gain insights on the playback quality, we next examine the fluctuations in average buffering levels over the course of a streaming session. Fig. V-B compares the average buffering levels of R^2, network coding, and Vanilla in two representative sessions, from which we draw two key observations. First, the buffering level ramps up quickly and remains stable in R^2, while Vanilla maintains a much lower level with a slight variation over time. Network coding maintains a slightly higher buffering level than Vanilla does. Second, the buffering level of R^2 increases as more peers become available, while that of Vanilla and network coding decrease as the network size increases. The satisfactory buffering levels in R^2 also explains the perfect playback quality (represented by nearly nonexistent playback skips) in Fig. 4(a). As observed, the random push algorithm and the priority region design guarantee in-time and fast delivery of each segment.

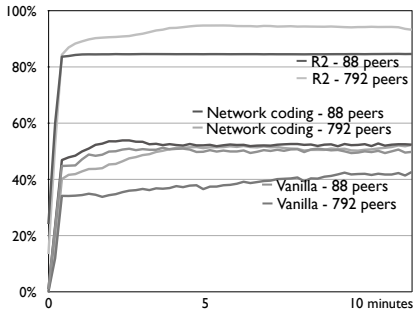


Fig. 5. Average buffering level during the sessions and on each peer in a 64 KB/sec streaming session deployed to networks that consist of 88 and 792 peers.

C. Initial Buffering Delays

To better illustrate the advantage of R^2 in effective segment transmission, we present the time that all three algorithms take to fill the priority region when a peer joins a session in Table I. Although network coding improves the time from 14 seconds to 12 seconds in traditional pull-based protocol, such a time still increases with the network size. In sharp contrast, R^2 is able to completely fill the priority region in less than 6 seconds regardless of the number of peers in the session, *i.e.*, the upload bandwidth on the seeds are fully utilized in transmitting encoded blocks, and very few blocks are being discarded.

Knowing that it takes less time for R^2 to fill the priority region, we turn our attention to the impact of the initial buffering delay and the length of the priority region. Intuitively, longer initial buffering delays should lead to better playback quality and higher buffering levels. However, we show in Fig. 6 that the effect of different initial buffering delays in R^2 is not as significant as it is in traditional protocols.

session size	88	132	176	220	264	308	352	396	440	484	528	572	616	660	704	748	792
R^2	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6
Network coding	10	10	11	11	11	11	11	12	12	11	12	12	12	12	12	12	12
Vanilla	10	11	11	11	12	12	13	14	13	13	14	14	14	14	14	14	13

TABLE I

THE AVERAGE TIME (IN SECONDS) TAKEN TO FILL THE PRIORITY REGION IN SESSIONS INVOLVING DIFFERENT NUMBERS OF PEERS.

We also observed that both Vanilla and network coding have unacceptable playback quality when the initial buffering delay is 8 seconds, which explains why they cannot support fast channel switching well.

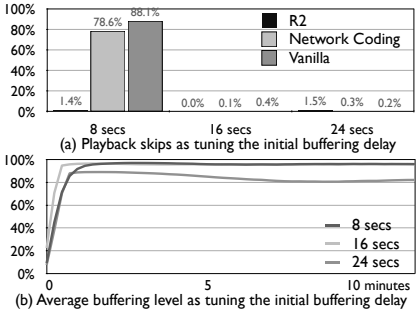


Fig. 6. The average playback skips and buffering level during the session when tuning the initial buffering delay, in a 64 KB/sec streaming session with 800 peers.

To gain a better understanding of the priority region setting, we fixed the initial buffering delay to 24 seconds, and increased the length of the priority region from 8 seconds to 20 seconds (more than half of the playback buffer). As the priority region grows, the earlier segments in the buffer become less "urgent", leading to lower playback quality, as shown in Fig. 7. However, we note that the length of the priority region does not materially affect the performance, since R^2 effectively utilizes all bandwidth to maintains high buffering levels.

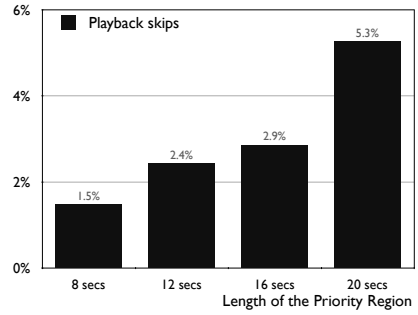


Fig. 7. Average playback skips when tuning the length of the priority region, in a 64 KB/sec streaming session with 800 peers.

D. Effects of Random Network Coding

With respect to the segment size, we performed two experiments to vary the time represented by a segment from 2 to 8 seconds. In the first experiment, we fix the number of blocks in a segment to 128, and increase the block size from 1 KB to 4 KB as the segment size increases. When a segment is too small, more overhead ensues when transmitting small blocks. On the other hand, when a segment becomes larger (8 seconds of playback), the priority region consists of only one segment, leading to less randomized segment selection. The result in Fig. 8 shows lower buffering levels during the initial buffering delay. We have observed that all of the playback skips using large 8-second segments have occurred during the first 8 seconds of playback.

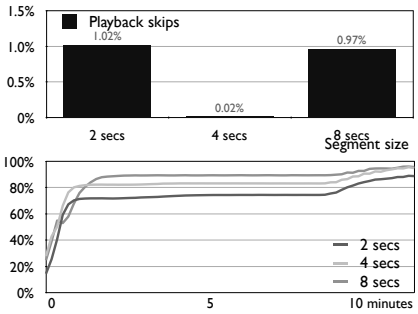


Fig. 8. The average playback skips and buffering levels during a 64 KB/sec streaming session, with a fixed number of blocks in each segment.

In the second experiment, we set the block size to 2048 bytes, and increased the number of blocks within a segment from 64 to 256 as the segment size increases. With a fixed playback buffer size, the number of segments included in the playback buffer increases as the segments become smaller. Therefore, we have observed slightly more randomness during segment selection, leading to slightly better playback quality, as shown in Fig. 9. Though larger segments offer higher buffering levels in both experiments, the priority region is not filled as quickly as during the initial buffering delay. Moreover, larger segment does not necessarily offer better quality since a missing segment may results in longer skips in seconds. In our experiments, it is ideal to have 4-second segment broken into 132 blocks.

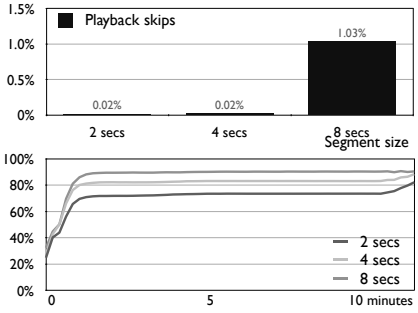
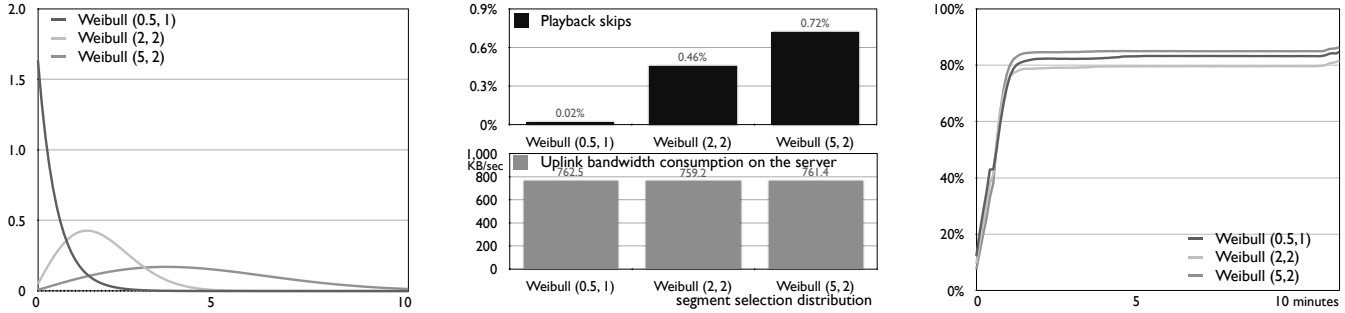


Fig. 9. Average playback skips and buffering levels during a 64 KB/sec streaming session with a fixed block size.



(a) The PDF of the probability distribution used in segment selection.

(b) Average playback skips and uplink bandwidth consumption on the server.

(c) Average buffering level during the session.

Fig. 10. The effects of different probability distributions in the segment selection algorithm, in a 64 KB/sec streaming session.

E. Tuning the Random Segment Selection Algorithm

In our previous experiment, we have obtained satisfactory results using Weibull$(0.5, 1)$ — equivalent to exponential distribution — for segment selection outside of the priority region. We further wish to gain a better understanding on the performance impact of different probability distributions used in our segment selection algorithm. In this experiment, we selected two representative parameter settings, Weibull$(2, 2)$ and Weibull$(5, 2)$. Both distributions are approaching the normal distribution, but with different mean values. The former gives more preference to earlier segments than the latter does. For clarity, the PDFs of our distributions are shown in Fig. 10(a). As shown in Fig. 10(b), distributions that favor the earlier segments offer better playback quality, without consuming additional upload bandwidth on the dedicated streaming server.

F. When Bandwidth Demand Meets the Supply

In our previous experiment, bandwidth supply outstrips demand since the peer upload bandwidth is higher than the streaming rate. It may appear that R^2 does not lead to improved performance when compared to Vanilla and network coding. The question naturally becomes: is this the case when the supply-demand relationship of bandwidth changes? In a session consisting of 800 peers, with all DSL-grade connections except the source, the average bandwidth supply in the session is 92 KB per second for each peer. We run a set of experiments with four different streaming rates: 64 KB per second to represent the case where the supply outstrips the demand, 70 and 75 KB per second to represent an approximate match between the supply and demand, as well as 80 KB per second, when the demand exceeds the supply of bandwidth. When the streaming rate is 80 KB per second, the average bandwidth demand is more than 92 KB per second from each peer, including protocol messages and redundant traffic.

From Fig. 11(a), we observed that R^2 significantly outperforms both Vanilla and network coding when there is a close match between supply and demand, and even when the demand exceeds the supply. We also observed that, regardless of the streaming rate, R^2 is able to consistently maintain a buffering level around 90%, while Vanilla and network coding are striving to maintain the buffering level above the priority region. Fig. 11(b) illustrates the difference in buffering levels

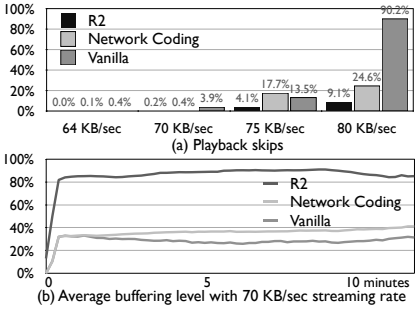


Fig. 11. Effects of the balance between bandwidth supply and demand with 800 peers.

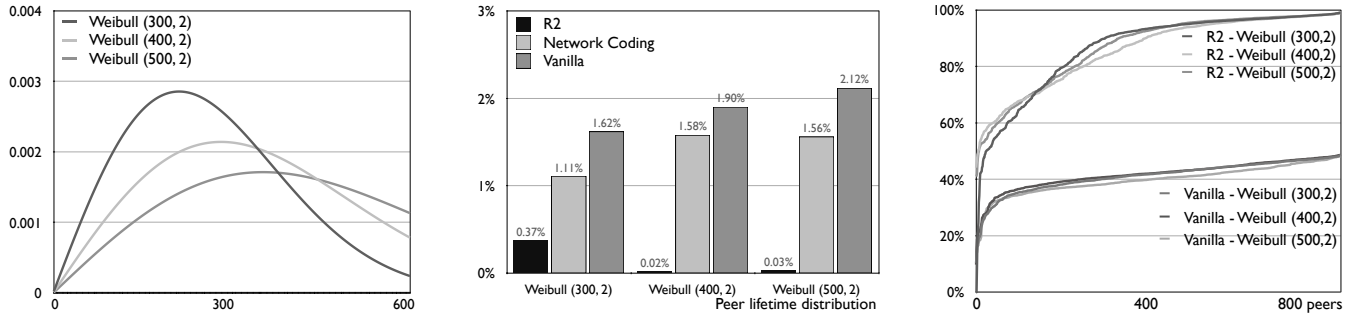
among the three protocols in a 70 KB per second streaming session. Despite the different buffering levels, we notice that R^2 offers a more rapid increase of buffering levels during the initial buffering delay, which is confirmed in Table II. As a result, R^2 can fill the priority region in less then 8 second. In sharp contrast, Vanilla cannot fill the priority region during the entire session when the streaming rate reaches 80 KB per second.

Streaming rate	64 KB/sec	70 KB/sec	75 KB/sec	80 KB/sec
R2	6	6	7	8
Network coding	11	12	14	15
Vanilla	13	14	15	—

TABLE II
THE TIME (IN SECONDS) TAKES TO FILL THE PRIORITY REGION WHEN TUNING THE STREAMING RATE.

G. Effects of Peer Dynamics

To emulate volatile peers with frequent departures, we again use the Weibull distribution — Weibull$(k, 2)$ — to randomly generate the lifetime of participating peers. With Weibull$(k, 2)$, we may conveniently decrease the mean peer lifetime by adjusting k from 500 to 300. For clarity, the plot of each distribution is shown in Fig. 12(a). The shorter the peer lifetime is, the more volatile the session becomes. Fig. 12(b) indicates that the playback skips in all three protocols do not vary significantly as we tune peer dynamics. However,



(a) The PDFs of the peer lifetime distributions. (b) The average playback skips. (c) The average buffering level on each peer.

Fig. 12. R^2 is more resilient to peer dynamics as compared to network coding and Vanilla, in a 64 KB/sec streaming session involving 800 peers.

as shown in Fig. 12(c), the buffering level is more than 90% on more than half of the R^2 peers, whereas all peers in Vanilla has less than half of the buffer filled during the entire streaming session. The intuition behind this phenomena is that all coded blocks are equally innovative with network coding; thus, the content in the buffers at all seeds of a peer are equally important. A peer does not need to identify the blocks or segments affected by a departing seed. All seeds of a peer are able to cooperatively serve missing segments. The performance of Vanilla with network coding is similar to Vanilla (that we choose not to show for clarity of the graph): since it has not been designed to implement perfect collaboration, a segment can only be served by one peer, which may be negatively affected by frequent peer departures.

VI. Concluding Remarks

This paper presents the design and performance evaluation of R^2, with a sole objective of redesigning the live P2P streaming protocol to take full advantage of random network coding. R^2 integrates the following original contributions into a coherent design. First, it employs a randomized push algorithm without the need of making explicit requests. Second, it utilizes random network coding within each segment, making it possible for multiple seeds to cooperatively serve the same downstream peer, with no messaging overhead. Similar to Chunked Codes [16], the use of dense linear codes within a segment reduces the coding complexity to a level that can be realistically implemented and used. Third, as all seeds give strict priority to segments close to the playback point, new peers in a session enjoy a shorter initial buffering delay. Fourth, with synchronized playback, the overlap of playback buffers on participating peers is maximized, leading to much more opportunities for peers to serve one another, and to reduced bandwidth costs on dedicated streaming servers. Finally, with larger segments and much smaller buffer maps (a few bytes), seeds in R^2 receive feedback from downstream peers in a timely manner. As shown in our experiments, R^2 enjoys clear advantages that should not be overlooked or underestimated.

References

[1] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, "CoolStreaming/DONet: A Data-driven Overlay Network for Efficient Live Media Streaming," in *Proc. of IEEE INFOCOM*, 2005.

[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] 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.

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

[9] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai, "Distributing Streaming Media Content Using Cooperative Networking," in *Proc. of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'02)*, May 2002.

[10] M. Castro, P. Druschel, A. M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: High-Bandwidth Multicast in Cooperative Environments," in *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, October 2003.

[11] J. Li, P. Chou, and C. Zhang, "Mutualcast: An Efficient Mechanism for One-to-Many Content Distribution," in *Proc. of the 1st ACM SIGCOMM Asia Workshop (SIGCOMM ASIA '05)*, April 2005.

[12] V. Venkataraman, P. Francis, and J. Calandrino, "Chunkyspread: Multi-tree Unstructured Peer-to-Peer Multicast," in *Proc. of the 5th International Workshop on Peer-to-Peer Systems*, February 2006.

[13] M. Zhang, Y. Xiong, Q. Zhang, and S. Yang, "A Peer-to-Peer Network for Live Media Streaming: Using a Push-Pull Approach," in *Proc. of ACM Multimedia 2005*, November 2005.

[14] 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.

[15] 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.

[16] P. Maymounkov, N. J. A. Harvey, and D. S. Lun, "Methods for Efficient Network Coding," in *Proc. of 44th Annual Allerton Conference on Communication, Control, and Computing (Allerton 2006)*, 2006.

[17] G. Ma, Y. Xu, M. Lin, and Y. Xuan, "A Content Distribution System based on Sparse Linear Network Coding," in *Third Workshop on Network Coding (Netcod 2007)*, January 2007.

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

Mea Wang. Mea Wang received her Bachelor of Computer Science (Honours) degree from the Department of Computer Science, University of Manitoba, Canada, in 2002. She received her Master of Applied Science degree from the Department of Electrical and Computer Engineering at the University of Toronto, Canada, in 2004. She is currently a Ph.D. candidate at the Department of Electrical and Computer Engineering, University of Toronto. Her research interests include peer-to-peer systems, network coding, and wireless networks.

Baochun Li. Baochun Li received his B.Engr. degree in 1995 from Department of Computer Science and Technology, Tsinghua University, China, and his M.S. and Ph.D. degrees in 1997 and 2000 from the Department of Computer Science, University of Illinois at Urbana-Champaign. Since 2000, he has been with the Department of Electrical and Computer Engineering at the University of Toronto, where he is currently an Associate Professor. He holds the Nortel Networks Junior Chair in Network Architecture and Services since October 2003, and the Bell University Laboratories Chair in Computer Engineering since July 2005. His research interests include application-level Quality of Service provisioning, wireless and overlay networks.