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 both extensive simulations and 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 multimedia 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 streaming 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:

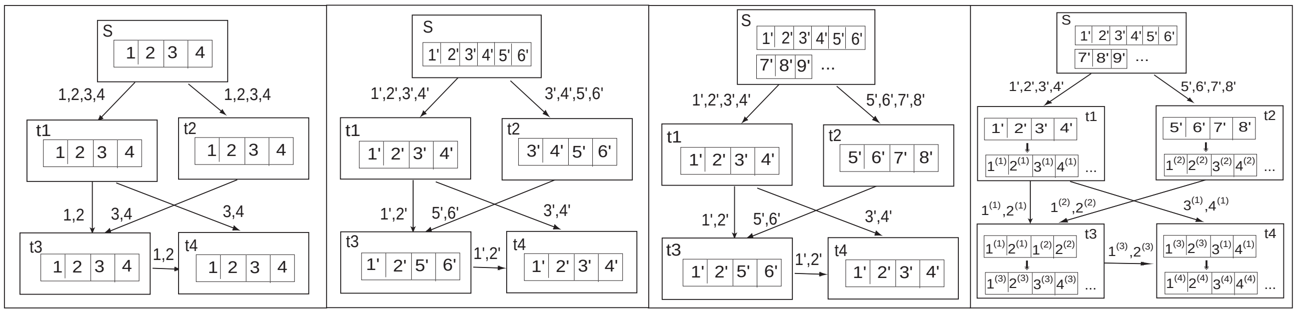
- ▷ *Network dynamics.* Peer-to-peer networks are inherently dynamic and unreliable: peer nodes may join and depart at will and without notice. The demand for stable streaming bit rates may not be satisfied.
- ▷ *Limited bandwidth availability.* Nodes in peer-to-peer networks reside at the edge of the 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.
- ▷ *Content redundancy and reconciliation.* It is typical in a peer-to-peer streaming session 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 with respect to content redundancy and reconciliation. As there are always risks that the same content may be unnecessarily supplied by multiple upstream peers, the peer nodes need to reconcile their differences to minimize such risks [1], [2].

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 streaming 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.

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. Simulation results are presented in Sec. IV. Our *rStream* implementation and its experimental results 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



(a) Peer-to-peer streaming without erasure codes
(b) Peer-to-peer streamig with fixed rate erasure codes
(c) Peer-to-peer streaming with rateless codes without recoding
(d) Peer-to-peer streaming with rateless encoding and recoding

Fig. 1. Peer-to-peer streaming with different coding schemes: a comparison.

In this paper, we consider a peer-to-peer streaming session with one streaming source and multiple participating *receivers*. A subset of the receivers retrieve the media contents directly from the source, while the others stream from one or more receivers in the *upstream*. Since the upstream peers participate in the same session, they are receivers themselves, with the exception of the streaming source. When a new peer joins the session, it is bootstrapped with a list of known peers in the session, who may serve as the initial set of upstream peers. This constructs the initial *mesh* overlay topology for the streaming session. The objective is to stream live multimedia 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 (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 multimedia 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.

A. Rateless codes

We now motivate the use of rateless codes. As a receiver retrieves media content from multiple upstream peers concurrently, it seeks to minimize the waste of bandwidth due to the delivery of duplicated content. This problem, commonly referred to as *content reconciliation problem*, is illustrated in an example in Fig. 1(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.

Erasure codes can be applied in this context to partially alleviate this problem. 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 has the favorable property that if any k (or slightly more than k) of the n transmitted symbols are received, the original k symbols can be recovered. Thus, an erasure code seamlessly tolerates packet losses and peer dynamics, making it ideal for peer-to-peer parallel transfers.

However, erasure codes do not provide a thorough solution to the reconciliation problem. To illustrate this, consider Fig. 1(b). With a $(6, 4)$ erasure code, S generates six coded blocks $1'$, $2'$, $\ldots$, $6'$ 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 most cases, but the problem may not be completely solved.

To address the challenges from content reconciliation, 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. The basic idea that underlines rateless codes is simple. 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, which is randomly chosen based on a special degree distribution. At the decoder, a decoding graph, that connects coded symbols to original symbols, is constructed based on the following encoding information: the number of original symbols each

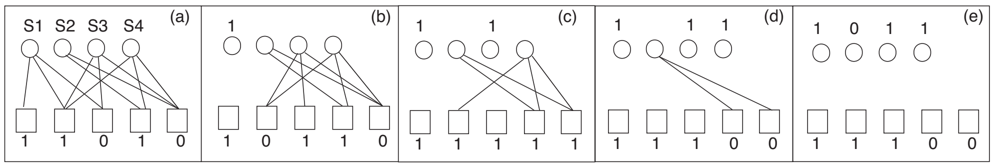


Fig. 2. Decoding with rateless codes: an example.

coded symbol is generated from (*i.e.*, the "degree") and the indices of these original symbols (neighbor indices). An example for the decoding graph is shown in Fig. 2(a), where $S1$, $S2$, $S3$, $S4$ are original symbols and 1, 1, 0, 1, 0 are coded symbols. The decoding process performed based on the decoding graph is also illustrated by the simple example in Fig. 2. With the decoding graph, the decoder finds that the first coded symbol is connected to only one original symbol S_1, so it sets $S_1 = 1$. Then the decoder performs the exclusive-or operation between S_1 and all coded symbols that are connected to S_1 in the graph, removing all the related edges. It next finds another coded symbol with degree 1, and repeats the above process until all original symbols are recovered.

With an appropriate choice of degree distribution, the coded symbols generated in this manner are potentially unique, and any $(1+\epsilon)k$ symbols ($\epsilon \ll 1$) can be used to recover the original symbols with a high probability by the decoding process. Therefore, rateless codes are useful towards finding a solution to the content reconciliation problem. In the example shown in Fig. 1(c), from the four original blocks 1, 2, 3 and 4, S generates an unlimited number of coded blocks $1'$, $2'$, $\ldots$, 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. In our example, t_1 and t_3 share $1'$ and $2'$, thus reconciliation is required at t_4.

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 [4]. Based on this favorable property, we propose a recoding scheme to be carried out by each peer to guarantee that the received blocks from any upstream peers are unique and thus useful.

In our protocol, the streaming source encodes the blocks of each media segment by an LT code based on 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 an LT encoder based on the same Robust Soliton distribution, and delivers these new coded blocks to other receivers. This *rateless recoding* protocol is summarized in Table 1. In our protocol, in order to minimize the bandwidth overhead of delivering encoding information of each coded block, we do not send the actual degrees and neighbor indices, but transmit the random seed used in the encoding process. In this way, all the encoding information can be quickly reconstructed at the receivers.

TABLE I
RECODING WITH RATELESS CODES

```
The streaming source: In the interval $[t_{i-1}, t_i]$ for streaming segment $s_i$ in the multimedia bitstream $\widetilde{M}$ of rate $r$
    For a peer $p$ streaming from the source at rate $x$:
    for $j = 1$ to $\frac{x}{r}K_i$
    1. Generate coded block $B_j^p$ from $s_i$'s original blocks $b_i^1, b_i^2, \ldots, b_i^k$ by
    (1.a) Randomly choose the degree $d_j^p$ from the Robust Soliton distribution;
    (1.b) Choose $d_j^p$ distinct original blocks uniformly at random and set $B_j^p$ to be exclusive-or of these blocks.
    2. Packetize $B_j^p$ into a packet together with the random seed used to generate it.
    3. Deliver the packet to neighbor $p$.

A receiver: After receiving $K_i$ packets for segment $s_i$
    Decode to obtain its $k$ original blocks.
    To serve another receiver $q$ at rate $y$:
    for $j = 1$ to $\frac{y}{r}K_i$
    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 random seed used to generate it.
    3. Deliver the packet to $q$.
```

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 high probability, 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 LT encoder. Thus, based on the decoding property of LT codes, after collecting $(1+\epsilon)k$ coded blocks from any upstream peers, a receiver can recover the original k blocks with high probability. $\square$

The rateless recoding protocol guarantees the uniqueness and equal usefulness of all the coded blocks in the session, thus eliminating the need for content reconciliation. In Fig. 1(d), for example, S generates a potentially unlimited number of blocks $1'$, $2'$, $\ldots$ from the original blocks 1, 2, 3 and 4. The difference between Fig. 1(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 $1^{(1)}$, $2^{(1)}$, $3^{(1)}$, $4^{(1)}$, $\ldots$, 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 $1^{(2)}$, $2^{(2)}$, $3^{(2)}$, $4^{(2)}$, $\ldots$. 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: $1^{(3)}$, $2^{(3)}$, $3^{(3)}$, $4^{(3)}$, $\ldots$. 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.

The use of rateless codes in *rStream* completely eliminates the need for content reconciliation when communicating with multiple upstream peers, while retaining all the advantages of traditional erasure codes, including their decoding efficiency and fault tolerance. We now discuss the efficiency of applying rateless recoding and leave the discussion of its superior failure resilience to Sec. III-D.

C. Efficiency of rateless recoding

Rateless codes are highly computationally efficient, as both encoding and decoding only involve exclusive-or operations. In 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 efficiency of the code itself, there is some additional overhead introduced by our recoding scheme. First, with recoding, a segment is recoded and relayed from the upstream peers only when it has been entirely received and successfully recovered at these upstream peers. Therefore, additional delays to receive an entire segment are introduced, which are determined by the length 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 choices of parameters k and l affect the efficiency of the recoding protocol. On one hand, k should be large enough to guarantee the generation of a potentially unlimited number of unique coded blocks. It is usually more efficient in practice to have larger values of l as well, so that we may have fewer blocks in each segment and thus less overhead with bookkeeping operations. On the other hand, the value of $k \cdot l$, which is the size of a media segment, may not be too large, as it plays an important role in affecting the end-to-end delay of receiving the segment at a peer. Furthermore, encoding/decoding speed of a rateless code varies with different values of k and l, and the number of extra blocks required for decoding a segment depends on k as well. We are going to further evaluate and discuss these parameters with the experiments based on our *rStream* implementation in Sec. V.

III. Optimal Peer Selection and Rate Allocation

In this section, we seek to answer the follow critical question: In mesh peer-to-peer networks, what is the best way for peers 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, give a distributed optimal rate allocation algorithm based on Lagrangian relaxation and subgradient algorithms, and then discuss its adaptation to peer dynamics, which completes the design of *rStream*.

A. Linear programming formulation

Due to the specific requirements of *live* multimedia streaming sessions, a minimal end-to-end latency at each receiver is important [7], [8]. Therefore, it is desirable to construct an optimal streaming topology, on top of which not only the streaming rate is satisfied, but also the end-to-end latencies at all receivers are minimized. In our linear programming (LP) models, we formulate the objective functions to reflect the minimization of such streaming latencies, and the constraints to reflect the streaming rate requirement and capacity limitations in the network. In what follows, we first motivate our LP formulations of multicast peer-to-peer streaming by analyzing a unicast streaming session from the streaming source to one receiver.

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 delay and f_{ij} be the transmission rate on link (i, j). Fig. 3(a) depicts an example of a unit unicast flow from S to t_4, with $r = 1$, $c_{ij} = 1, \forall (i, j) \in A$, 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. Notice that different paths may share certain 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. 3(b) illustrates the decomposition of the unit unicast flow into three fractional flows, with rates 0.2, 0.3 and 0.5, respectively.

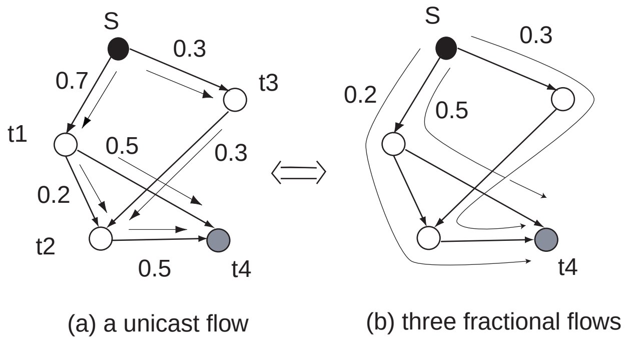


Fig. 3. An example of a unicast flow from S to t_4 and its decomposition into three fractional flows.

We calculate the average end-to-end 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. 3, the end-to-end 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

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

For the general case, we prove $\sum_{(i,j)\in A} c_{ij} f_{ij}/r$ also represents the average end-to-end 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 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 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

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

□

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 latency of the unicast flow and derive

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

subject to

$$\begin{array}{ll} \sum_{j:(i,j)\in A} f_{ij} - \sum_{j:(j,i)\in A} f_{ji} = b_i, & \forall i \in N, \\ 0 \leq f_{ij} \leq u_{ij}, & \forall (i,j) \in A, \end{array}$$

where

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

(1) is a standard minimum cost flow problem. Practically, by minimizing the average end-to-end delay, we allocate the rates in such a way that high end-to-end latency links, such as satellite and transcontinental links, are avoided as much as possible. When c_{ij}'s ($\forall (i,j) \in A$) are of similar magnitude, the average hop count from the streaming source to the receiver is minimized.

We call the optimal unicast flow decided by this linear program 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 [9]. 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. 1, 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.

LP for multicast peer-to-peer streaming

Our linear optimization model for *rStream* aims to minimize the overall end-to-end streaming delays from the source to all receivers. With respect to capacity constraints, we consider upload and download capacities at each peer, rather than link capacities. 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.

Again, let r be the required streaming rate of the session[1]. f^t denotes the conceptual flow from S to receiver t, $\forall t \in T$. f^t_{ij} denotes the rate of f^t on link (i,j). x_{ij} is the actual transmission rate and c_{ij} is the delay on link (i,j), $\forall (i,j) \in A$. Let O_i and I_i be the upload capacity and download capacity at node i, respectively. The linear program is as follows:

P:

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

subject to

$$\sum_{j:(i,j)\in A} f^t_{ij} - \sum_{j:(j,i)\in A} f^t_{ji} = b^t_i, \qquad \forall i \in N, \forall t \in T \tag{2}$$

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

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

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

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

where

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

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^t_{ij}'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 ${f^t_{ij}}^*$ 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:

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

Such an optimal rate allocation scheme $(z_{ij}, \forall (i,j) \in A)$ guarantees r at all the receivers, and achieves minimal end-to-end 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

[1]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, we omit this slight rate difference. In our implementation, we take this into consideration and use $(1+\epsilon)r$ to compute the optimal rates.

the general case, 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. As no special interest is involved, we omit detailed discussions for the multiple session case.

B. Efficient subgradient solutions

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 [10], [11], which can be efficiently implemented in a distributed manner.

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

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

and the polytope P is defined by the following constraints:

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

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

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

subject to

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

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

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

subject to

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

$$f_{ij}^t \geq 0, \qquad \forall (i,j) \in A.$$

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 [12]. 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 [13] and relaxation algorithms [14]. As the algorithms are all essentially the same as Bellman-Ford algorithm, we employ the distributed Bellman-Ford algorithm [14], [15] as our solution.

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

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

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, constraint (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 [16]. Therefore, we use the algorithm introduced by Sherali *et al.* [16] 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** [16]. We can thus calculate $\widehat{f_{ij}^t}[k]$ by

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

C. Distributed algorithm

Based on the subgradient algorithm, we now design a distributed algorithm to solve the linear program **P**, given in Table II. In practice, the algorithm to be executed on a link (i,j) is delegated by the 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 II
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 $\mu^*$ 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 $z_{ij} = \max_{t \in T} \widehat{f_{ij}^t}^*$, $\forall (i,j) \in A$.
```

Although 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 bandwidth can be indeed saturated.

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.

D. 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*, we include the handling of such dynamics as one of our main design objectives. The distributed optimal rate allocation algorithm is invoked and executed in a dynamic manner with peer dynamics.

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

2) Peer departures and failures: Peer departures and failures may lead to interrupted playback at the remaining receivers. To prevent such unexpected playback interruptions, we include a "precautionary" mechanism at each peer. During the streaming process, a peer always tries to obtain slightly more upload bandwidth from its upstream peers beyond the allocated optimal rates, making its aggregate receiving rate slightly higher at αr. α is referred to as the *tolerance factor*, with $\alpha \geq 1$.

Thus, when a peer detects the departure of its upstream peer(s), it first estimates whether its remaining streaming rate is still no less than r. If so, it is not affected; otherwise, it attempts to acquire more upload bandwidth from its remaining upstream peers. Only when the peer still fails to stream at the required rate, it sends a re-calculation request to the source for the new optimal rate allocation.

The value of α is adjusted based on an estimation of the upstream peer departure/failure probability at each receiver. When its upstream peers are departing frequently, α is increased, as long as the peer can acquire αr from its upstream peers. When the departure probability is low, α is reduced so that other peers can acquire more bandwidths from the common upstream peers. The relationship between α and its failure tolerance is to be investigated in our subsequent performance evaluation.

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 previous optimal rates, rather than from the very beginning when all the values are zero, thus expediting its convergence. If the algorithm converges, the peers adjust their rates to the new optimal values. Otherwise, it is evident that the upload capacities in the session are currently unable to support all the receivers at the required streaming rate, and the original rate allocation is kept intact.

We conclude with the note that, the efficient process of handling peer dynamics is only possible with our rateless recoding protocol. This is due to the favorable property of rateless recoding that *any* peer in the session can be used to serve as an upstream peer, as long as sufficient bandwidth exists in the system. This eliminates the complexity of content reconciliation, including the selection of a peer with the blocks in demand.

IV. Simulations

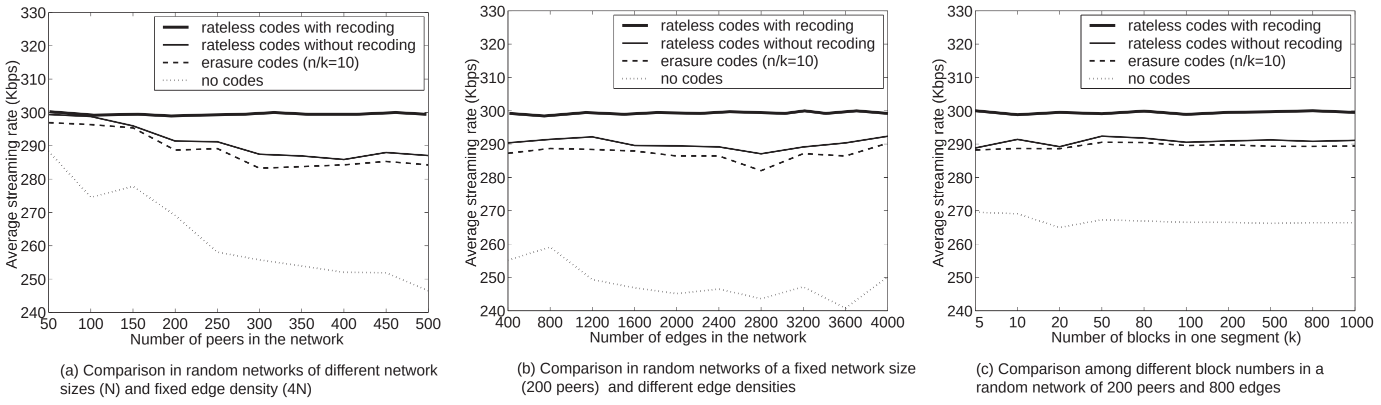


Fig. 4. Comparison of average streaming rates in the system among 4 different coding schemes: no codes, erasure codes with rate $n/k = 10$, rateless codes without recoding, and rateless codes with recoding.

In this section, we discuss results of the extensive simulations we've carried out to investigate the performance of *rStream*. We have also implemented a prototype *rStream* system, and will discuss experimental experiences with our implementation in the next section.

In order to study *rStream*'s performance over realistic network topologies, we generate random networks with power-law degree distributions with the BRITE topology generator [17]. We simulate a live streaming session of a high-quality 300 Kbps multimedia bitstream from a streaming source with 10 Mbps of upload capacity. There are two classes of receivers: ADSL/cable modem peers and Ethernet peers. In our general setting, ADSL/cable modem peers take 70% of the total population with $1.5 - 4.5$ Mbps of download capacity and $0.6 - 0.9$ Mbps of upload capacity, and Ethernet peers take the other 30% with both upload and download capacities of $8 - 12$ Mbps.

A. Benefits of the rateless recoding protocol

We first evaluate the benefits of our rateless recoding protocol in solving the reconciliation problem and maximizing bandwidth utilization. In our simulations, each media segment consists of 50 blocks, and four different coding schemes are applied: no coding, fixed-rate erasure codes, rateless codes without recoding, and rateless codes with recoding. Under each scheme, we stream the media at the optimal rates calculated by our optimization model, without content reconciliation among peers. We then eliminate the duplicated blocks obtained from different upstream peers and calculate the actual streaming rate at each receiver.

In all the comparison scenarios shown in Fig. 4, only *rStream*'s rateless recoding scheme can actually achieve the streaming rate of 300 Kbps at all receivers. In other schemes, the streaming rates are reduced at different degradation levels, caused by the duplication in the received blocks. This also shows that *rStream* performs best with respect to bandwidth utilization. In our simulation with fixed-rate erasure codes, we notice 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.

In Fig. 4(a), the comparison is made in networks of different numbers of peers (henceforth referred as *network sizes*) and a fixed *edge density*, in which the number of edges is about four times the number of peers. We find that, under the other three schemes, the average streaming rates drop with the increase of network sizes. This is caused by more severe conflicts of blocks held by different peers in larger networks, where the total number of different blocks in each media segment is limited. When the edge density changes in a network of a fixed network size, the average streaming rates do not change significantly for each coding scheme, as shown in Fig. 4(b).

We also investigate the impact of the number of blocks in each media segment on the block conflicts at the receivers (Fig. 4(c)). Varying the number of blocks from 5 to 1000, we find the average streaming rates remain approximately the same for each coding scheme. Thus, 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.

B. Resilience to peer failures

We next investigate the resilience of *rStream* in case of peer failures. Here we focus on evaluating the effects of α, the failure tolerance factor. In the implementation section, we will further examine the actual effect of failure tolerance achieved by the rateless recoding, by evaluating the throughput during a realistic streaming session with peer joins and departures.

In this simulation, we randomly choose different percentages of peers to fail, and assume all the receivers are using the same value of α. At different values of α, we calculate the remaining streaming rates at the remaining peers in the network, and seek to answer the question: at what percentage of peer failures can we still maintain the streaming rate of 300 Kbps at all receivers?

From Fig. 5, besides the fact that higher failure percentages can be tolerated by larger values of α in each network, we also observe that this failure tolerance improves quickly with the increase of edge densities in the network, under each fixed value of α. A setting of $\alpha = 1.4$ can tolerate peer failure percentage of up to 55% in a dense network. The impact of edge densities is further illustrated in Fig. 6. In this simulation, when α is set to a relatively small value of 1.2, which means we stream at the aggregate receiving rate of 360 Kbps, the 300 Kbps required rate can still be guaranteed at the receivers in a dense network in case of 30% peer failures. We believe that dense networks are more akin to realistic peer-to-peer streaming networks, in which each peer knows quite a number of upstream peers. Such satisfactory failure tolerance in dense networks achieved by small values of α suggests that the pre-fetching scheme may deliver good performance in practice, with respect to tolerating peer departures and failures.

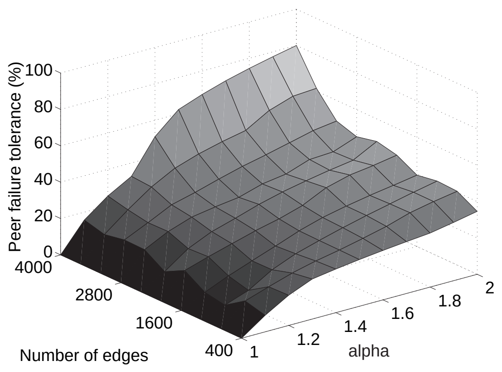


Fig. 5. Tolerance of peer failures with different α values: a 200-peer network.

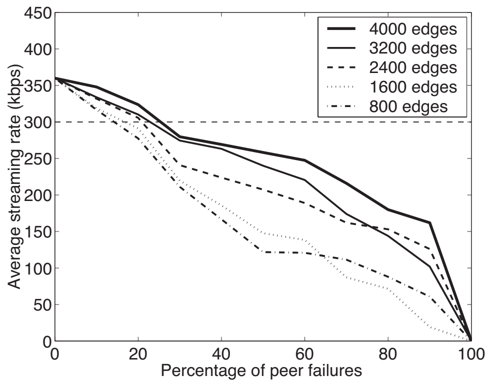


Fig. 6. Average streaming rates in case of peer failures in 200-peer networks: α = 1.2.

C. Performance of the subgradient algorithm

We first show the convergence speed of our subgradient algorithm to obtain the optimal streaming topologies in static networks in Fig. 7. In each of the static networks, the distributed algorithm runs from the beginning, where all the flow rates are initialized to be 0. We can see that it takes around 70 iterations to converge to optimality in a network of 50 peers, and this number increases slowly to about 170 for a network of 500 peers. However, the convergence speed remains approximately the same in a fixed-sized network with different edge densities.

In addition, from Fig. 8, we observe that the convergence speed to the primal feasible solution in a 300-peer static network is usually much faster than the speed to optimality. In addition, the number of iterations needed to converge to feasibility drops quickly with the increase of the percentage of Ethernet peers, which bring more abundant upload capacities into the system. In order to converge to a feasible solution which achieves 90% optimality, the algorithm takes only 75% of the number of iterations required for convergence to the optimal solution. Therefore, in such a static network, we can obtain a feasible solution to a certain degree of the optimality in a much shorter time, when it is not always necessary to achieve the optimal solution.

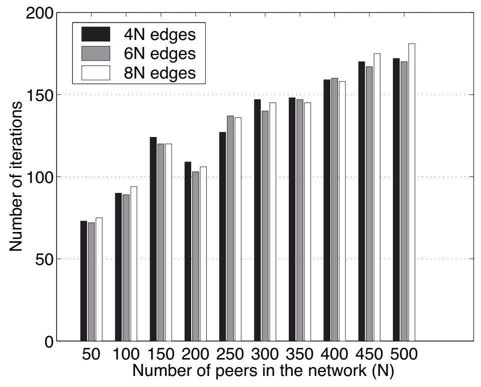


Fig. 7. Convergence speed in static networks.

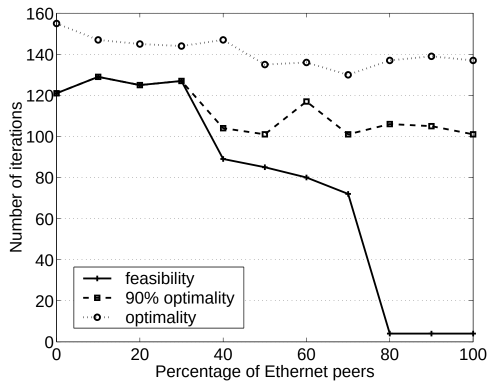


Fig. 8. Convergence speed to feasibility, 90%-optimality and optimality in static networks of 300 peers and 2400 edges.

We next investigate the convergence speed of the iterative algorithm in a dynamic peer-to-peer streaming network, which is a more practical scenario. In this simulation, 300 peers sequentially join a streaming session. The distributed optimal rate allocation algorithm is invoked about every 15 peer joins, and always runs from the previous optimal flow rates. We show the number of additional iterations needed to converge to the new optimal rates in Fig. 9.

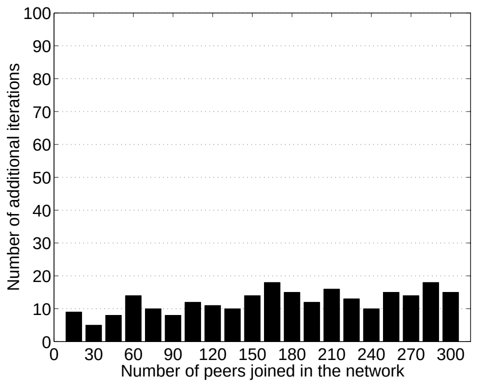


Fig. 9. Convergence speed in a dynamic network with up to 300 peers.

We observe that the convergence to new optimal rates from the previous ones in such dynamic scenario is much faster, compared to running from the very beginning in the case of static networks of the same sizes. Independent of the current network size, the algorithm always takes less than 20 iterations to converge. A similar experiment with peer departures exhibits similar results, whose figure is therefore omitted. Since a dynamic network is more akin to realistic peer-to-peer streaming scenarios, this suggests that our optimal rate allocation algorithm provides good scalability in practice.

Finally, we compare our optimal rate allocation algorithm with a commonly used peer selection heuristic [2], [18]. In this peer selection heuristic, a receiver distributes the required streaming rate among its upstream peers in proportion to their upload capacities. We compare the end-to-end latencies at receivers in the resulting streaming topologies. In this simulation, the end-to-end latency at each receiver is calculated as the weighted average of the end-to-end delays of flows from all its upstream peers, and the weight for each flow is the portion of the assigned transmission rate from the upstream peer in the required streaming rate.

The results illustrated in Fig. 10 meet our expectations. In networks of different sizes and edge densities, our end-to-end latency minimization algorithm is able to achieve much lower latencies than the heuristic, which does not take link delay into consideration. We further notice that the denser the network is, the higher the average end-to-end latency is by the heuristic. However, our optimal algorithm achieves lower latencies in denser networks. When the edge density is $4N$ in a network of N peers, the average end-to-end latency of the heuristic is about 1.5 times higher than that of our optimal algorithm, while this ratio becomes 2 in a network with $8N$ edges. For such an achievement of lower latencies in denser networks with *rStream*, we believe the reason is that there are more choices of upstream peers in a denser network and our algorithm can always find the best set of upstream peers on low delay paths. Thus, in realistic peer-to-peer streaming networks with high edge densities, the advantage of our algorithm is more evident over the commonly used heuristic.

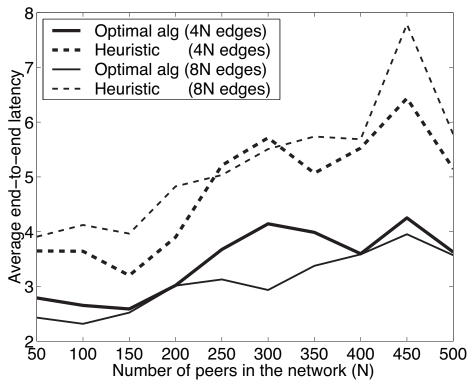


Fig. 10. Average end-to-end latency: a comparison between rStream and a peer selection heuristic.

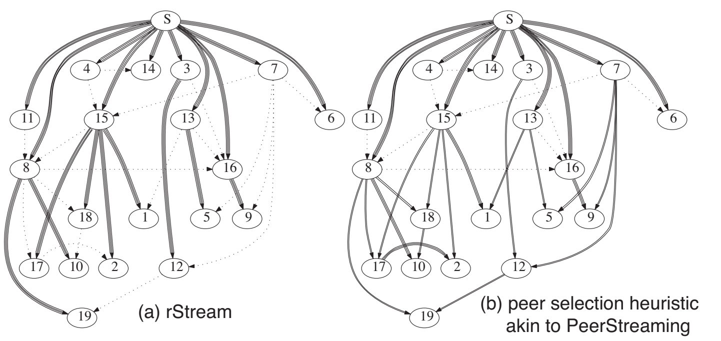


Fig. 11. Peer-to-peer streaming topologies of 20 peers: a comparison.

The streaming topologies shown in Fig. 11(a) and Fig. 11(b) further illustrate the superiority of our optimal algorithm. In these graphs, distances between pairs of peers represent 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 peer selection heuristic, peers simply distribute their streaming rate among upstream peers, which may lead to large end-to-end latencies.

V. IMPLEMENTATION

In this section, we discuss our prototype implementation of the proposed *rStream* protocol with the C++ programming language, and present experimental results from emulated peer-to-peer environments. Our implementation includes the rateless-code encoder and decoder at each peer, as well as message switching and buffer management in the application layer to implement the *rStream* protocol. 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.

A. Prototype implementation: detailed design

We first present important design decisions in our prototype implementation, which contribute to significant improvements of system performance.

Rateless-code encoder and decoder. We have implemented LT codes in our current implementation, based on the Robust Soliton distribution. In addition to the basic functions in LT codes [3], our decoder implementation introduces two new features.

First, the decoding is "streamable," in the sense that the decoding graph is constructed gradually and original blocks are recovered on the fly with the data transmission. In our implementation, whenever a new coded block arrives at the decoder, its information will be added to the decoding graph and to derive original blocks as soon as they can be decoded with the received information. Compared to the traditional implementation which decodes a segment of k original blocks after $(1+\epsilon)k$ coded blocks have arrived, our "streamable" decoder represents two advantages: (1) It maximally utilizes the time to receive coded blocks to construct the decoding graph, minimizing the 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.

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 critical to improve the performance of *rStream* protocol.

Peer selection. During streaming, when the distributed optimal rate allocation algorithm is invoked, it is executed at the background without affecting the streaming process with current optimal rates, and adjusts the rates only after their convergence.

A practical issue may occur with the streaming of rateless-code coded streams. As the number of coded blocks needed to recover a segment may vary slightly around $(1+\epsilon)k$ where ϵ is $O(\ln^2(k/\delta)/\sqrt{k})$ for LT codes, an upstream peer may not have successfully decoded a segment when a downstream peer requests it. Our implementation handles such situations by having the downstream peer temporarily increase its retrieving rates from other upstream peers that have the segment ready.

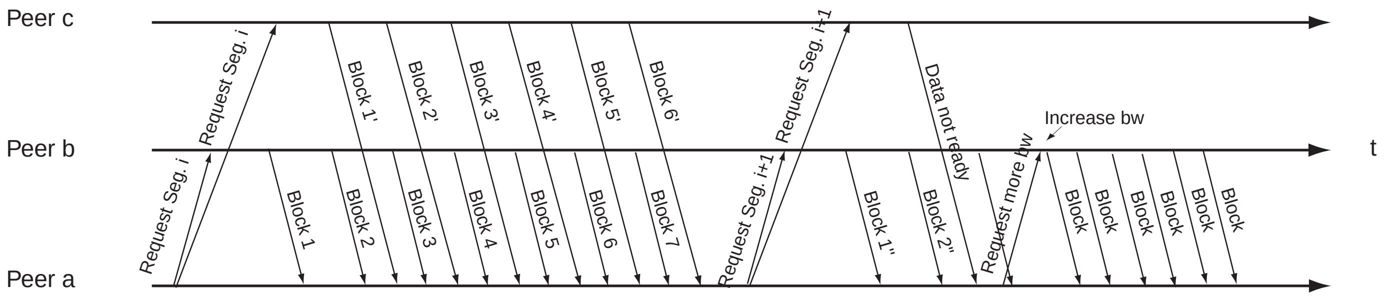


Fig. 12. Distributing methodology: the pull-push paradigm.

Distributing methodology. The *rStream* implementation disseminates the media segments with a hybrid of pull and push methods. Each peer asks its selected upstream peers to send the next media segment it is seeking at the optimal rates. Upon such a request, if the segment is available at an upstream peer, it starts generating coded blocks of the segment, and continuously pushes them to the requesting peer. 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 request for the next segment to each of its upstream peers, which serves as a "stop" signal for an upstream peer to terminate the push of the current segment. Then, if the requested new segment is ready, the upstream peer starts delivering its coded blocks; otherwise, a "data-not-ready" acknowledgment is sent instead. This pull-push paradigm is illustrated in Fig. 12, where peer a first requests segment i from upstream peers b and c, and then segment $i+1$ after i is successfully decoded. When the negative "data-not-ready" acknowledgment for segment $i+1$ is received at peer a from c, it asks for a higher transmission rate from peer b, which thereafter increases its bandwidth uploading to peer a.

By pushing coded blocks of a segment from the upstream peers instead of pulling individual blocks, we minimize the control message overhead, and more importantly, streaming delays. Meanwhile, having each peer explicitly ask selected upstream peers for segments, our implementation provides the flexibility for each peer to reconfigure its connectivity and retrieving rates, so that the protocol adapts well to network dynamics.

We note that this efficient pull-push paradigm is only achievable with rateless recoding, as every pushed block is useful for decoding. In addition, as a "streamable" decoder minimizes the delay between reception of the last coded block used for decoding and successful recovery of the segment, the number of non-used coded blocks injected into the network from the upstream peers during this time is also minimized.

B. Experimental results

We now present evaluation results based on our implementation. 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. Basic experimental settings, including network topologies and node capacity distribution, are the same as those in our simulations.

1) Performance of LT-code encoder/decoder: We first examine the net encoding and decoding speed with our LT-code implementation. In this experiment, original and coded blocks are continuously fed into the encoder or decoder, respectively. Table III-VI 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 c is a positive constant and δ is the allowed failure probability for decoding from $k + O(\sqrt{k}\ln^2(k/\delta))$ coded blocks.

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

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

Our results in Table III and V reveal an increasing trend for encoding and decoding speed as c increases and δ decreases. This is because with larger c and smaller δ, more low degrees are generated with the Robust Soliton distribution, and thus it takes fewer XOR operations to encode or decode a block. Fixing c and δ, results in Table IV and VI show that the speed decreases with the increase of k (as the block degree

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

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

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.

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

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

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

Next, we investigate the average number of coded blocks needed to successfully decode a segment, which are received from one or more upstream peers. Results are given as multiples of k, *i.e.*, $1+\epsilon$, in Table VII and VIII. 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, as coded blocks produced by different LT-code encoders are equally useful.

From these results, we can see that the parameter k affects the tradeoff between encoding/decoding speed and bandwidth consumption to deliver sufficient coded blocks for decoding. Since the encoding/decoding speed is always much higher than usual media streaming rates, larger values of k are more appropriate to reduce bandwidth consumption. Besides, larger values of l are better for higher encoding/decoding speed. Nevertheless, $k \cdot l$, the segment size, also affects the end-to-end delays at peers in the network, to be further discussed in the following section.

2) Performance of rStream *streaming:* Next, we evaluate our protocol implementation in emulated streaming applications.

Control message overhead. In this experiment, we stream a 20-minute 300 Kbps media stream in multiple networks. Control message overhead in each network is calculated as the average ratio of control traffic involved in the pull-push mechanism to data traffic at the peers. Results in Fig. 13 exhibit that our mechanism involves very low control message overhead. In further details, we observe that (1) the overhead remains at similar values 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 the number of control messages grows 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 for each media segment, and there are fewer segments if the segment size is larger. However, in all cases, the control message overhead is less than 0.3% of the total traffic, which is not significant compared to data traffic.

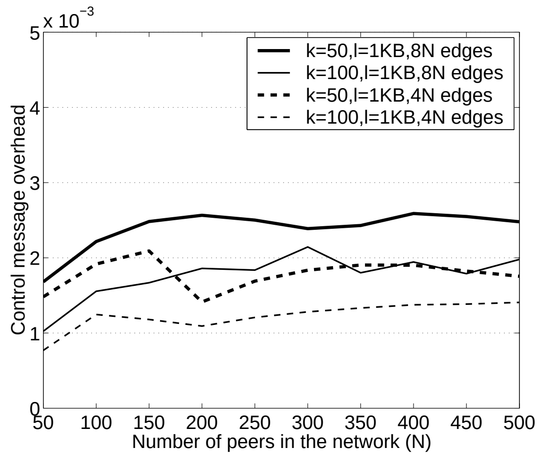


Fig. 13. Control message overhead in random networks of different sizes.

End-to-end delay. In this experiment, a $r = 300$ Kbps media stream is delivered in various networks at their computed optimally rates. End-to-end delay at a peer is measured as the difference between the time when the source starts to generate and deliver coded blocks of a segment and the time when the peer has successfully decoded the segment.

In *rStream,* peer end-to-end delay is determined by the number of overlay hops that the peer is away from the source, as well as the delay on each hop. The latter mainly consists of transmission delay of roughly kl/r. With our optimal mesh topologies, the network diameter is $O(\log(N))$, leading to log-scale increment of the delay with the network size, as shown in Fig. 14(A). We have also observed from Fig. 14(A) that delay is smaller with a higher edge density, as the network diameter becomes smaller in this case, and delay is higher when the segment size is larger.

Fig. 14(B) shows CDFs of the peer end-to-end delay distribution. In a network with up to 500 peers, end-to-end delays at most peers are moderate, and only a small portion of all the peers experiences a relatively longer delay.

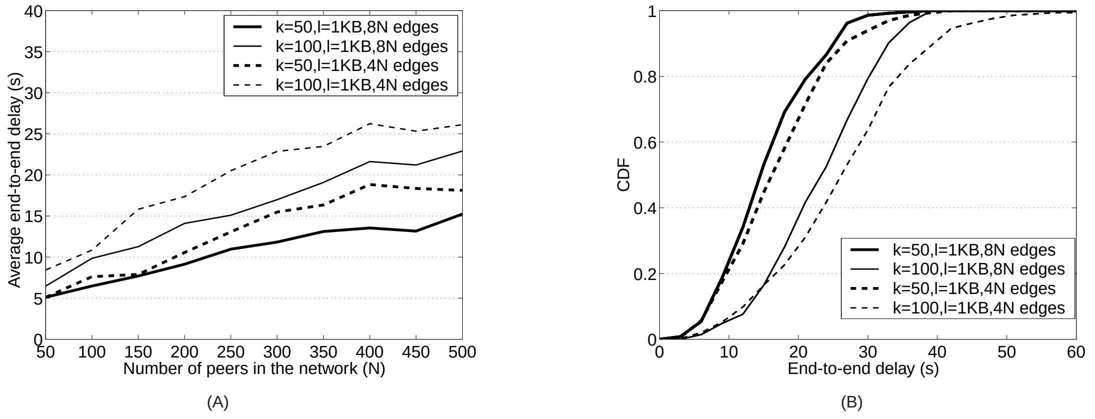


Fig. 14. End-to-end delay in random networks. (A) Average peer end-to-end delay in networks of different sizes. (B) Cumulative density function of peer end-to-end delays in a 500-peer network.

From the above results, we can see that to guarantee a small end-to-end delay, k and l should not be too large. Combining this observation with our discussion in the previous section, we see that choices of k and l affect another tradeoff between bandwidth consumption and end-to-end delay in a streaming session. Therefore, if the streaming session requires smaller delays and 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.

Throughput/goodput during peer dynamics. A consistent peer streaming rate is critical to guarantee smooth playback throughout the streaming session. To investigate the smoothness of *rStream* streaming, we use two metrics: *throughput*, which is the aggregate receiving rate of coded media bitstreams at a peer, computed by $\frac{l \times no.\ of\ coded\ blocks\ received\ in\ t}{t}$; and *goodput*, which is the actual streaming rate at a peer, *i.e.*, aggregate receiving rate of original media contents, computed by $\frac{l \times no.\ of\ original\ blocks\ recovered\ in\ t}{t}$. In the following experiments, a 300 Kbps media stream is delivered at optimally allocated rates in a 200-peer dynamic network, where the peers join and depart following an on/off model with both on/off intervals following an exponential distribution with an expected length of T.

Fig. 15(A) shows the average goodput achieved at the peers in a 45-minute period, with different peer on/off interval lengths and network edge densities. In this experiment, coding parameters are set as follows: $k = 100$, $l = 1KB$, $c = 0.05$, and $\delta = 0.1$. Only slightly larger jitters are observed in the case of $T = 30s$ than that of $T = 60s$. When a peer has more neighbors, its goodput is more stable. Nevertheless, in all cases, the average goodput is steadily maintained around 300 Kbps, which demonstrates excellent dynamic resilience of *rStream*, supported by its dynamics handling protocol discussed in Sec. III-D.

Fig. 15(B) demonstrates the average throughput and goodput achieved at the peers with various coding parameter values. Here the peer on/off interval length is set to 60 seconds, and the edge density is 4. In all scenarios, the achieved goodput is consistent at $r = 300$ Kbps, and throughput at $(1+\epsilon)r$, where ϵ is 1.23, 1.26, 1.30 and 1.34, decided by the four sets of coding parameter values respectively. From these results, we can also conclude that variation in the number of coded blocks needed to decode a segment, *i.e.*, the slightly different values of ϵ for decoding different segments, introduces little jitter in the streaming as well.

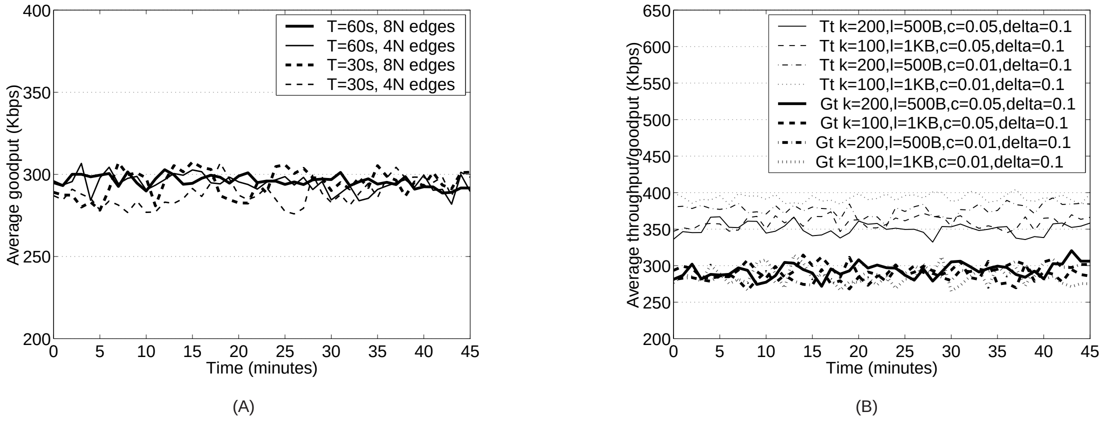


Fig. 15. Throughput/goodput in a 200-peer dynamic network. (A) Average goodput in case of different join/leave intervals and network edge densities. (B) Average throughput/goodput with different coding parameter values. (Tt: Throughput, Gt: Goodput)

VI. Related Work

Earlier work on peer-to-peer multimedia streaming has been based on a single multicast tree [19], [20], 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 [21] and SplitStream [22]. 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*, CoolStreaming [2] and PeerStreaming [18] fall into the category of streaming over mesh topologies. In CoolStreaming, 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, 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 and insufficient upload capacities. *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 [18] 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 [23] 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.* [24] 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.* [25] 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 [25], 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 by simply incorporating a tolerance factor, which can be flexibly adjusted according to the network dynamics. The problem of content reconciliation is not addressed in [25], 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, simulation and emulation 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] 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.

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

[9] 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.

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

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

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

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

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

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.