rStream: Resilient and Optimal Peer-to-Peer Streaming with Rateless Codes

Chuan Wu, *Student Member, IEEE,* Baochun Li, *Senior Member, IEEE*
Department of Electrical and Computer Engineering
University of Toronto
{*chuanwu, bli*}*@eecg.toronto.edu*

***Abstract*— Due to the lack of stability and reliability in peer-to-peer networks, multimedia streaming over peer-to-peer networks represents several fundamental engineering challenges. First, multimedia streaming sessions need to be resilient to volatile network dynamics and node departures that are characteristic in peer-to-peer networks. Second, they need to take full advantage of the existing bandwidth capacities, by minimizing the delivery of redundant content and the need for content reconciliation among peers during streaming. Finally, streaming peers need to be optimally selected to construct high-quality streaming topologies, so that end-to-end latencies are taken into consideration. The original contributions of this paper are two-fold. First, we propose to use a recent coding technique, referred to as *rateless codes*, to code the multimedia bitstreams before they are transmitted over peer-to-peer links. The use of rateless codes eliminates the requirements of content reconciliation, as well as the risks of delivering redundant content over the network. Rateless codes also help the streaming sessions to adapt to volatile network dynamics. Second, we minimize end-to-end latencies in streaming sessions by optimizing towards a latency-related objective in a linear optimization problem, the solution to which can be efficiently derived in a decentralized and iterative fashion. The validity and effectiveness of our new contributions are demonstrated in extensive experiments in emulated realistic peer-to-peer environments with our *rStream* implementation.**

***Index Terms*— Distributed networks, distributed applications, peer-to-peer protocol, media streaming**

I. Introduction

With peer-to-peer media streaming, streaming servers do not need to directly support a large number of unicast sessions, which effectively eliminates server overloading, and reduces the bandwidth costs on servers by a few orders of magnitude. Despite such an important advantage, peer-to-peer streaming poses significant technical challenges, especially when it comes to real-world and large-scale deployment:

- ▷ *Network dynamics.* Peer-to-peer networks are inherently dynamic and unreliable: peers may join and depart at will and without notice. The demand for stable streaming bit rates may not be satisfied.
- ▷ *Limited bandwidth availability.* Nodes in peer-to-peer networks reside at the edge of the Internet, leading to limited per-node availability of upload and download capacities. To further exacerbate the situation, the available per-node bandwidth differs, by at least an order of magnitude. To ensure uninterrupted streaming playback, typical streaming bit rates in modern streaming codecs must be accommodated for the entire peer-to-peer topology. Even if a particular streaming rate may be satisfied, we may wish to minimize the end-to-end latencies to peer nodes in the streaming session. It is nontrivial to construct a feasible topology to satisfy an arbitrary streaming bit rate, not to mention that with minimized average end-to-end latency.
- ▷ *Delivery redundancy and content reconciliation.* It has become typical in recent peer-to-peer streaming proposals for a peer node to concurrently download from multiple upstream peers, and serve multiple downstream peers. While it improves overall bandwidth availability and resilience to dynamics, there exist fundamental problems in such *parallel retrieval* with respect to delivery redundancy and reconciliation. As there are always risks that the same content may be unnecessarily delivered by multiple upstream peers, the peer nodes need to reconcile the differences among the contents held by different upstream peers before downloading [1], [2], a problem referred to as *content reconciliation*.

The completion of this research was made possible thanks to Bell Canada's support through its Bell University Laboratories R&D program.

While there exists previous work on peer-to-peer streaming (a discussion of which is postponed to Sec. VI), to the best of our knowledge, this paper represents the first attempt to battle on all three fronts of the peer-to-peer streaming challenge. Our main contribution is a peer-to-peer streaming protocol referred to as *rStream*, which involves the combination of *rateless codes* and *optimal peer selection.* We first argue that the recent advances of *rateless fountain codes*, including LT codes [3], Raptor codes [4] and online codes [5], can be readily used in peer-to-peer streaming with substantial advantages. As a class of erasure codes, rateless codes provide natural resilience to losses, and therefore provide the best possible resilience to peer dynamics. Being *rateless*, there is potentially no limit with respect to the number of uniquely coded "blocks," coded from a set of original data blocks. This completely eliminates the needs for content reconciliation, as no redundant contents exist in the network. A sufficient number of coded blocks from any set of peers may be used to recover the original content.

Based on the foundation of rateless codes, we propose an optimal peer selection strategy to guarantee bandwidth availability and to minimize end-to-end latencies. We first formulate the optimal peer selection and rate allocation problem as a linear optimization problem, and then derive an efficient and decentralized algorithm to solve the problem. As rateless codes naturally eliminate the need for content assignment on each link, we are able to deliver useful media content at the optimally computed rates. Our algorithm is reactive to network dynamics, including peer joins, departures and failures.

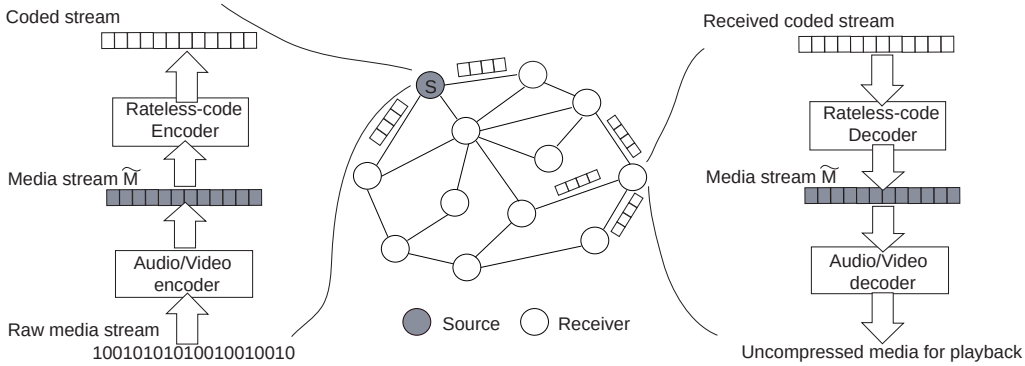


Fig. 1. An illustration of the mesh topology and peer-to-peer streaming model.

The remainder of this paper is organized as follows. In Sec. II, we present our network model and the case for using rateless codes. In Sec. III, we address the optimal peer selection problem, by formulating it as a linear optimization problem, and by designing a distributed algorithm to derive the optimal peer selection and rate allocation strategies. We present the complete *rStream* media distribution and dynamics handling protocols in Sec. IV. Extensive evaluation results based on *rStream* implementation in emulated peer-to-peer streaming environments are presented in Sec. V. We discuss related work and conclude the paper in Sec. VI and Sec. VII, respectively.

II. Resilient Peer-to-Peer Streaming with Rateless Codes

In this paper, we consider a peer-to-peer live streaming session with one streaming *source* and multiple participating *receivers*. We assume there exists a stand-alone bootstrapping mechanism in the peer-to-peer network, consisting of one or multiple bootstrapping servers. When a new peer joins the session, it is bootstrapped with a list of existing peers in the session. During streaming, each receiver peer is served by one or more peers in the *upstream*, and may serve one or more receivers in the *downstream*, constituting a *mesh* streaming topology. The objective is to stream live media content, coded as a constant bit rate bitstream with a current generation codec such as H.264/AVC, H.263 or MPEG-4, to all the participating receivers in the session.

Such a mesh topology can be modeled as a directed graph $G = (N, A)$, where N is the set of vertices (source and peers) and A is the set of directed arcs (directed overlay links). Let S be the streaming source, and let T be the set of receivers in the streaming session. We have $N = S \cup T$. The source S streams a media bitstream $\widetilde{M}$ to the receivers in T. Independent of the codec used in $\widetilde{M}$, we treat $\widetilde{M}$ as a stream of symbols, partitioned into consecutive segments $s_1, s_2, \ldots$. A segment typically consists of one media frame, a group of frames (GOF), or simply a period of time (*e.g.*, one second). Each segment is further divided into k blocks. Each block has a fixed length of l bytes. In *rStream*, we encode each media segment with a rateless code and distribute the coded blocks. An example of the mesh topology and the streaming model is illustrated in Fig. 1.

A. Rateless codes

We now motivate the use of rateless codes. The benefits of rateless codes are related to the fundamental challenges in peer-to-peer streaming: *volatile network dynamics* and *content reconciliation*.

In recent years, *erasure codes* have been applied in peer-to-peer content distribution to cope with network dynamics. A (n, k) erasure code, such as Reed-Solomon codes and Tornado codes [6], is a forward error correction code with k as the number of original symbols, and n as the number of generated symbols from the k original symbols. A (n, k) erasure code is loss resilient, due to its favorable property that if any k (or slightly more than k) of the n transmitted symbols are received, the k original symbols can be recovered. Such loss resilience makes erasure codes an ideal solution for reliable transmission over an unreliable transportation protocol, such as UDP. Also, since any symbol from any upstream peer can be used for decoding, a receiver does not rely on a specific upstream node for the supply of certain original symbols, and no specific upstream node may become a bottleneck. This makes erasure codes failure-resilient. Thus, an erasure code seamlessly tolerates packet losses and peer dynamics, making it ideal for peer-to-peer parallel streaming.

In addition, the use of erasure codes partially alleviates the content reconciliation problem in parallel retrievals. We illustrate the problem with an example in Fig. 2(a). In this example, S transmits the component blocks 1, 2, 3 and 4 of a media segment to t_1 and t_2 directly, and thus t_1 and t_2 have the same four blocks. When t_3 concurrently streams from t_1 and t_2, it has to decide which block to retrieve from which upstream peer. This also occurs on t_4 which concurrently retrieves from t_1 and t_3.

If an erasure code is applied at the data source, the probability of content conflicts among upstream peers can be decreased. However, since the total number of encoded symbols are fixed, the problem is not completely solved with a traditional erasure code. To illustrate this, consider Fig. 2(b). With a $(6, 4)$ erasure code, S generates six coded blocks $1'$, $2'$, ..., $6'$ based on the four original blocks 1, 2, 3 and 4. t_1 and t_2 both directly retrieve four coded blocks from S, and thus inevitably hold two common blocks. This leads to the need for reconciliation at t_3, and later at t_4. Even with a high-rate erasure code where n is much larger than k, content reconciliation may not be necessary in many cases, but the problem is still not completely solved.

To address the challenges from content reconciliation, as well as to provide better resilience to network dynamics, we propose to use a recently developed category of coding schemes, *rateless codes*. Typical rateless codes include LT codes [3], Raptor codes [4] and online codes [5]. With rateless codes, the number of coded symbols that can be generated from k original symbols is up to 2^k, which is potentially unlimited when k is large. Rateless codes are also failure-tolerant as it retains the desirable property that the k original symbols are decodable from any set of slightly more

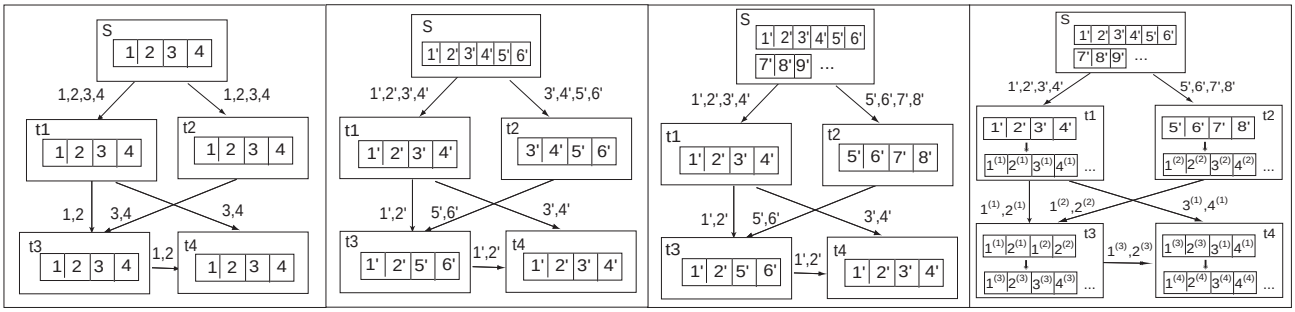


Fig. 2. Peer-to-peer streaming with different coding schemes: an example.

than k coded symbols with high probability.

As compared to traditional erasure codes, rateless codes possess the excellent property of simple and efficient encoding and decoding with XOR only operations. We briefly illustrate the basic idea in the encoding and decoding process. For complete descriptions, please refer to [3], [4]. Given k original symbols, a rateless-code encoder generates coded symbols on the fly, by performing exclusive-or operations on a *subset* of original symbols, randomly chosen based on a special degree distribution, such as the *Robust Soliton distribution* for LT codes. A *decoding graph* that connects coded symbols to input symbols is defined by the encoding process. The number of original symbols each coded symbol is generated from (*i.e.*, the "degree" of the coded symbol in the decoding graph) and the indices of these original symbols (neighbor indices in the decoding graph) are communicated to the receiver for the purpose of decoding, together with the coded symbol. At the decoder, it reconstructs the decoding graph. In each round of the decoding process, the decoder identifies a coded symbol of degree one, and recovers the value of its unique neighbor among the input symbols. Then the value of the recovered input symbol is exclusive-or'ed to the values of all its neighboring coded symbols, and all the incident edges are removed. Such a process is repeated until all the input symbols are recovered.

As a rateless code can potentially provide a nearly unlimited number of unique coded symbols, it further decreases the probability of block conflicts among upstream nodes in the parallel retrievals. Therefore, rateless codes are useful towards finding a solution to the content reconciliation problem. In the example shown in Fig. 2(c), from the four original blocks 1, 2, 3 and 4, S generates an unlimited number of coded blocks $1'$, $2'$, ..., with a rateless-code encoder. t_1 and t_2 are able to each obtain four unique coded blocks from S, thus reconciliation is not required at t_3. Unfortunately, content reconciliation may still be required at t_4, whose upstream peers t_1 and t_3 share common blocks $1'$ and $2'$.

B. Recoding with rateless codes

In order to completely eliminate the need for content reconciliation, we explore another desirable property of rateless codes. With rateless codes, the receiver may decode from output symbols generated by different rateless-code encoders, as long as they operate on the same set of input symbols with the same rateless code [4]. Based on this favorable property, we propose a recoding scheme to be carried out by each peer, such that freshly coded blocks are produced at each receiver and all the received blocks from any upstream nodes are unique and useful for decoding at a receiver.

In our protocol, the streaming source encodes the blocks of each media segment by a rateless code based on a special degree distribution, such as the LT code with the Robust Soliton Distribution [3], and streams the coded blocks. After a peer retrieves K_i coded blocks for segment s_i of $\widetilde{M}$, where $K_i = (1+\epsilon)k$, it decodes the K_i coded blocks and obtains the original k blocks. Upon retrieving requests from other peers, it generates new coded blocks from these original blocks by the same rateless-code encoder based on the same degree distribution, and delivers these new coded blocks. This *rateless recoding* protocol is summarized in Table 1, with the example of an LT code.

TABLE I
RECODING WITH RATELESS CODES AT EACH RECEIVER

After receiving K_i packets for segment s_i

Decode to obtain its k original blocks.

To serve another receiver q at rate y:

While q still needs new coded blocks

1. Generate coded block B_j^q from s_i's original blocks $b_i^1, b_i^2, \ldots, b_i^k$ by

(1.a) Randomly choose the degree d_j^q from the Robust Soliton distribution;

(1.b) Choose d_j^q distinct original blocks uniformly at random, and set B_j^q to be exclusive-or of these blocks.

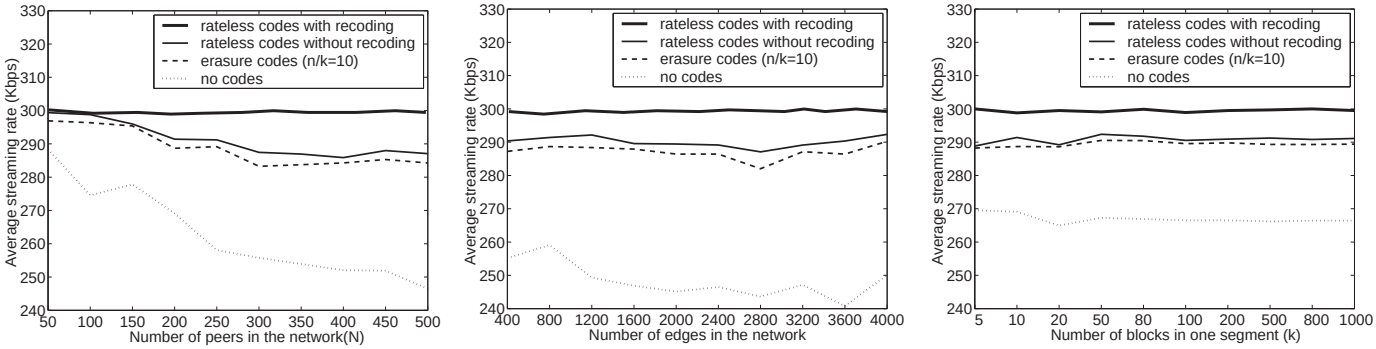
2. Packetize B_j^q into a packet together with the degree d_j^q and the set of neighbor indices.

3. Deliver the packet to q at rate y.

The following proposition proves the correctness of our recoding protocol.

Proposition 1. *The k original blocks of segment s_i in $\widetilde{M}$ can be recovered from any set of $(1+\epsilon)k$ coded blocks with the same high probability as the original code, in a peer-to-peer streaming session implementing the recoding protocol in Table 1.*

Proof: We present a brief outline of the proof. The coded blocks a receiver receives for recovering segment s_i are either coded by the streaming source or recoded by an upstream peer, both from the same set of k original blocks of s_i. Since all the encoders follow the same encoding steps and generate each block independently from any other one based on the same Robust Soliton distribution, the coded blocks are all potentially unique as if they are produced by a same encoder. Thus, after collecting $(1+\epsilon)k$ coded blocks from any upstream peers, a receiver can recover the k original blocks with the same high probability as the original code. □



(a) Comparison in random networks of different network sizes (N) and fixed edge density (4N)

(b) Comparison in random networks of a fixed network size (200 peers) and different edge densities

(c) Comparison among different block numbers in a random network of 200 peers and 800 edges

Fig. 3. Comparison of average streaming rates in large peer-to-peer streaming networks with 4 different coding schemes.

The rateless recoding protocol guarantees the uniqueness and equal usefulness of all the coded blocks in the session, thus completely eliminating the need for content reconciliation. In Fig. 2(d), for example, S generates a potentially unlimited number of blocks $1'$, $2'$, ... from the original blocks 1, 2, 3 and 4. The difference between Fig. 2(d) and (c) is that, rather than simply relaying received blocks, all peers recode the recovered original blocks and deliver the freshly coded blocks. After receiving blocks $1'$, $2'$, $3'$ and $4'$ from S, t_1 decodes them to derive 1, 2, 3 and 4, then it encodes them again into $1^{(1)}$, $2^{(1)}$, $3^{(1)}$, $4^{(1)}$, ..., upon requests from t_3 and t_4. Similarly, t_2 recovers the four original blocks from $5'$, $6'$, $7'$ and $8'$, and recodes to obtain $1^{(2)}$, $2^{(2)}$, $3^{(2)}$, $4^{(2)}$, ... Thus t_3 can safely retrieve unique coded blocks $1^{(1)}$, $2^{(1)}$, $1^{(2)}$, $2^{(2)}$ from t_1 and t_2. After decoding, t_3 further recodes to obtain unique blocks for delivery: $1^{(3)}$, $2^{(3)}$, $3^{(3)}$, $4^{(3)}$, ... Therefore, t_4 is able to concurrently retrieve blocks $1^{(3)}$, $2^{(3)}$, $3^{(1)}$, $4^{(1)}$ without reconciliation between t_1 and t_3.

C. Best answer to content reconciliation problem

To investigate the effectiveness of rateless recoding in eliminating delivery redundancy and maximizing bandwidth utilization, we now show a comparison study result among the four coding schemes in large scale networks in Fig. 3: no coding, fixed-rate erasure codes, rateless codes without recoding, and rateless codes with recoding. In this empirical study, we simulate a 300 Kbps live streaming session on random networks generated with BRITE [7] based on power-law degree distributions. We assume no constraints of peer upload/download capacities in this study, and heuristically assign transmission rates on the links to provide an aggregate receiving rate of 300 Kbps at each peer. Under each scheme, the media is streamed without content reconciliation among peers. At each receiver, duplicated and non-useful received blocks (for decoding) are eliminated, and the actual streaming rate is calculated.

We first fix k, the number of blocks in each media segment, to 50, and compare the four schemes in networks of different numbers of peers and various *edge densities* in Fig. 3(a) and (b). The results exhibit that only *rStream*'s rateless recoding scheme can actually achieve an average streaming rate around 300 Kbps at the receivers. In other schemes, the streaming rates are reduced at different degradation levels, caused by the duplication in the received blocks. In our investigation with fixed-rate erasure codes, we also noticed that increasing the rate n/k of the codes helps alleviate the conflicts at receivers. Nevertheless, this improvement is upper bounded by the results of the scheme of rateless encoding without recoding.

We next investigate the impact of the number of blocks in each media segment on the block conflicts at the receivers (Fig. 3(c)). Varying k from 5 to 1000, we find the average streaming rates remain approximately the same for each coding scheme. Thus, we conclude that by varying the number of blocks per segment, we are not able to alleviate the severity of block conflicts in those coding schemes where content reconciliation is required. All these exhibit the benefits of rateless recoding in eliminating delivery redundancy without the need of reconciliation.

D. Efficiency of rateless recoding

Finally, we discuss the efficiency of our rateless recoding scheme. As previously mentioned, rateless codes are highly computationally efficient. For the example of LT codes, it takes on average $O(\ln(k/\delta))$ block exclusive-or operations to generate a coded block from k original blocks, and $O(k\ln(k/\delta))$ block exclusive-or operations to recover the k original blocks from $k + O(\sqrt{k}\ln^2(k/\delta))$ coded blocks with probability $1-\delta$. Each block exclusive-or operation includes l bitwise exclusive-or operations. As exclusive-or operations can be implemented very efficiently, rateless codes can achieve a high encoding/decoding bandwidth, and thus they can always be used to encode and decode on the fly with the streaming process.

Besides the many advantages of using rateless codes, there exists additional overhead with the recoding process. First, a segment is recoded and relayed from an upstream peer only when it has been entirely received and successfully recovered at the upstream peer. Such additional delay to receive an entire segment is determined by the size of the segment, $k \cdot l$, and the streaming rate. Second, to decode a segment of k original blocks, additional bandwidths are required to send the extra ϵk coded blocks, where ϵ is $O(\ln^2(k/\delta)/\sqrt{k})$ for LT codes.

We can see that values of k and l play a significant role in deciding the efficiency of the recoding process. On one hand, k should be sufficiently large to guarantee the generation of a potentially unlimited number of coded blocks for each segment. In addition, it is usually more efficient to have larger values of l in practice as well, to achieve less overhead with respect to bookkeeping operations. On the other hand, the value of $k \cdot l$, the size of a media segment, may not be too large, in order to reduce the initial waiting time at each receiver. Furthermore, the encoding/decoding speed and extra block overhead of a rateless code are contingent upon k and l as well. We are going to explore and discuss results based on different values of these parameters with our *rStream* implementation in Sec. V.

III. Optimal Peer Selection and Rate Allocation

In this section, we seek to answer a question that is critical to any peer-to-peer streaming schemes: in a mesh peer-to-peer network, what is the best way to select upstream peers and allocate transmission rates of the rateless-code coded streams, such that a specified streaming bit rate is satisfied and continuous playback is guaranteed at all the receivers? We formulate the problem as a linear optimization problem, and then design an efficient distributed optimal rate allocation algorithm.

A. Linear programming formulation

In *live* media streaming sessions, it is desirable to achieve minimal end-to-end latencies at the receivers [8], [9]. With our rateless recoding, coding delays are introduced at each intermediate receiver, a tradeoff besides the many advantages of recoding. Therefore, it is especially important for each receiver to select the upstream peers close to the streaming source in terms of overlay hops, and to minimize the end-to-end link latencies along the paths. In our optimization problem, we aim to construct an optimal streaming topology, on top of which not only the streaming rate is satisfied, but also the end-to-end link latencies are minimized at all the receivers. We formulate the objective function to reflect the minimization of such latencies, and the constraints to represent the streaming rate requirement and capacity limitations in the network. In what follows, we motivate our linear program (LP) formulation of multicast peer-to-peer streaming by first analyzing a unicast streaming session from the streaming source to one receiver.

1) LP for unicast streaming: A unicast flow from the streaming source to a receiver is a standard network flow observing the property of flow conservation at each intermediate node. Let r be streaming rate of this unicast flow, c_{ij} be the link delay and f_{ij} be the transmission rate on overlay link (i,j). Fig. 4(a) depicts an example of a unit unicast flow from S to t_4, with $r = 1$, $c_{ij} = 1, \forall (i,j) \in A$[1], and the rates f_{ij} labeled on the arcs. Such a unicast flow can be viewed as multiple fractional flows, each going along a different overlay path. Different paths may share some same overlay links, and the transmission rate on each shared link is the sum of rates of all fractional flows that go through the link. Fig. 4(b) illustrates the decomposition of the unit unicast flow into three fractional flows, with rates 0.2, 0.3 and 0.5, respectively.

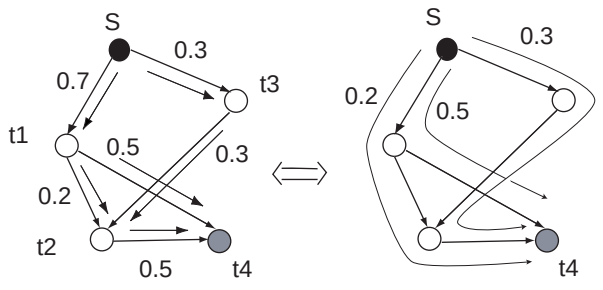


Fig. 4. An example of a unicast flow from S to t_4 and its decomposition into three fractional flows.

We calculate the average end-to-end link latency of a unicast flow as the weighted average of the end-to-end latencies of all its fractional flows, with the weight being the ratio of the fractional flow rate to the aggregate unicast flow rate. In Fig. 4, the end-to-end link delays of the three paths are 3, 3 and 2 respectively, and thus the average end-to-end latency is $0.2 \times 3 + 0.3 \times 3 + 0.5 \times 2$. We further notice that

$$\begin{aligned} & 0.2 \times (1+1+1) + 0.3 \times (1+1+1) + 0.5 \times (1+1) \\ = \; & 1 \times (0.2+0.5) + 1 \times 0.3 + 1 \times 0.2 \\ & + 1 \times 0.5 + 1 \times 0.3 + 1 \times (0.2+0.3) \\ = \; & 1 \times 0.7 + 1 \times 0.3 + 1 \times 0.2 + 1 \times 0.5 + 1 \times 0.3 + 1 \times 0.5 \\ = \; & \sum_{(i,j) \in A} c_{ij} f_{ij} / r. \end{aligned}$$

In general, we can prove $\sum_{(i,j)\in A} c_{ij} f_{ij}/r$ represents the average end-to-end link delay of a unicast flow, as given in the following proposition:

Proposition 2. *Let r be the streaming rate of a unicast session, c_{ij} be the link delay and f_{ij} be the transmission rate on link (i,j), $\forall (i,j) \in A$. $\sum_{(i,j)\in A} c_{ij} f_{ij}/r$ represents the average end-to-end link delay of this unicast flow.*

Proof: Let P be the set of paths from the streaming source to the receiver in the session. Let $f^{(p)}$ be the rate of the fractional flow going along path $p \in P$. The average end-to-end latency at the receiver is

$$\begin{aligned} \sum_{p \in P} \frac{f^{(p)}}{r} \Big(\sum_{(i,j):(i,j) \text{ on } p} c_{ij} \Big) &= \frac{1}{r} \sum_{(i,j)\in A} c_{ij} \Big(\sum_{p:(i,j) \text{ on } p} f^{(p)} \Big) \\ &= \frac{1}{r} \sum_{(i,j)\in A} c_{ij} f_{ij}. \end{aligned}$$

□

Next, we formulate a linear program to achieve minimum-delay unicast streaming. Let u_{ij} be the capacity of link (i,j). Omitting constant r, we use $\sum_{(i,j)\in A} c_{ij} f_{ij}$ to represent the average end-to-end link latency of the unicast flow and derive

$$\min \sum_{(i,j)\in A} c_{ij} f_{ij} \tag{1}$$

subject to

$$\begin{aligned} \textstyle\sum_{j:(i,j)\in A} f_{ij} - \sum_{j:(j,i)\in A} f_{ji} = b_i, & \quad \forall i \in N, \\ 0 \le f_{ij} \le u_{ij}, & \quad \forall (i,j) \in A, \end{aligned}$$

where

$$b_i = \begin{cases} r & \text{if } i = S, \\ -r & \text{if } i = t, \\ 0 & \text{otherwise.} \end{cases}$$

An intuitive thought about the optimization formulation might be that, the end-to-end latency of a unicast flow should be formulated as the maximum of the end-to-end latencies of all its fraction flows, instead of the average. However, we model the average end-to-end latency not only because we are able to derive the nice linear formulation in (1), which is indeed a standard minimum cost flow problem, but also since the minimization of the average end-to-end delay achieves the same results as the minimization of maximum end-to-end latency, *i.e.*, the rates are allocated in such a way that high latency links, such as satellite and transcontinental links, are avoided as much as possible. Besides, when c_{ij}'s ($\forall (i,j) \in A$) are of similar magnitude, we can understand the objection function as the minimization of the average hop count from the streaming source to the receiver.

We call the optimal unicast flow decided by this linear program

[1]Note that c_{ij}'s may well take different values. In this simple illustrative example, we make c_{ij}'s all equal for easier understanding.

TABLE II
LP FOR *rStream* PEER-TO-PEER STREAMING

P:

$$\min \sum_{t\in T}\sum_{(i,j)\in A} c_{ij} f_{ij}^t$$

subject to

$$\sum_{j:(i,j)\in A} f_{ij}^t - \sum_{j:(j,i)\in A} f_{ji}^t = b_i^t, \quad \forall i \in N, \forall t \in T \tag{2}$$

$$f_{ij}^t \geq 0, \quad \forall (i,j) \in A, \forall t \in T \tag{3}$$

$$f_{ij}^t \leq x_{ij}, \quad \forall (i,j) \in A, \forall t \in T \tag{4}$$

$$\sum_{j:(i,j)\in A} x_{ij} \leq O_i, \quad \forall i \in N \tag{5}$$

$$\sum_{j:(j,i)\in A} x_{ji} \leq I_i, \quad \forall i \in N \tag{6}$$

where

$$b_i^t = \begin{cases} r & \text{if } i = S, \\ -r & \text{if } i = t, \\ 0 & \text{otherwise.} \end{cases}$$

TABLE III
SUMMARY OF NOTATIONS

r	the required streaming rate of the session
c_{ij}	the link delay on overlay link (i,j), $\forall (i,j) \in A$
f^t	the conceptual flow from S to receiver t, $\forall t \in T$
f_{ij}^t	the rate of f^t on link (i,j)
x_{ij}	the transmission rate on link (i,j), $\forall (i,j) \in A$
O_i	the upload capacity at node i
I_i	the download capacity at node i

a *minimum-delay flow*. Such a minimum-delay flow is useful in modeling minimum delay multicast streaming in a peer-to-peer network, as a minimum-delay multicast streaming flow can be viewed as consisting of multiple minimum-delay flows from the source to each of the receivers. Here we make use of the concept of *conceptual flow* introduced in [10]. A multicast flow is conceptually composed of multiple unicast flows from the sender to all receivers. These unicast conceptual flows co-exist in the network without contending for link capacities, and the multicast flow rate on a link is the maximum of the rates of all the conceptual flows going along this link. For the example shown in Fig. 2, the multicast streaming flow from S to t_1, t_2, t_3 and t_4 can be understood as consisting of four conceptual flows from S to each of the receivers. When each conceptual flow is a minimum-delay flow, the end-to-end delays of the multicast session are minimized. Based on this notion, we proceed to formulate the optimal rate allocation problem for multicast peer-to-peer streaming.

2) LP for multicast peer-to-peer streaming: Our linear optimization model for *rStream* aims to minimize the end-to-end link delays from the source to all receivers. Based on the initial mesh topology decided by the neighbor assignment from the bootstrapping service, it optimally allocates the transmission rate on each overlay link to construct a minimum (link) delay *streaming topology*. In our formulation, we consider upload and download capacity constraints at each peer, rather than link capacity constraints. This comes from practical observations that bandwidth bottlenecks usually occur on "last-mile" access links at each of the peers in a peer-to-peer network. The linear program is formulated in Table II[2], with notations summarized in Table III.

In **P**, each conceptual flow f^t is a valid network flow, subject to constraints (2)(3)(4) similar to those in the LP in (1). The difference lies in that f_{ij}^t's, $\forall t \in T$, are bounded by the transmission rate x_{ij} on link (i,j), while x_{ij}'s are further restricted by upload and download capacities at their incident nodes.

An optimal solution to problem **P** provides an optimal rate ${f_{ij}^t}^*$ for the conceptual flow f^t on link (i,j), $\forall (i,j) \in A$. Let z be the optimal multicast streaming flow in the network. We compute the optimal transmission rates as:

$$z_{ij} = \max_{t\in T} f_{ij}^t, \quad \forall (i,j) \in A. \tag{7}$$

Such an optimal rate allocation $(z_{ij}, \forall (i,j) \in A)$ guarantees r at all the receivers, and achieves minimal end-to-end link latencies as well. At the same time, it computes an optimal peer selection strategy, *i.e.*, an upstream peer is selected at a receiver if the optimal transmission rate between them is non-zero.

Note that here we have formulated the optimal rate allocation problem for a single streaming session. In general, multiple streaming sessions may co-exist in the same network. In fact, **P** can be readily extended to model the multiple-session scenario, and the resulting LP is its multicommodity variant. Interested readers are referred to our previous work [11] for detailed discussions of the multiple session case.

B. Efficient subgradient solution

We now design an efficient distributed algorithm to solve the linear program **P**. General LP algorithms, such as the Simplex, Ellipsoid and Interior Point methods, are inherently centralized and costly, which are not appropriate for our purpose. Our solution is based on the technique of Lagrangian relaxation and subgradient algorithm [12], [13], which can be efficiently implemented in a fully distributed manner.

1) Lagrangian dualization: We start our solution by relaxing the constraint group (4) in **P** to obtain its Lagrangian dual. The reason of selecting this set of constraints to relax is that the resulting Lagrangian subproblem can be decomposed into classical LP problems, for each of which efficient algorithms exist. We associate Lagrangian multipliers μ_{ij}^t with the constraints in (4) and modify the objective function as:

$$\sum_{t\in T}\sum_{(i,j)\in A} c_{ij} f_{ij}^t + \sum_{t\in T}\sum_{(i,j)\in A} \mu_{ij}^t (f_{ij}^t - x_{ij})$$
$$= \sum_{t\in T}\sum_{(i,j)\in A} (c_{ij} + \mu_{ij}^t) f_{ij}^t - \sum_{t\in T}\sum_{(i,j)\in A} \mu_{ij}^t x_{ij}.$$

We then derive the Lagrangian dual of the primal problem **P**:

DP: $$\max_{\mu \geq 0} L(\mu)$$

where

$$L(\mu) = \min_{P} \sum_{t\in T}\sum_{(i,j)\in A} (c_{ij} + \mu_{ij}^t) f_{ij}^t - \sum_{t\in T}\sum_{(i,j)\in A} \mu_{ij}^t x_{ij} \tag{8}$$

[2]If we consider the extra bandwidth required by rateless-code coded streams, the aggregate receiving rate at each peer should be $(1+\epsilon)r$. In our LP formulation, we omit this slight difference. In our implementation, we take this into account and use $(1+\epsilon)r$ to compute the optimal rates.

and the polytope P is defined by the following constraints:

$$\begin{aligned} \textstyle\sum_{j:(i,j)\in A} f_{ij}^t - \sum_{j:(j,i)\in A} f_{ji}^t = b_i^t, &\quad \forall i \in N, \forall t \in T, \\ f_{ij}^t \geq 0, &\quad \forall (i,j) \in A, \forall t \in T, \\ \textstyle\sum_{j:(i,j)\in A} x_{ij} \leq O_i, &\quad \forall i \in N, \\ \textstyle\sum_{j:(j,i)\in A} x_{ji} \leq I_i, &\quad \forall i \in N. \end{aligned}$$

Here, the Lagrangian multiplier μ_{ij}^t can be understood as the link price on link (i,j) for the conceptual flow from source S to receiver t. Such interpretation should be clear as we come to the adjustment of μ_{ij}^t in the subgradient algorithm.

We observe that the Lagrangian subproblem in Eq. (8) can be decomposed into a maximization problem in (9),

$$\max \sum_{t\in T} \sum_{(i,j)\in A} \mu_{ij}^t x_{ij} \tag{9}$$

$$\begin{aligned} \text{subject to} \quad & \textstyle\sum_{j:(i,j)\in A} x_{ij} \leq O_i, \quad \forall i \in N, \\ & \textstyle\sum_{j:(j,i)\in A} x_{ji} \leq I_i, \quad \forall i \in N, \end{aligned}$$

and multiple minimization problems in (10), each for one $t \in T$,

$$\min \sum_{(i,j)\in A} (c_{ij} + \mu_{ij}^t) f_{ij}^t \tag{10}$$

$$\begin{aligned} \text{subject to} \quad & \textstyle\sum_{j:(i,j)\in A} f_{ij}^t - \sum_{j:(j,i)\in A} f_{ji}^t = b_i^t, \quad \forall i \in N, \\ & f_{ij}^t \geq 0, \quad \forall (i,j) \in A. \end{aligned}$$

We notice that the maximization problem in (9) is an inequality constrained transportation problem, which can be solved in polynomial time by distributed algorithms, *e.g.*, the Auction algorithm [14]. Each minimization problem in (10) is essentially a shortest path problem, which finds the shortest path to deliver a conceptual flow of rate r from source S to a receiver t. For the classical shortest path problem, efficient distributed algorithms exist, *e.g.*, Bellman-Ford algorithm, label-correcting algorithms [15] and relaxation algorithms [16]. As the algorithms are all essentially the same as Bellman-Ford algorithm, we employ the distributed Bellman-Ford algorithm [16], [17] as our solution.

2) Subgradient algorithm: We now describe the subgradient algorithm, applied to solve the Lagrangian dual problem **DP**. The algorithm starts with a set of initial non-negative Lagrangian multiplier values $\mu_{ij}^t[0]$, $\forall (i,j) \in A, \forall t \in T$. At the k^{th} iteration, given current Lagrangian multiplier values $\mu_{ij}^t[k]$, we solve the transportation problem in (9) and the shortest path problems in (10) to obtain new primal variable values $x_{ij}[k]$ and $f_{ij}^t[k]$. Then, the Lagrangian multipliers are updated by

$$\begin{aligned} \mu_{ij}^t[k+1] = \max(0, \mu_{ij}^t[k] + \theta[k](f_{ij}^t[k] - x_{ij}[k])), \\ \forall (i,j) \in A, \forall t \in T, \end{aligned} \tag{11}$$

where θ is a prescribed sequence of step sizes that decides the convergence and the convergence speed of the subgradient algorithm. When θ satisfies the following conditions, the algorithm is guaranteed to converge to $\mu^* = (\mu_{ij}^t, \forall (i,j) \in A, \forall t \in T)$, an optimal solution of **DP**:

$$\theta[k] > 0, \lim_{k\to\infty} \theta[k] = 0, \text{ and } \sum_{k=1}^{\infty} \theta[k] = \infty.$$

Eq. (11) can be understood as the adjustment of link price for each conceptual flow on each link. If the rate of the conceptual flow exceeds the transmission rate on the link, (4) is violated, so the link price is raised. Otherwise, the link price is reduced.

For linear programs, the primal variable values derived by solving the Lagrangian subproblem in Eq. (8) at μ^* are not necessarily an optimal solution to the primal problem **P**, and even not a feasible solution to it [18]. Therefore, we use the algorithm introduced by Sherali *et al.* [18] to recover the optimal primal values ${f_{ij}^t}^*$. At the k^{th} iteration of the subgradient algorithm, we also compose a primal iterate $\widehat{f_{ij}^t}[k]$ via

$$\widehat{f_{ij}^t}[k] = \sum_{h=1}^{k} \lambda_h^k f_{ij}^t[h], \forall (i,j) \in A, \forall t \in T \tag{12}$$

where $\sum_{h=1}^{k} \lambda_h^k = 1$ and $\lambda_h^k \geq 0$, for $h = 1, \ldots, k$. Thus, $\widehat{f_{ij}^t}[k]$ is a convex combination of the primal values obtained in the earlier iterations.

In our algorithm, we choose the step length sequence $\theta[k] = a/(b+ck), \forall k, a > 0, b \geq 0, c > 0$, and convex combination weights $\lambda_h^k = 1/k, \forall h = 1, \ldots, k, \forall k$. These guarantee the convergence of our subgradient algorithm; they also guarantee that any accumulation point $\widehat{f}^*$ of the sequence $\{\widehat{f}[k]\}$ generated via (12) is an optimal solution to the primal problem **P** [18]. We can thus calculate $\widehat{f_{ij}^t}[k]$ by

$$\begin{aligned} \widehat{f_{ij}^t}[k] &= \sum_{h=1}^{k} \frac{1}{k} f_{ij}^t[h] = \frac{k-1}{k} \sum_{h=1}^{k-1} \frac{1}{k-1} f_{ij}^t[h] + \frac{1}{k} f_{ij}^t[k] \\ &= \frac{k-1}{k} \widehat{f_{ij}^t}[k-1] + \frac{1}{k} f_{ij}^t[k]. \end{aligned}$$

C. Distributed algorithm

Based on the subgradient algorithm, we now design a distributed algorithm to solve **P**,given in Table IV. In practice, the algorithm to be executed on a link (i,j) is delegated by receiver j. Therefore, the algorithm is executed in a fully decentralized manner, in that each peer is only responsible for computation tasks on all its incoming links with only local information, *e.g.*, knowledge of neighbor nodes, delay on its adjacent links, etc.

TABLE IV
THE DISTRIBUTED OPTIMAL RATE ALLOCATION ALGORITHM

1. Choose initial Lagrangian multiplier values $\mu_{ij}^t[0]$, $\forall (i,j) \in A$, $\forall t \in T$.

2. Repeat the following iteration until the sequence $\{\mu[k]\}$ converges to μ^* and the sequence $\{\widehat{f}[k]\}$ converges to $\widehat{f}^*$:

At times $k = 1, 2, \ldots$, $\forall (i,j) \in A$, $\forall t \in T$

1) Compute $x_{ij}[k]$ by the distributed auction algorithm;

2) Compute $f_{ij}^t[k]$ by the distributed Bellman-Ford algorithm;

3) Compute $\widehat{f_{ij}^t}[k] = \frac{k-1}{k} \widehat{f_{ij}^t}[k-1] + \frac{1}{k} f_{ij}^t[k]$;

4) Update Lagrangian multiplier $\mu_{ij}^t[k+1] = \max(0, \mu_{ij}^t[k] + \theta[k](f_{ij}^t[k] - x_{ij}[k]))$, where $\theta[k] = a/(b+ck)$.

3. Compute the optimal transmission rate $z_{ij} = \max_{t\in T} {\widehat{f_{ij}^t}}^*$, $\forall (i,j) \in A$.

While the optimal transmission rates on the links can be computed, we argue that only by transmitting with our rateless recoding protocol, can the optimal rates be actually achieved. This is because these optimal rates may only be achieved at each receiver when media contents from its upstream peers are carefully reconciled and not duplicated. With *rStream*'s rateless recoding, delivery redundancy is eliminated and thus the available

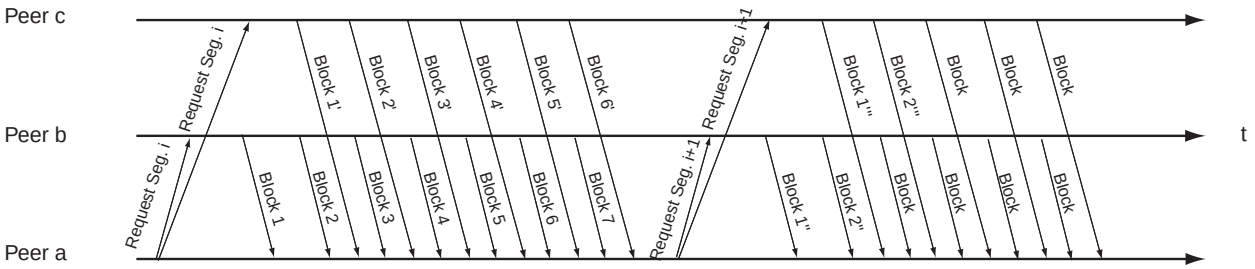


Fig. 5. Media distribution protocol: the pull-push paradigm.

bandwidth can be indeed saturated. We further note that in streaming, there is only one flow on each link (i, j), composed of coded blocks delivered at the optimal rate of z_{ij}. The introduction of conceptual flows is only to facilitate our optimization formulation and the derivation of optimal rates z_{ij}'s over the links. Thanks to rateless recoding, we are exempted from the elaborate media assignment required for each flow if no coding is applied, but are able to saturate the limited bandwidth capacities.

IV. rStream: the complete protocol

Combining the optimal rate allocation algorithm with the rateless recoding protocol, *rStream* represents a complete peer-to-peer streaming solution, which guarantees resilience and optimality at the same time. In what follows, we present the practical *rStream* media distribution protocol and the protocol to handle peer dynamics, based on the two components.

A. Media distribution

In *rStream*, the media segments are disseminated with a hybrid of pull and push methods.

At each receiver, it maintains a receiving range of the media segments, typically from the next media segment to play to a number of future segments beyond. The segment retrieval strategy, *i.e.*, which segment to retrieve next from each upstream peer, is decided at the downstream peer, representing the "pull" part of the distribution protocol. While such retrieval strategy design is key in a media streaming protocol without coding, this task is significantly alleviated in *rStream*. When receiver v is deciding the *available* segment to retrieve from upstream peer u, a segment that has been successfully decoded at u, it selects a segment that is in its receiving range with the nearest playback time. Due to rateless recoding, even if v concurrently downloads coded blocks for a same segment from multiple upstream peers, the received blocks can all be used for decoding the segment. Therefore, not only the need for reconciling blocks inside each segment is eliminated, but also the reconciliation for the retrieval of segments from different upstream peers becomes unnecessary. We further consider that an upstream peer may not have any segments available in the receiving range of a receiver. In this case, the downstream peer temporarily increases its streaming rates from other upstream peers that have available segments to serve it.

Upon receiving a request for a specific segment, an upstream peer starts generating coded blocks of the segment, and continuously *pushes* them to the requesting peer at the optimal rate computed by the optimal rate allocation. The receiver passes received blocks onto its decoder and decodes on the fly. When all the original blocks in the segment have been recovered, the receiver sends a "stop" signal to all its upstream peers that are serving it this segment, and requests for new segments selected with the segment retrieval strategy discussed above. Then a respective upstream peer terminates the push of the current segment, and starts delivering coded blocks for the new segment. This pull-push paradigm is illustrated in Fig. 5, where peer a first requests segment i from upstream peers b and c, and then segment $i + 1$ after i is successfully decoded.

With the simple segment pulling strategy and block pushing method inside each segment, we minimize the control message overhead, and more importantly, the streaming delays, as compared to schemes which pull individual blocks. Meanwhile, having each peer explicitly ask selected upstream peers for segments, our protocol provides the flexibility for each peer to reconfigure its connectivity in case of network dynamics. Finally, we emphasize that this efficient pull-push paradigm is only achievable with rateless recoding, as every pushed block is useful for decoding.

B. Handling peer dynamics

In peer-to-peer streaming, peers may arbitrarily join a streaming session at any time, and may depart or fail unexpectedly. In *rStream*, the distributed optimal rate allocation algorithm is invoked and executed in a dynamic manner, and the peer connectivity is reconfigured with adjusted rates with peer dynamics.

1) Peer joins: In *rStream*, a new peer is admitted into a streaming session only if its download capacity is no lower than the required streaming rate r. It then immediately starts streaming with the available upload capacities acquired from its upstream peers, assigned by the bootstrapping service. Meanwhile, it sends a request to the streaming source, asking for computation of new optimal rate allocation on the links.

2) Peer departures and failures: During streaming, when a peer detects the failure or departure of an upstream peer, it attempts to acquire more upload bandwidth from its remaining upstream peers. Only when the peer fails to acquire the required streaming rate, it sends a re-calculation request to the source for the new optimal rate allocation.

At the source, when the number of received re-computation requests exceeds a certain threshold, the source broadcasts such a request, such that all peers activate a new round of distributed algorithm execution, while continuing with their own streaming at the original optimal rates. Note that in such a dynamic environment, a new round of algorithm execution always starts from the previously converged optimal rates, rather than from the very beginning when all the values are zero, thus expediting its convergence. The peers adjust their rates to the new optimal values after the rate allocation converges.

We conclude with the note that, this simple and efficient process of handling peer dynamics is only achievable with rateless recoding. As rateless recoding guarantees that all coded blocks in the session are useful, we can rest assured that all additionally acquired or newly allocated bandwidths can be fully used to deliver useful blocks for decoding, without the need of

reconciliation. In addition, the dynamic execution of optimal rate allocation algorithm provides optimal streaming topologies at any time. Working together, they provide excellent failure resilience and streaming delay minimization at the same time.

V. Performance evaluation

In this section, we present results from extensive experiments in emulated peer-to-peer streaming environments, based on our implementation of the *rStream* protocols with the C++ programming language. Our implementation includes the rateless-code encoder and decoder at each peer, message switching and buffer management in the application layer to implement the *rStream* protocols, as well as distributed optimal rate allocation. Our implementation also supports the measurement and emulation of network parameters, *e.g.*, node upload/download capacities. It compiles and runs in all major UNIX variants (such as Linux and Mac OS X), and uses standard Berkeley sockets to establish TCP/UDP connections between peers. All our experiments are conducted on a high-performance cluster consisting of 50 Dell 1425SC and Sun v20z dual-CPU servers, each equipped with dual Intel Pentium 4 Xeon 3.6GHz and dual AMD Opteron 250 processors.

A. Rateless-code encoder and decoder

1) Optimized decoder design: In our implementation, we have implemented LT codes based on the Robust Soliton distribution. In addition to the basic functions in LT codes [3], our decoder implementation represents two new designs, which contribute to significant improvements of decoding latency.

First, the decoding is "streamable," in the sense that the decoding graph is constructed incrementally and original blocks are recovered on the fly with the data transmission. Whenever a new coded block arrives at the decoder, its encoding information (degree and indices of original blocks that it is generated from) is added to the decoding graph, and original blocks are recovered as soon as enough information has been received to decode them. As compared to the traditional implementation which decodes a segment of k original blocks after $(1+\epsilon)k$ coded blocks have all arrived, our "streamable" decoder represents three advantages: (1) It maximally utilizes the time that it waits for new incoming blocks to construct the decoding graph and decode, thus minimizing the extra decoding delay during streaming; (2) The encoding information of received blocks is accumulated, and no re-computation of such information occurs during decoding. In the traditional implementation, if the first decoding attempt with $(1+\epsilon)k$ coded blocks fails, the entire decoding graph needs to be constructed again when additional blocks are received; (3) In combine with our pull-push media distribution scheme, the "streamable" decoder minimizes the delay between reception of the last received block used for decoding and the successful recovery of a segment, thus minimizing the number of non-used coded blocks injected into the network from upstream peers.

Second, the decoding graph is implemented with the most efficient double-linked dynamic data structures in C++. All insertion and removal operations are achieved in $O(1)$ time, instead of $O(k)$. This further improves the efficiency of the decoders, which is also critical to improve the performance of *rStream* protocol.

2) Performance of LT-code encoder/decoder: We now present evaluation results with our LT-code implementation. The experiments are divided into two parts.

Exp. 1. We first examine the net encoding and decoding speed by continuously feeding original/coded blocks into the encoder/decoder. Table V-VIII show the speed obtained with various values of coding parameters. Again, k is the number of original blocks in each segment and l is block size. c and δ are parameters in the Robust Soliton distribution of LT codes [3], where δ is the allowed failure probability for decoding from $k + O(\sqrt{k}\ln^2(k/\delta))$ coded blocks, and c is a positive constant which, together with δ, decides the probability of deriving a low encoding degree with the Robust Soliton Distribution. As c increases and δ decreases, more low degree coded blocks are generated.

Our results in Table V and VII reveal an increasing trend for encoding and decoding speed as c increases and δ decreases. As more low degrees are generated with the Robust Soliton distribution in this case, encoding/decoding involves fewer blocks and requires fewer XOR operations. Fixing c and δ, results in Table VI and VIII show that the speed decreases with the increase of k (as the block degree increases), and increases with the increase of l (as bookkeeping overhead decreases). Nevertheless, the achieved coding bandwidth in all cases is much larger than overlay link capacities. We can now conclude that, with our "streamable" decoder implementation, there is little recoding delay introduced to streaming, which will be further validated with experiments in Sec.V-B.2.

Exp. 2. We next investigate the average number of coded blocks needed to successfully decode a segment, which are received from one or multiple upstream peers. Results are given as multiples of k, *i.e.*, $1+\epsilon$, in Table IX and X. We observe that $1+\epsilon$ decreases with the increase of δ and k. For fixed coding parameter values, results remain the same no matter the coded blocks are generated at one or multiple upstream peers. This validates proposition 1 in Sec. II-B, *i.e.*, coded blocks produced by different LT-code encoders are equally useful.

From the above results, we conclude that the value of parameter k represents a tradeoff between encoding/decoding speed and bandwidth consumption to deliver sufficient coded blocks for decoding. Considering that the encoding/decoding speed is always much higher than regular media streaming rates, larger values of k are more appropriate to reduce bandwidth consumption. Besides, larger values of l provide higher encoding/decoding speed. Nevertheless, $k \cdot l$, the segment size, also affects the end-to-end delay at peers in the network, to be further discussed in Sec. V-B.2.

B. Performance of rStream Streaming

We now evaluate the *rStream* protocol implementation in emulated streaming environments. In order to emulate realistic networks, all the initial mesh topologies used in our experiments are random networks generated with power-law degree distributions with the BRITE topology generator [7]. In each network, a 300 Kbps media bitstream is streamed from a streaming source with 10 Mbps of upload capacity. We consider two classes of receivers: ADSL/cable modem peers and Ethernet peers. Unless otherwise stated, ADSL/cable modem peers take 70% of the total population with download capacities in the range of 512 Kbps - 3 Mbps and upload capacities in the range of $200-800$ Kbps, and Ethernet peers take the other 30% with both upload and download capacities in the range of $3-8$ Mbps. Overlay link delays are sampled from the distribution of pairwise ping times between PlanetLab nodes [19].

TABLE V
ENCODING SPEED (MBPS): K = 100, L = 1KB

c \ δ	0.01	0.05	0.1	0.5	1.0
0.01	35.65	37.37	37.84	38.26	37.52
0.05	25.53	27.39	28.20	32.68	34.02
0.1	31.19	31.44	31.93	33.26	34.51
0.5	75.46	60.25	59.23	50.82	50.90
1.0	119.78	108.29	103.40	73.65	69.15

TABLE VI
ENCODING SPEED (MBPS): C = 0.05, δ = 0.1

l \ k	10	50	100	500	1000
200	62.98	30.19	26.00	18.85	16.11
500	67.06	32.26	27.52	19.98	17.66
1K	68.96	32.87	28.20	20.32	18.16
2K	68.81	33.37	29.09	20.69	18.54
3K	69.13	34.16	29.21	20.81	18.59

TABLE VII
DECODING SPEED (MBPS): K = 100, L = 1KB

c \ δ	0.01	0.05	0.1	0.5	1.0
0.01	28.33	29.32	30.53	29.60	28.34
0.05	16.67	19.77	21.71	26.52	28.28
0.1	19.69	21.56	23.45	26.34	29.34
0.5	50.53	48.63	46.24	44.43	41.68
1.0	53.70	51.61	47.70	46.71	44.25

TABLE VIII
DECODING SPEED (MBPS): C = 0.05, δ = 0.1

l \ k	10	50	100	500	1000
200	44.39	18.17	15.79	10.92	8.40
500	58.89	22.62	19.17	13.74	11.63
1K	61.80	24.35	21.71	14.70	13.16
2K	66.11	24.80	22.65	16.30	14.75
3K	68.76	25.00	22.71	16.69	14.93

TABLE IX
NUMBER OF CODED BLOCKS FOR DECODING AS MULTIPLES OF k: C = 0.05, δ = 0.1, L = 1KB (N_{up}: NUMBER OF UPSTREAM PEERS)

N_{up}\k	10	50	100	500	1000
1	1.49	1.39	1.26	1.20	1.16
2	1.48	1.39	1.27	1.22	1.15
5	1.47	1.41	1.27	1.20	1.17
10	1.48	1.39	1.26	1.21	1.15
20	1.47	1.40	1.27	1.20	1.19

TABLE X
NUMBER OF CODED BLOCKS FOR DECODING AS MULTIPLES OF k: K = 100, L = 1KB, NUMBER OF UPSTREAM PEERS = 1

c \ δ	0.01	0.05	0.1	0.5	1.0
0.01	1.34	1.36	1.34	1.35	1.33
0.05	1.50	1.38	1.26	1.25	1.21
0.1	1.62	1.48	1.37	1.26	1.21
0.5	1.72	1.49	1.46	1.34	1.33
1.0	3.12	2.83	2.74	1.99	1.88

1) Control message overhead: We first investigate the control message overhead in *rStream*'s distribution protocol. In this set of experiments, we stream a 20-minute 300 Kbps media stream in networks of different sizes and edge densities, with various media segment sizes, i.e., $k \cdot l$. Fig. 6(A) exhibits the ratio of total control message sizes (in Bytes) over the overall media data message sizes (in Bytes) during the entire streaming period in each network. Fig. 6(B) further illustrates the average control messaging bandwidth consumed at each peer, calculated by $\frac{\textit{total size of control messages sent at a peer}}{\textit{streaming time at the peer}}$.

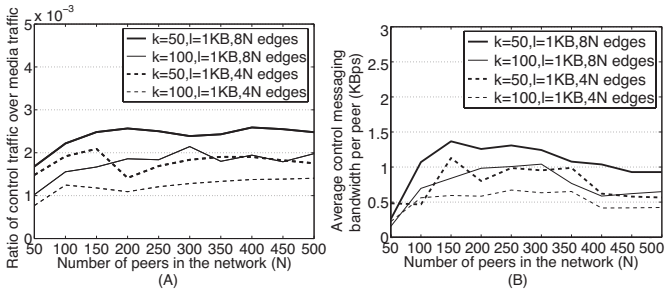


Fig. 6. Control message overhead in random networks of different sizes.

The results reveal that our pull-push mechanism involves very low control message overhead. In details, we observe that (1) the overhead remains at similar levels for networks of different sizes, showing good scalability of the pull-push mechanism; (2) the overhead increases with the increase of edge densities in networks of the same size, as more control messages are required when each peer has more upstream peers; and (3) the overhead decreases with the increase of media segment size, $k \cdot l$, as control messages are sent per segment basis, and there are fewer segments in total if the segment size is larger. In general, the control message overhead is non-significant as compared to media traffic.

2) End-to-end delay: We next investigate the optimality of the streaming topologies derived with *rStream*'s optimal rate allocation algorithm, and the actual end-to-end delay experienced at the peers when streaming over such optimal topologies, including recoding delay at intermediate peers, link latencies, etc. The distributed optimization algorithm, as summarized in Table IV, is carried out at the background during the streaming process, and rates are adjusted after the computation converges.

Exp. 1. As a natural first step, we start our investigations by studying the convergence performance of the distributed optimization algorithm.

Fig. 7 shows the convergence speed when the distributed algorithm runs from the beginning with all flow rates initialized to 0, in various static networks. With respect to the number of iterations executed by the iterative algorithm to converge to optimality, Fig. 7(A) shows that the number slowly increases with network sizes, and remains at the same level in a fixed-sized network with different edge densities. Fig. 7(B) further illustrates the running time for these iterations. As a fully distributed algorithm, its each iteration takes a similar length of time regardless of the network sizes, and therefore, the time to convergence only increases slowly with the increase of number of iterations executed.

When we consider the practical scenario that a convergence to absolute optimum is usually not necessary in realistic applications, we further investigate the convergence speeds to feasibility and 90% optimality, and compare them with that to optimality in Fig. 8. We observe that the convergence to the primal feasible solution is usually much faster, and even the convergence to a feasible solution, which achieves 90% optimality, is 20% faster.

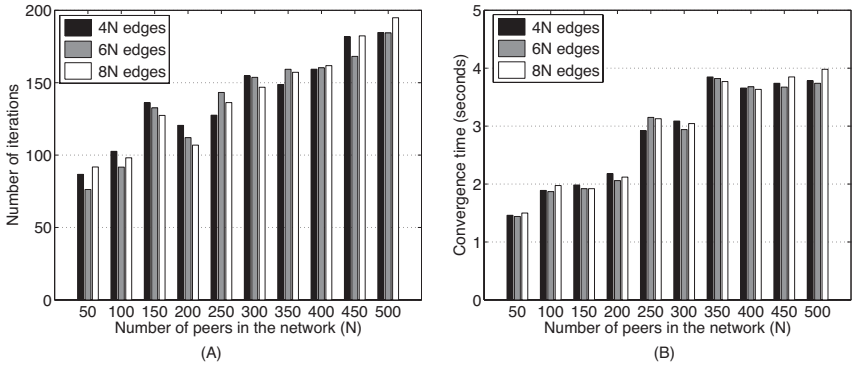


Fig. 7. Convergence speed in static networks.

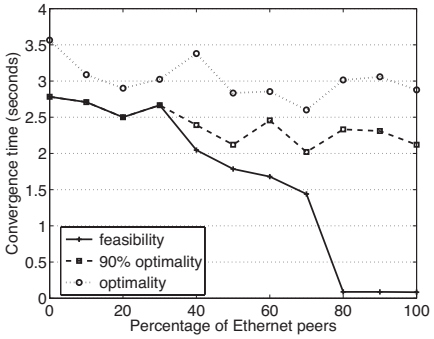


Fig. 8. Convergence time in static networks of 300 peers and 2400 edges.

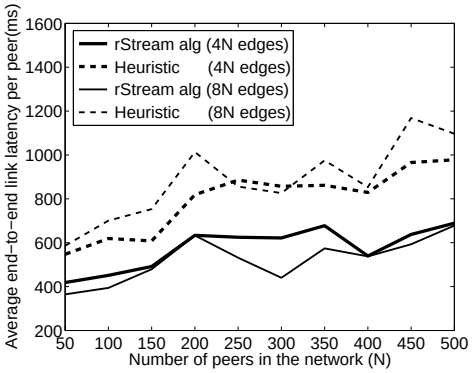


Fig. 9. Average end-to-end link latency: a comparison between rStream and a peer selection heuristic.

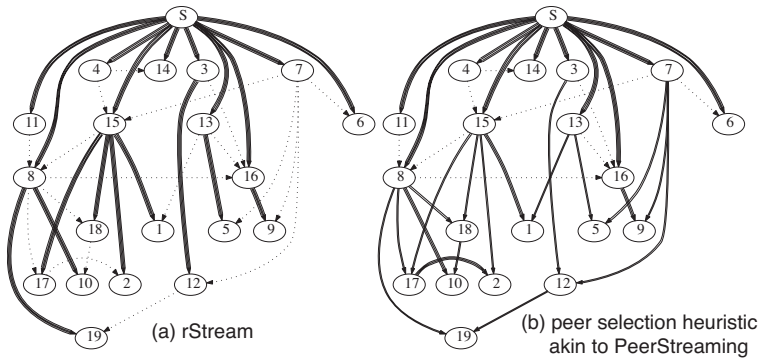


Fig. 10. Peer-to-peer streaming topologies of 20 peers: a comparison.

In addition, the time for convergence to feasibility drops quickly with the increase of Ethernet peer percentages, which brings more abundant upload capacities into the system.

Based on these observations, we conclude that in practical applications, the algorithm can obtain a feasible solution to a certain degree of optimality in a short time, even when it runs from the very beginning in large networks. In our evaluation of dynamic scenarios in Sec. V-B.3, we will show that the convergence to new optimum from previous ones is even faster.

Exp. 2. We now investigate the optimality of the streaming topologies obtained with the optimal rate allocation algorithm, by comparing them with those constructed with a commonly used peer selection heuristic [2], [20]. In this set of experiments, we only consider end-to-end link latencies on the topologies, as used as optimization objective of our algorithm, and leave the investigation of overall end-to-end streaming delays to the next experiment.

In the peer selection heuristic, each receiver distributes the required streaming rate among its upstream peers in proportion to their upload capacities. Its end-to-end link latency is calculated as the weighted average of the end-to-end link delays of flows from all its upstream peers, and the weight for each flow is the ratio of the assigned transmission rate from the upstream peer to the required streaming rate.

Fig. 9 exhibits that the *rStream* optimal rate allocation algorithm achieves much lower latencies than the heuristic, in networks of various sizes and edge densities. When the initial input topology is dense with more edges per peer, the optimization algorithm converges to better optimal topology with smaller end-to-end link latencies, while the heuristic results in higher latency in denser networks. This reveals that when there are more choices of upstream peers in a denser network, our algorithm can always find the best set of upstream peers on low delay paths. Therefore, in realistic peer-to-peer streaming networks with high edge densities, the advantage of our algorithm is more evident over the commonly used heuristic.

To visualize the optimal topology, Fig. 10 illustrates example streaming topologies derived with both algorithms from a same 20-peer input topology. In these graphs, distances between pairs of peers represent link latencies, and the widths of edges show the transmission rates on them. The dotted lines represent links that are not used in the resulting streaming topologies. It can be seen that by our optimal peer selection, receivers are streaming from the best upstream peers with minimal end-to-end latencies, while with the heuristic, peers simply distribute their streaming rate among upstream peers, which may lead to large end-to-end latencies.

Exp. 3. We next investigate overall end-to-end streaming delays experienced at the receivers when streaming over the derived optimal topologies, including encoding/decoding delays at intermediate peers, transmission delays, and link latencies. The overall end-to-end delay at each peer to receive a segment is measured as the difference between the time when the streaming source starts to generate and deliver coded blocks for a segment and the time when the peer has successfully decoded the segment.

In *rStream*, the overall end-to-end streaming delay at each peer is decided by the number of overlay hops that the peer is away from the source, as well as the delay on each hop. With respect to the former, in our optimal streaming topology construction, the diameter of a topology with N peers is $O(\log(N))$, leading to

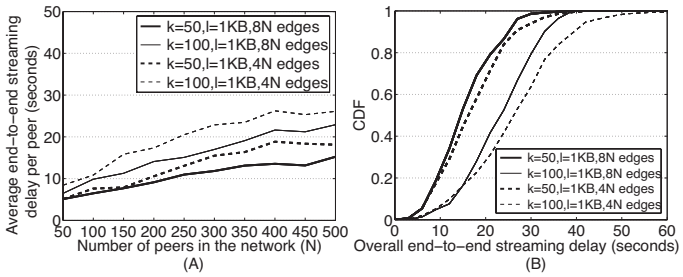


Fig. 11. End-to-end streaming delay in random networks. (A) Average peer end-to-end streaming delay in networks of different sizes. (B) Cumulative density function of peer end-to-end streaming delays in a 500-peer network.

log-scale increment of the delay with network size in Fig. 11(A). Fig. 11(A) also shows that the delay is smaller with a higher edge density. This is because with *rStream*'s optimal rate allocation, a peer has higher probability to be closer to the source in terms of hop counts if it knows more neighbors.

With respect to delay on each hop, it consists of decoding and encoding delay at the upstream peer, transmission delay at the scale of kl/r, and the overlay link delay. Based on our optimized decoder design and encoding/decoding efficiency shown in Sec. V-A.2, the recoding delay is non-significant, as compared to other delays. With various values of k and l — thus various transmission delay on each hop — Fig. 11(A) further exhibits the effect of media segment sizes on the end-to-end streaming latency, *i.e.*, the larger each segment is, the higher the end-to-end delay is.

Zooming into a 500-peer network, Fig. 11(B) plots CDFs of end-to-end streaming delay distribution at the participating peers. We observe that the delay at most peers is moderate, and only a small portion of peers experience a relatively longer delay.

From the above results, we see that k and l should be kept at moderate values in order to guarantee small end-to-end delays at the peers. Combining this point with our previous discussions, we conclude that choices of k and l represent another tradeoff between bandwidth consumption and end-to-end streaming delay in streaming sessions. Therefore, if a streaming session has stringent delay restrictions but presents lower bandwidth demand, we may choose relatively small values for k and l; otherwise, it is more appropriate to use a large k and a large l.

3) rStream streaming during peer dynamics: A consistent streaming rate is critical to guarantee smooth playback at the peers throughout the streaming session. Working together, the dynamic execution of optimal rate allocation algorithm and the rateless recoding scheme provide such streaming rate stability in *rStream*.

Exp. 1. In order to show its practicality in dynamic scenarios, we first investigate the convergence speed of the optimization algorithm in dynamic networks. In this experiment, during a 45-minute streaming session, 300 peers sequentially join the session in the first 20 minutes, and then start to depart from 25 minutes on. The distributed optimal rate allocation algorithm is invoked about every 15 peer joins or departures, and always runs from the previous optimal flow rates, following the dynamic execution method described in Sec. IV-B.

The number of additional iterations and time needed to converge to the new optimal rates in both the peer joining phase and departure phase are illustrated in Fig. 12. The results reveal that the convergence to new optimal rates in such dynamic scenarios is much faster, as compared to running from the very beginning

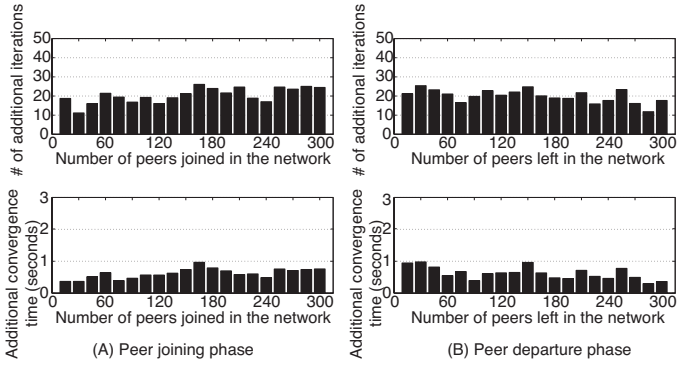


Fig. 12. Convergence speed in a dynamic network with up to 300 peers.

in the case of static networks of the same sizes. Independent of the current network size, the algorithm always takes less than 30 iterations ($1-2$ seconds) to converge.

We note that although this is a specially designed dynamic case, it reflects the capability of the optimization algorithm to converge promptly from one optimum to another in practical dynamic scenarios. In a realistic peer-to-peer streaming network, peer joins and departures may occur concurrently and consistently during the entire streaming session. In this case, our algorithm always improves the rate allocation towards optimality in the current network and can converge quickly as long as there exists a stable operating point in the dynamic overlay.

Exp. 2. We next investigate a more practical dynamic scenario where peer joins/departures occur concurrently and consistently. In this set of experiments, a 300 Kbps media streaming session is emulated in a 200-peer dynamic network. The peers join and depart following an On/Off model, with On/Off intervals both following an exponential distribution with an expected length of T seconds. Each peer is bootstrapped with D upstream peers upon joining. The optimal rate allocation algorithm is dynamically executed at background to adjust the streaming rates.

To investigate the smoothness of *rStream* streaming, we monitor two rate metrics at the peers during a 45-minute period of time: (1) *throughput*, which is the aggregate rate of receiving coded media bitstreams at a peer, computed by $\frac{l \times \textit{no. of coded blocks received in } t}{t}$, and (2) *goodput*, representing the aggregate rate of deriving original media contents, *i.e.*, the actual streaming rate, computed by $\frac{l \times \textit{no. of original blocks decoded in } t}{t}$.

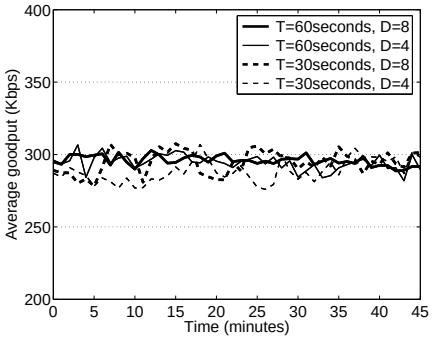


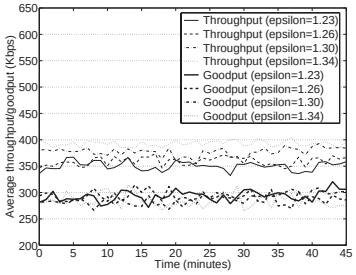
Fig. 13. Average goodput in a dynamic streaming session with 200 peers: $k = 100$, $l = 1KB$, $c = 0.05$, $\delta = 0.1$.

We first plot the average goodput achieved at peers in Fig. 13, with varying peer churn rates and network edge densities, and

fixed coding parameter values at $k = 100$, $l = 1KB$, $c = 0.05$, and $\delta = 0.1$. We observe no significant streaming rate fluctuations in all scenarios, *i.e.*, the average goodput is steadily maintained around 300 Kbps. In the case that each peer joins/leaves every 30 seconds, in the 200-peer network, there are $6 - 7$ peer joins/departures every second. Even with such consistent peer churns, the streaming rates at existing peers remain rather satisfactory at all times. In addition, we observe that the goodput is more stable when peers have more neighbors. These results demonstrate the excellent dynamic resilience of *rStream*, supported by its dynamics handling protocol discussed in Sec. IV-B.

k	l	c	delta	epsilon
200	500B	0.05	0.1	1.23
100	1KB	0.05	0.1	1.26
200	500B	0.01	0.1	1.30
100	1KB	0.01	0.1	1.34

(A)



(B)

Fig. 14. Average throughput/goodput in a dynamic streaming session with 200 peers: $T = 60$ seconds, $D = 4$.

Fixing the peer On/Off interval length to 60 seconds and upstream peer number to 4, we further investigate the throughput and goodput achieved at the peers at various coding parameter values in Fig. 14. As discussed in Sec. V-A.2, the values of four coding parameters, k, l, c and δ, decide ϵ, a factor representing extra bandwidth needed to delivery the additional coded blocks for decoding each media segment. We experimented with four sets of coding parameter values, and the corresponding ϵ values are given in Fig. 14(A). We further note that the value of ϵ reflects the *average* percentage of extra coded blocks required for decoding under each set of coding parameter values, and in actual streaming, the number of coded blocks used for successful recovery of each segment varies. Nevertheless, Fig. 14(B) exhibits that with each set of coding parameters, the achieved goodput is consistently around $r = 300$ Kbps, while the throughput is always near $(1 + \epsilon)r$. Such streaming rate stability reveals the fact that the slight variation in the number of coded blocks used to decode different segments actually introduces little jitter in the streaming.

VI. Related Work

Earlier work on peer-to-peer multimedia streaming has been based on a single multicast tree [21], [22], rooted at the streaming source, and constructed with a minimized height and a bounded node degree. The challenge, however, surfaces when interior peers in the tree do not have sufficient available capacities to upload to multiple children nodes, and when they depart or fail, which interrupts the streaming session and requires expensive repair processes.

Streaming based on multiple multicast trees has been proposed to address this problem, as in CoopNet [23] and SplitStream [24]. The media can be split into multiple sub-streams and each sub-stream is delivered along a different multicast tree. As a result, these systems accommodate peers with heterogeneous bandwidths by having each peer join different numbers of trees. It also is more robust to peer departures and failures, as an affected receiving peer may still be able to continuously display the media at a degraded quality, while waiting for the tree to be repaired. These advantages come with a cost, however, as all the trees need to be maintained in highly dynamic peer-to-peer networks. *rStream* uses a combination of rateless codes and mesh topologies to provide resilience and flexibility as well, but without the costs of explicit tree maintenance.

Similar to *rStream*, there have recently emerged a number of peer-to-peer streaming proposals that use mesh topologies, *e.g.*, CoolStreaming [2], Chainsaw[25], GridMedia[26], and PeerStreaming [20]. In these proposals, each peer periodically exchanges media availability with its neighbors, and the media segments to be retrieved from each neighbor are dependent upon the number of potential suppliers for each segment and the available upload capacities of neighbors. In PeerStreaming[20], the media downloading load is distributed among the set of supplying peers in proportion to their upload capacities. Compared to *rStream*, these heuristics fall short of achieving optimality, and may starve the peers with high download demands. *rStream* also considers the heterogeneity of link delays, which has not been previously taken into consideration.

In all mesh-based proposals, the need for content reconciliation naturally arises. Byers *et al.* [1] provide algorithms for estimation and approximate reconciliation of sets of symbols between pairs of collaborating peers. These algorithms may be very resource intensive with respect to both computation and messaging. Although Byers *et al.* advocate Tornado codes to provide reliability and flexibility, the sets of coded symbols acquired by different peers are still likely to overlap, as Tornado codes are not rateless. PeerStreaming [20] also employs a high rate erasure code, which is a modified Reed-Solomon code on the Galois Field GF(2^{16}), and ensures with high probability that the serving peers hold parts of the media without conflicts. PROMISE [27] uses Tornado codes to tolerate packet losses and peer dynamics, and performs rate assignment of the coded streams to a selected set of supplying peers. By applying these erasure codes with fixed rates, the need for content reconciliation is mitigated, but not eliminated. In comparison, by using rateless codes and recoding, *rStream* completely excludes any necessity for content reconciliation among peers.

Maymounkov *et al.* [5] use online codes, which belong to the class of rateless codes, to download large-scale content in peer-to-peer networks. It takes advantage of the benefits of rateless codes. However, by only encoding at the source peer and not recoding at the relaying peers, there may still be significant content overlap between a pair of relay peers, when they download from the same upstream peer. Further, it may not be readily applied to peer-to-peer streaming, as media streams are delay-sensitive and may be generated on-the-fly (*e.g.*, live streaming). Huang *et al.* [28] design an on-demand media streaming scheme based on the combination of media segmentation and rateless encoding with Raptor codes. However, they mainly utilize the higher encoding and decoding efficiency property of such rateless codes, but do not explore the useful properties related to their ratelessness.

With respect to peer selection, most existing work employs various heuristics without formulating the problem theoretically, with only one exception: Adler *et al.* [29] propose linear programming models that aim at minimizing costs in peer-to-peer content distribution. *rStream* is tailored to the specific requirements of media streaming, by minimizing streaming latencies. In [29], i supplying peers are allowed to fail, by having constraints guaranteeing the aggregate rate from any subset composed of $n-i$

supplying peers out of the total n is larger than the streaming rate. *rStream* handles peer failures based on the combination of rateless recoding and dynamic execution of optimal rate allocation. In addition, the problem of content reconciliation is not addressed in [29], which is the motivating factor towards the use of rateless codes in *rStream*.

VII. Conclusion

We conclude this paper by reinforcing our strong argument that rateless codes are ideal companions to peer-to-peer streaming solutions, and are orthogonal to any multimedia codecs, including H.264/AVC. A typical multimedia stream, such as an MPEG-4 or H.264 stream, can be treated as a bitstream demanding a constant bit rate, which can be segmented and treated by rateless codes. Using examples, analysis, and experiment results, we have made it very clear that rateless codes represent our best possible option to provide resilience to dynamics typically found in peer-to-peer networks, and to completely eliminate the need for content reconciliation. Combined with optimal peer selection and rate allocation strategies that can be computed on-the-fly in a decentralized manner, we believe that rateless codes provide a solid foundation towards winning the battle on all fronts of the peer-to-peer streaming challenge: dynamics, reconciliation, and bandwidth. We believe that our positive experimental results with *rStream* implementation in the emulated realistic peer-to-peer streaming environments have revealed the effectiveness of *rStream* in real-world scenarios. In ongoing work, we are working towards a large-scale deployment and evaluation of our *rStream* implementation in the Internet.

References

[1] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed Content Delivery Across Adaptive Overlay Networks," in *Proc. of ACM SIGCOMM 2002*, August 2002.

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

[3] M. Luby, "LT Codes," in *Proc. of the 43rd Symposium on Foundataions of Computer Science*, November 2002.

[4] A. Shokrollahi, "Raptor Codes," in *Proc. of the IEEE International Symposium on Information Theory (ISIT) 2004*, June 2004.

[5] P. Maymounkov and D. Mazieres, "Rateless Codes and Big Downloads," in *Proc. of the 2nd Int. Workshop Peer-to-Peer Systems (IPTPS)*, February 2003.

[6] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A Digital Fountain Approach to Reliable Distribution of Bulk Data," in *Proc. of ACM SIGCOMM 1998*, September 1998.

[7] A. Medina, A. Lakhina, I. Matta, and J. Byers, "BRITE: Boston University Representative Internet Topology Generator," http://www.cs.bu.edu/brite, Tech. Rep., 2000.

[8] D. A. Tran, K. A. Hua, and T. T. Do, "A Peer-to-Peer architecture for Media Streaming," *IEEE Journal on Selected Areas in Communications*, vol. 22, pp. 121–133, January 2004.

[9] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable Application Layer Multicast," in *Proc. of ACM SIGCOMM 2002*, August 2002.

[10] Z. Li, B. Li, D. Jiang, and L. C. Lau, "On Achieving Optimal Throughput with Network Coding," in *Proc. of IEEE INFOCOM 2005*, March 2005.

[11] C. Wu and B. Li, "Optimal Peer Selection for Minimum-Delay Peer-to-Peer Streaming with Rateless Codes," in *Proc. of ACM Workshop on Advances in Peer-to-Peer Multimedia Streaming(P2PMMS 2005), in conjunction with ACM Multimedia 2005*, November 2005.

[12] D. Bertsekas, *Nonlinear Programming*. Athena Scientific, 1995.

[13] N. Z. Shor, *Minimization Methods for Non-Differentiable Functions*. Springer-Verlag, 1985.

[14] D. P. Bertsekas and D. A. Castanon, "The Auction Algorithm for the Transportation Problem," *Annals of Operations Research*, vol. 20, pp. 67–96, 1989.

[15] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[16] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, 1989.

[17] D. P. Bertsekas and R. Gallager, *Data Networks, 2nd Ed.* Prentice Hall, 1992.

[18] H. D. Sherali and G. Choi, "Recovery of Primal Solutions when Using Subgradient Optimization Methods to Solve Lagrangian Duals of Linear Programs," *Operations Research Letter*, vol. 19, pp. 105–113, 1996.

[19] "All-Sites-Pings for PlanetLab," http://ping.ececs.uc.edu/ping/.

[20] J. Li, "PeerStreaming: A Practical Receiver-Driven Peer-to-Peer Media Streaming System," Microsoft Research MSR-TR-2004-101, Tech. Rep., September 2004.

[21] H. Deshpande, M. Bawa, and H. Garcia-Molina, "Streaming Live Media over a Peer-to-Peer Network," Standford Database Group 2001-20, Tech. Rep., August 2001.

[22] D. A. Tran, K. A. Hua, and T. Do, "ZIGZAG: An Efficient Peer-to-Peer Scheme for Media Streaming," in *Proc. of IEEE INFOCOM 2003*, March 2003.

[23] V. N. Padmanabhan, H. J. Wang, P. A. Chow, and K. Sripanidkulchai, "Distributing Streaming Media Content Using Cooperative Networking," in *Proc. of NOSSDAV 2002*, May 2002.

[24] 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) 2003*, October 2003.

[25] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. E. Mohr, "Chainsaw: Eliminating Trees from Overlay Multicast," in *Proc. of the 4th International Workshop on Peer-to-Peer Systems*, February 2005.

[26] M. Zhang, L. Zhao, Y. Tang, J. Luo, and S. Yang, "Large-Scale Live Media Streaming over Peer-to-Peer Networks through Global Internet," in *Proc. of ACM Multimedia 2005*, November 2005.

[27] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava, "PROMISE: Peer-to-Peer Media Streaming Using CollectCast," in *Proc. of ACM Multimedia 2003*, November 2003.

[28] C. Huang, R. Janakiraman, and L. Xu, "Loss-resilient On-demand Media Streaming Using Priority Encoding," in *Proc. of ACM Multimedia 2004*, October 2004.

[29] M. Adler, R. Kumar, K. W. Ross, D. Rubenstein, T. Suel, and D. D. Yao, "Optimal Peer Selection for P2P Downloading and Streaming," in *Proc. of IEEE INFOCOM 2005*, March 2005.

Chuan Wu. Chuan Wu received her B.Engr. and M.Engr. degrees from Department of Computer Science and Technology, Tsinghua University, China, in 2000 and 2002, respectively. She is currently a Ph.D. candidate at the Department of Electrical and Computer Engineering, University of Toronto, Canada. Her research interests include distributed algorithm design to improve Quality of Service of overlay multicast applications. She is particularly interested in applying optimization and game theory to guide practical protocol design.

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.