CollectCast: A Peer-to-Peer Service for Media Streaming

Mohamed Hefeeda[a] Ahsan Habib[b] Dongyan Xu[a] Bharat Bhargava[a] Boyan Botev[a]

[a]Department of Computer Sciences
Purdue University, West Lafayette
{mhefeeda,dxu,bb, botev}@cs.purdue.edu

[b]School of Information and Management Systems
University of California, Berkeley
habib@sims.berkeley.edu

Abstract

We present the design, implementation, and evaluation of a novel P2P service called *CollectCast*. CollectCast operates entirely at the application level but infers and exploits properties of the underlying network. The major properties of CollectCast include the following: (1) it infers and leverages the underlying network topology and performance information for the selection of senders; (2) it monitors the status of peers and connections and reacts to peer/connection failure or degradation with low overhead; (3) it dynamically switches active senders and standby senders, so that the collective network performance out of the active senders remains satisfactory. We perform both real-world measurements and simulations of CollectCast. Our simulation results show that CollectCast-based P2P streaming achieves better performance than P2P streaming based only on end-to-end network performance information. The real-world measurements are obtained by implementing a P2P media streaming system (called PROMISE) on top of CollectCast. We have installed and tested PROMISE on the PlanetLab test bed. The results of the packet-level and frame-level performance obtained from streaming several MPEG-4 movies demonstrate the potential benefits for the applications built on top of CollectCast.

1 Introduction

Peer-to-peer (or P2P) systems have gained tremendous momentum in recent years. In a P2P system, peers communicate directly with each other for the sharing and exchange of data as well as other resources such as storage and CPU capacity. Paralleling research in other aspects of P2P, such as lookup [27, 34, 30], storage [12, 31], and multicast [9, 1, 36], we in this paper focus on P2P real-time media streaming. Different from general P2P file sharing, P2P media streaming poses more stringent resource requirements for real-time media data transmission. However, as first addressed in our earlier work [37], for a media file of playback rate R_0, a single sending peer may not be able or willing to contribute an outbound bandwidth of R_0. Moreover, downloading the entire media file before playback is not the best solution, due to the potentially large media file size and thus long

download time. As our solution, we propose a P2P media streaming model that involves multiple sending peers in one streaming session.

Despite recent research results of ours and others, a number of challenges intrinsic in P2P media streaming have not been addressed. In this paper, we present our solution to the following challenge: in a highly diverse and dynamic P2P network, how to select, monitor and possibly switch sending peers for each P2P streaming session, so that the best possible streaming quality can be maintained? The dynamics and diversity are reflected in both peers and network connections between peers: (1) a sender may stop contributing to a P2P streaming session at any time, (2) the outbound bandwidth contributed by a sender may change, (3) the connection between a sender and the receiver may exhibit different end-to-end bandwidth, loss, and failure rate, and more importantly (4) the underlying network topology determines that the connections between the senders and the receiver are *not* independent of each other, with respect to their loss and failure rate. As a result, the quality of a P2P streaming session depends on judicious selection of senders, constant monitoring of sender/network status, and timely switching of senders when the sender or network fails or seriously degrades. Unfortunately, previous works in P2P media streaming do not provide a systematic solution to the above challenge. For example, some previous works simply assume that a receiver receives media data from only *one* sender [2, 36, 9]. For the works that do assume multiple senders for one receiver [20, 24], there is no study on the *selection* of the best senders.

In this paper, we present the design, implementation, and evaluation of a novel P2P service called *CollectCast*. CollectCast operates entirely at the application level but infers and exploits properties (topology and performance) of the underlying network. CollectCast has a pattern of "one receiver collecting data from multiple senders". Unlike other multiple-to-one network services such as concast [6], each CollectCast session involves two sets of senders: the *standby senders* and the *active senders*. Members of the two sets may change dynamically during the session. CollectCast reflects the P2P philosophy of dynamically and opportunistically aggregating the limited capacity of peers to perform a task (streaming) traditionally performed by a dedicated entity (a media server). The major properties of CollectCast include the following: (1) it infers and leverages the underlying network topology and performance information for the selection of senders. This is based on a novel application of several network performance inference techniques; (2) it monitors the status of peers and connections and reacts to peer/connection failure or degradation with low overhead; (3) it dynamically switches active senders and standby senders, so that the collective network performance out of the active senders remains satisfactory. We perform both real-world measurements and simulations of CollectCast. Our simulation results show that CollectCast-based P2P streaming achieves better performance than P2P streaming based only on end-to-end network performance information. To perform the real-world measurements, we implement a P2P media streaming system on top of CollectCast. We call this system PROMISE. PROMISE has been installed and tested on the PlanetLab test bed [26]. The results of the packet-level and frame-level performance obtained from streaming several MPEG-4 movies demonstrate the potential benefits for the applications built on top of

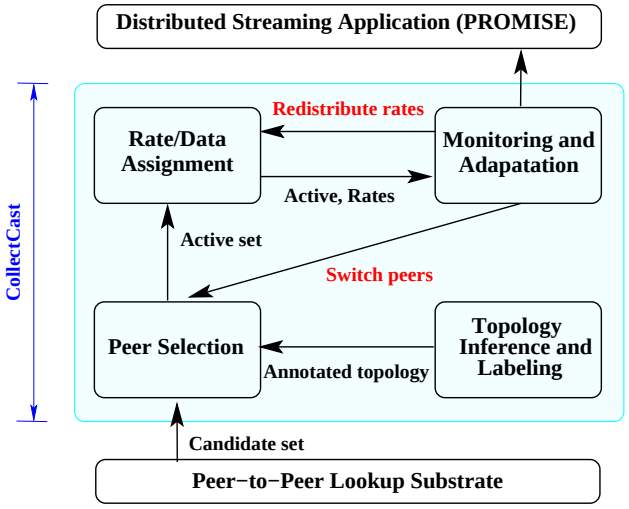


Figure 1: Different components of CollectCast and the interaction among them.

CollectCast.

The rest of the paper is organized as follows. An overview of CollectCast is given in Section 2. The following four sections provide the details of CollectCast: peer selection in Section 3, rate and data assignment in Section 4, monitoring and adaptation in Section 5, and topology inference and labeling in Section 6. An extension of CollectCast that makes it TCP-friendly is presented in Section 7. We evaluate CollectCast through simulation in Section 8. Section 9 describes a prototype system (PROMISE) built on top of CollectCast and presents the measurement results obtained from running PROMISE on PlanetLab nodes. Section 10 discusses the related work. Finally, Section 11 concludes the paper.

2 Overview of CollectCast

This section first provides an overview of the different components of CollectCast and how they interact. Details of CollectCast components will be presented in Sections 3–7.

2.1 Components of CollectCast

CollectCast is a new network service targeted towards P2P media streaming applications. Its objective is to judiciously choose the sending peers and orchestrates them to achieve the best quality streaming for the receiver in a highly diverse and dynamic P2P network. As shown in Figure 1, CollectCast is to be layered on top of a P2P lookup substrate and is comprised of four components: (1) topology inference and labeling, (2) peer selection, (3) rate and data assignment, and (4) monitoring and adaptation. The components of CollectCast are divided into receiver-side (Figure 2) and sender-side (Figure 3) functions. The receiver plays the leading role in CollectCast.

CollectCast leverages one of the P2P lookup substrates proposed in the literature to manage peer membership

and perform object look up. Other components of CollectCast are independent of the underlying P2P lookup substrate. Therefore, CollectCast can use substrates such as Pastry [30], Chord [34], or CAN [27]. We note that each of these P2P lookup substrates returns only one peer for an object lookup request, if the object exists in the system. In our prototype, we have modified Pastry to return multiple peers for each lookup request. We used Pastry because it has been implemented [14] and the code is written in Java with good portability. We do not discuss the details of the P2P lookup substrate in this paper; interested reader is referred to [27, 34, 30]. In the following paragraph, we describe the interaction among the different components to establish and manage a streaming session.

A streaming session in CollectCast is established as follows. A peer requesting a movie runs the receiver procedure shown in Figure 2. The procedure first issues a lookup request to the underlying P2P lookup substrate, which returns a set of *candidate* peers who have the movie. The candidate set typically contains 10 to 20 peers. The protocol then invokes the topology inference and labeling components to construct and annotate the topology connecting the candidate peers with the receiver. The topology is annotated by the available bandwidth and loss rate. Using the annotated topology, the selection algorithm determines the *active* sender set. The active set is the best subset of peers that is likely to yield the best quality for this streaming session. The rest of the candidate peers are kept in a *standby* sender set, from which replacement peers will substitute failed or degraded peers from the active set. Then, the rate and data assignment component is called to determine the appropriate rate and data portions for each active peer. The rate of each active sender is based on the sender's offered rate and the goodness of the path from that sender to the receiver. Once the rates and data are assigned, the receiver establishes parallel connections with all peers in the active set. Two connections are established with each peer. A UDP connection for sending the stream packets,[1] and a TCP connection for sending control packets. The monitoring and adaptation component oversees the streaming session to maintain the quality. It measures the streaming rate and packet loss rate for each active sender. If the rate coming from a peer drops due to a peer failure or network congestion, the monitoring and adaptation component will try to redistribute the rate among the alive peers. If redistribution will not yield the full quality, a peer switching is performed to replace the failed peer with another peer(s) from the standby set. The topology is updated with new values measured *passively* during streaming and the peer selection component is invoked to update the active set.

3 Peer Selection in CollectCast

The key component of CollectCast is peer selection. Since the P2P environment is highly diverse and dynamic, selecting the best peers to serve a streaming session is critical to providing the desired high quality streaming. The selection technique should avoid peers that fail often and share congested network paths. This section presents

[1] Adjusting the rate of the UDP connection to compete fairly with TCP traffic of other applications is discussed in Section 7.

Receiver Side

```
1.  CAND ← P2PSubstrateLookup(fileId);
2.  T ← BuildTopology(CAND, receiverId);
3.  ACTV ← SelectPeers(T);
4.  while the session is not over do
5.     Connect(ACTV); /* Establish the streaming session */
6.     SendControlPackets(ACTV);
7.     needToSwitch ← false;
8.     while needToSwitch == false do
9.        needToSwitch ← ReceiveSegment();
10.    end while
11.    T ← UpdateTopology(T, newMeasuredValues);
12.    ACTV ← SelectPeers(T);
13. end while
```

Figure 2: CollectCast: Receiver side

Sender Side

```
1.  /* Wait for a control packet */
2.  while this peer is an active supplier do
3.     ctrPkt ← ReceiveControlPacket();
4.     rate ← GetAssignedRate(ctrPkt);
5.     dataToSend ← GetAssignedData(ctrPkt);
6.     do
7.        SendData(dataToSend, rate);
8.        UpdateStatistics();
9.     while no control packet received;
10. end while
```

Figure 3: CollectCast: Sender side

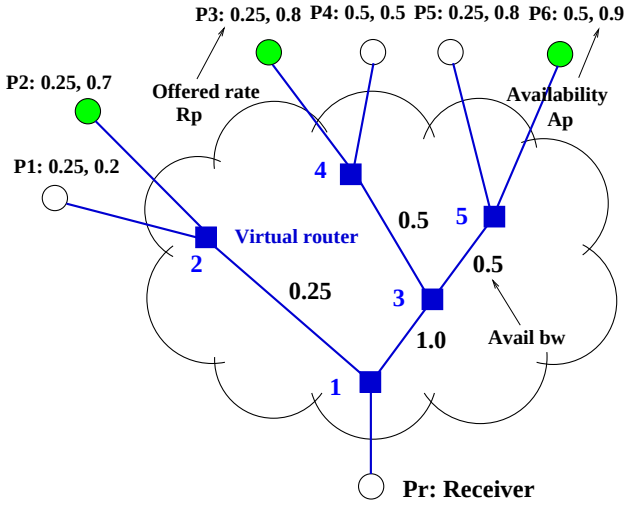


Figure 4: Topology-aware selection. It constructs an approximate topology and considers shared segments.

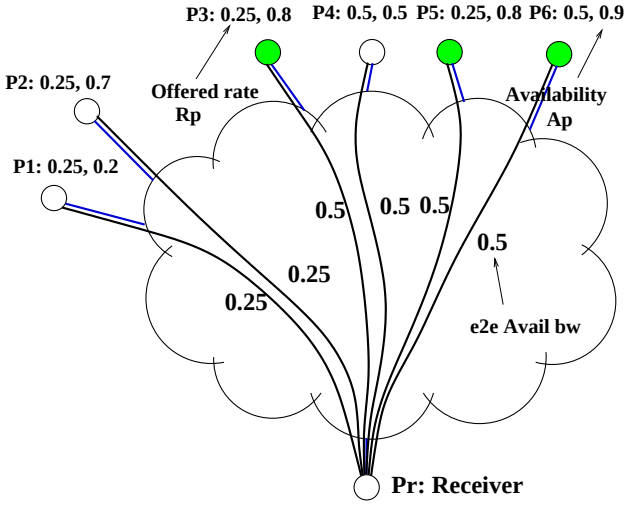


Figure 5: End-to-end selection. It does not not consider shared segments.

three selection techniques: random, end-to-end, and topology-aware. The input to the selection technique is a set of candidate peers returned from the P2P lookup substrate. The output is a subset of the candidate peer set (called the active peer set) to start streaming the movie.

The random technique randomly chooses a number of peers that can fulfill the aggregate rate requirement, even though these peers may have low availability and share a congested path. The end-to-end technique estimates the "goodness" of the path from each candidate peer to the receiver. Based on the quality of the *individual* paths and on the availability of each peer, the technique chooses the active set. The end-to-end technique does *not* consider shared segments among paths, which may become bottlenecks if peers sharing a tight segment are chosen in the active set. In contrast to the end-to-end technique, the topology-aware technique *infers* the underlying topology and its characteristics and . considers the goodness of each *segment* of the path. Thus, it can make

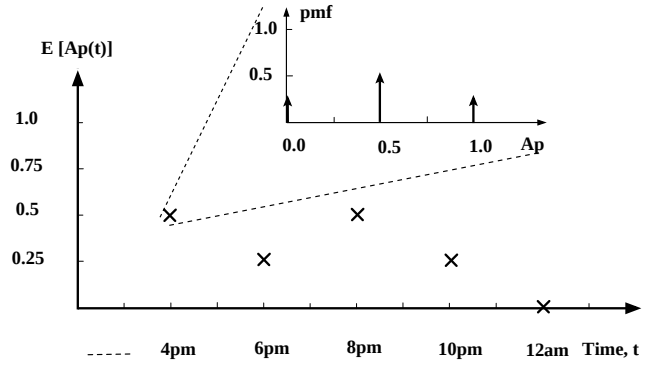


Figure 6: Peer availability: Modeled as a discrete-time stochastic process $\mathcal{A}_p(t)$. At each time instant t_i, the availability is the random variable $\boldsymbol{A}_p(t_i)$. The y-axis shows the expected value of $\boldsymbol{A}_p(t_i)$. The subfigure at the upper-right corner shows a possible distribution for $\boldsymbol{A}_p(t_i)$.

a judicious selection by avoiding peers whose paths are sharing a tight segment.

Illustrative example. Consider the example shown in Figures 4 and 5. The lookup step returns peers $P_1, P_2, \ldots, P_6$ as a candidate set to the receiving peer P_r. The random technique may choose P_1, P_3, P_4 as the active set, even though some of these peers have low availability (P_1), and others share a congested path (P_3, P_4). The end-to-end technique considers the goodness of *individual* paths and the availability of peers. Therefore, it selects peers P_3, P_5, P_6. It is not, however, aware of the shared segment between the two paths $P_5 \rightsquigarrow P_r$ and $P_6 \rightsquigarrow P_r$, which can not afford the aggregate rate from P_5 and P_6. Finally, the topology-aware technique makes an informed decision, using the annotated topology, and selects the best set: P_2, P_3, P_6.

3.1 Notations and Peer Model

Before presenting the details of the selection technique used in CollectCast, let us define the notations and the peer model used in the paper.

Notations. We use the following notations throughput the paper. We use bold symbols (e.g., $\boldsymbol{G}_p$) to represent random variables and regular symbols (e.g., R_p) to represent constant values. The symbol $\mathcal{A}_p(t)$ is used to represent an ensemble of random variables (i.e., a stochastic process) indexed by the time t. An edge from node i to node j is denoted by $i \rightarrow j$. A path with one or more edges from node x to node y is denoted by $x \rightsquigarrow y$. The expectation of a random variable $\boldsymbol{x}$ is denoted by $\overline{\boldsymbol{x}}$. The playback rate of the media file is referred to as R_0.

Peer model. We assume that peers exhibit heterogeneous characteristics and they do not have server-like capability: they contribute limited capacity, and may fail or reduce their sending rates unexpectedly. Therefore, multiple sending peers may be needed to serve a requesting peer at any time. In order to capture the heterogeneous characteristics of peers, we associate each peer p with two parameters: offered rate R_p and availability $\mathcal{A}_p(t)$. The offered rate is the maximum sending rate that a peer can (or is willing to) contribute to the system. A lower bound on the offered rate (R_p^{min}) is imposed by the system to limit the maximum number of peers required

to serve a request. This limits the number of concurrent connections (and hence the control overhead) that the requesting peer needs to maintain.

The availability is the fraction of time a peer is available for serving. We model the availability of a peer p as a family of random variables, collectively referred to as the discrete-time stochastic process $\mathcal{A}_p(t)$ (Figure 6). At each time instant t_i, the distribution of the random variable $\boldsymbol{A}_p(t_i)$ describes the behavior of p at this instant. For example, the subfigure at the upper-right corner of Figure 6 indicates that at time $t = 4$pm, the peer is using: (i) all of its bandwidth (i.e., not available for serving) 25% of the time, (ii) half of its bandwidth 50% of the time, and (iii) nothing of its bandwidth in the remaining 25% of the time. Modeling the availability in this way captures the relation between the time of the day and the varying usage of the bandwidth by the peer. For instance, in the morning, a peer may use a small portion for its bandwidth for e-mail checking, while in the evening it may use a larger portion for music download.

The offered rate and availability information is either entered by the user during the initial set up, or collected by a daemon running on each participating peer. If the user enters this information, this would indicate that the maximum contribution he is willing to offer, despite the amount of resources available. The daemon can easily find out the speed of the Internet connection the peer is using, and hence the maximum offered rate. For the availability, the daemon collects statistics during the regular operation of the peer and uses them to estimate and periodically refine the availability. One way of doing that is to divide the time into equal length intervals of size one hour each. For each interval, the daemon measures the amount of traffic sent during this interval. By knowing the connection speed, the remaining (available) bandwidth can be calculated. This process is repeated over several days to obtain enough samples in order to estimate a relatively accurate distribution of the availability. Note that, clock differences among peers do not any cause problem for this approach, since peers collect this information independently. Moreover, when a peer is requested to report this information (which will be used in the selection algorithm described in Section 3), it sends the information for the *current* interval, regardless of the time of the requesting peer.

3.2 Topology-Aware Selection

This section presents the details of the topology-aware selection technique. We first define the *goodness topology* and how it is annotated by network performance metrics (e.g., available bandwidth and loss rate) and peers characteristics (e.g., offered rate and availability). Then, we use the goodness topology to estimate the peer goodness for the session being established. Finally, we state the peer selection problem, formulate it as an optimization problem, and present an algorithm to solve it.

Goodness topology $\mathcal{T}$. It is a directed graph that interconnects the candidate peers and the receiving peer (Figure 4). Each edge (hereafter called a path segment, or simply a segment) $i \rightarrow j \in \mathcal{T}$ is annotated with a goodness random variable $\boldsymbol{g}_{i \rightarrow j}$. Each leaf node represents a peer p from the set of candidate peers $\mathbb{P}$ and has

two attributes: a fixed offered rate R_p and a random variable $\boldsymbol{A}_p$ that describes the availability of p for streaming at the current time[2].

The goodness topology is built in two steps. In the first step, *network tomography* techniques are used to infer the approximate topology and annotate its edges with the metrics of interest, e.g., loss rate, delay, and available bandwidth. This is called the *inferred topology*. A segment in the inferred topology may represent a sequence of links with no branching points in the physical topology. This hides unnecessary details and yields a compact representation of the physical topology. We assume that routes from candidate peers to the receiver do not change during the course of the streaming session. This indicates that the inferred topology is a tree-structured graph rooted at the receiver. Previous studies [3, 10] adopted the same assumption, which is backed by Internet measurement studies. For example, [39] indicates that the end-to-end Internet paths often remain stable for a significant period of time. More details on building the inferred topology are given in Section 6. The second step transforms the inferred topology to the goodness topology. The transformation process is basically computing a "logical" goodness metric for each segment from its properties.

Segment goodness. The segment goodness $\boldsymbol{g}_{i\to j}$ is, in general, a function of one or more properties of the segment $i \to j$, depending on the feasibility and ease of measuring these properties segment-wise. Segment properties may include loss rate, delay, jitter, and available bandwidth. We represent the segment goodness as a function of the loss rate and available bandwidth because these two metrics: (1) can be measured segment-wise [3], and (2) are the most influential on the receiving rate, and hence on the quality. A segment with high available bandwidth and low loss is unlikely to introduce high jitter or long queuing delay. The goodness of segment $i \to j$ is defined as: $\boldsymbol{g}_{i\to j} = w_{i\to j}\boldsymbol{x}_{i\to j}$, where $w_{i\to j}$ is a *weight* that depends on the available bandwidth and level of sharing on segment $i \to j$, and $\boldsymbol{x}_{i\to j}$ is a binary random variable that depends on the loss rate. $\boldsymbol{x}_{i\to j}$ is defined in terms of the packet loss rate as follows:

$$\boldsymbol{x}_{i\to j} = \begin{cases} 1, & \text{if a packet is not lost on } i \to j \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

If the average loss rate on segment $i \to j$ is $\overline{\boldsymbol{l}}_{i\to j}$, then the mean of $\boldsymbol{x}_{i\to j}$ is: $E[\boldsymbol{x}_{i\to j}] = \overline{\boldsymbol{x}}_{i\to j} = 1 \times (1 - \overline{\boldsymbol{l}}_{i\to j}) + 0 = 1 - \overline{\boldsymbol{l}}_{i\to j}$.

The weight $w_{i\to j}$ is determined by the available bandwidth on segment $i \to j$ (denoted by $b_{i\to j}$) and the aggregate rate from peers sharing this segment if they are selected in the active set. The segment weight is a *per-peer* metric, that is, the weight of segment $i \to j$ (and hence, its goodness) could differ for two peers sharing

[2]Note that, $\boldsymbol{A}_p$ should be $\boldsymbol{A}_p(t_{cur})$, where t_{cur} is time interval corresponding to the current time. Since all calculations will be made with regard to the same time interval, we will drop the time index for the sake of clarity of presentation.

segment $i \rightarrow j$. The weight of segment $i \rightarrow j$ for a peer p is denoted by $w_{i \rightarrow j}^{(p)}$ and is given by:

$$w_{i \rightarrow j}^{(p)} = \min\Big(1, \max(0, (b_{i \rightarrow j} - \sum_{s \in S, i \rightarrow j \in s \rightsquigarrow r} R_s)/R_p)\Big), \tag{2}$$

where S is the set of peers selected to be in the active set thus far, and $s \rightsquigarrow r$ is the path from the sending peer s to the receiving peer r. The intuition behind this formulation is that, if a segment has a bandwidth equal to or higher than the aggregate rate contributed from peers sharing this segment, then this segment will not throttle this aggregate rate, and hence its weight is set to 1. Otherwise, the weight is a fraction proportional to the shortage in the bandwidth if peer p along with peers in S are chosen to serve. The example given later in this section explains numerically how to compute these weights.

Peer goodness. We define the goodness of a peer p, $\boldsymbol{G}_p$, as a function of its availability and the goodness of all segments comprising the path $p \rightsquigarrow r$. $\boldsymbol{G}_p$ has the following form:[3]

$$\boldsymbol{G}_p = \boldsymbol{A}_p \prod_{i \rightarrow j \in p \rightsquigarrow r} \boldsymbol{g}_{i \rightarrow j} = \boldsymbol{A}_p \prod_{i \rightarrow j \in p \rightsquigarrow r} w_{i \rightarrow j}^{(p)} \boldsymbol{x}_{i \rightarrow j}. \tag{3}$$

Peers with high expected goodness values (close to 1) indicate that these peers are likely to provide good and sustained sending rate. This is because they are unlikely to stop sending packets *and* these packets will be transmitted through network paths of low dropping probability.

Best active peers set. This is the subset of peers that are likely to provide the "best" quality to the receiver. The perceived quality is quantified by the aggregated receiving rate. We are now ready to state the selection problem:

> ***Active Peers Selection Problem***. *Given the annotated goodness topology $\mathcal{T}$, find the set of active peers $\mathbb{P}^{actv} \subseteq \mathbb{P}$ that maximizes the expected aggregated rate at the receiver, provided that the receiver inbound bandwidth is not exceeded.*

Mathematically, this can be phrased as: find $\mathbb{P}^{actv}$ that

$$\text{Maximizes} \quad E\Big[\sum_{p \in \mathbb{P}^{actv}} \boldsymbol{G}_p R_p\Big] \tag{4}$$

$$\text{Subject to} \quad R_l \leq \sum_{p \in \mathbb{P}^{actv}} R_p \leq R_u, \tag{5}$$

where $\boldsymbol{G}_p$ and R_p are the goodness and offered rate of peer p, respectively, and R_l, R_u are the lower and upper rate targets. Section 4 shows how R_l, R_u are determined.

Selection algorithm. Given the problem formulation above, finding the best active set $\widehat{\mathbb{P}}^{actv}$ is straightforward. Figure 7 describes an algorithm to determine $\widehat{\mathbb{P}}^{actv}$ given the goodness topology $\mathcal{T}$. The algorithm

[3]For the feasibility of the analysis, we are making a reasonable assumption: the quality of individual segments of the path is independent from each other and from the availability of the peer.

Selection Algorithm

1. Enumerate all possible sets that satisfy constraints in (5): $\mathbb{P}^1, \mathbb{P}^2, \ldots, \mathbb{P}^M$.
2. $\widehat{\mathbb{P}}^{actv} = null;\ \ maxE = 0$
3. **for** each $\mathbb{P}^m, 1 \leq m \leq M$ **do**
4. Set $dif_{i\to j} = b_{i\to j}, \forall i \to j \in \mathcal{T}$
5. $E = 0$
6. **for** each $p \in \mathbb{P}^m$ **do**
7. $G_p = \overline{\boldsymbol{A}}_p$
8. **for** each segment $i \to j \in p \rightsquigarrow r$ **do**
9. $G_p = G_p \times \min(1, \overline{\boldsymbol{x}}_{i\to j} \times dif_{i\to j}/R_p)$
10. $dif_{i\to j} = \max(0, dif_{i\to j} - R_p)$
11. **endfor**
12. $E \ = E + G_p$
13. **endfor**
14. **if** $E < maxE$ **then**
15. $maxE = E$
16. $\widehat{\mathbb{P}}^{actv} = \mathbb{P}^m$
17. **endfor**
18. **return** $\widehat{\mathbb{P}}^{actv}$

Figure 7: Pseudo code for selecting the best active peers set.

determines the expected aggregated rate for all possible active sets and selects the one with the highest rate. There are several code optimization possibilities which are not discussed for the sake of clarity.

Complexity. The selection algorithm enumerates all possible sets that satisfy the constraints in (5). However, the input (the candidate set) to the algorithm is fairly small (10 to 25 peers from which we choose 3 to 5 active peers). Checking the constraints in (5) is a matter of adding a few numbers and comparing with the bounds. Many sets will be disqualified by the constraint. For the remaining qualified sets, selecting the best among them is also a simple computation. In addition, the selection algorithm is invoked only a few times: at the beginning of the session and when a peer switching is needed. In our implementation, the selection algorithm is called no more than five times during a 60-minute streaming session, and each call takes a few tens of milliseconds on a reasonable PC. Therefore, although designing more efficient selection algorithms is possible, we believe that the payoff will not be significant.

Complete example. This examples shows the details of selecting the best peers in the topology shown in Figure 4. To simplify the discussion, we set $R_l = R_u = R_0$ and the loss rate in all path segments to 0, that is, $\overline{\boldsymbol{x}}_{i\to j} = 1, \forall i, j$. The playback rate R_0 is 1 Mb/s. The possible active sets that satisfy the constraints in 5 are: $\{P_4, P_6\}$, $\{P_3, P_5, P_6\}$, $\{P_2, P_5, P_6\}$, $\{P_1, P_5, P_6\}$, $\{P_3, P_4, P_5\}$, $\{P_2, P_4, P_5\}$, $\{P_1, P_4, P_5\}$, $\{P_1, P_3, P_4\}$, $\{P_2, P_3, P_4\}$, $\{P_2, P_3, P_6\}$, $\{P_1, P_3, P_6\}$, $\{P_1, P_2, P_4\}$, $\{P_1, P_2, P_6\}$, and $\{P_1, P_2, P_3, P_5\}$. The expected aggregated rate is then computed for every set. For instance, the expected aggregated rate for $\{P_3, P_5, P_6\}$ is $1 \times .8 + 1 \times .8 + .25/.50 \times .9 = 2.05$. P_5 and P_6 have a shared segment ($5 \to 3$) of bandwidth .5. If we assign $w_{5\to 3}^{(P_5)} = 1$ (because the available bandwidth on the path is greater than P_5's offered rate), P_6 will get a left-over bandwidth of 0.25, which makes the weight $w_{5\to 3}^{(P_6)} = 0.25/0.50$. If we assign the $w_{5\to 3}^{(P_6)} = 1$, P_5 gets a weight of 0 because no bandwidth is left for this peer on the shared segment. We consider all combinations of ordered peers in a particular peer set to maximize the expected rate. The expected rate of all possible sets are 1.4, 2.05,

1.95, 1.45, 1.85, 2.0, 1.5, 1.75, 1.25, 2.4, 1.9, 1.2, 1.6, and 2.3, respectively. The highest aggregate rate comes from the set $\{P_2, P_3, P_6\}$.

3.3 End-to-End Selection

As a comparison to the topology-aware selection, we consider selecting the active peers based only on end-to-end information. Instead of building the underlying topology, the end-to-end technique uses the end-to-end path bandwidth and loss rate in addition to peer availability. It exploits no information about the path segments shared among peers and therefore imposes less overhead than the topology-aware selection. However, as our evaluation shows (Section 8), while better than random selection, it does not perform as well as the topology-aware selection. We can formulate the end-to-end selection as a special case of the topology-aware selection as follows. Instead of writing the peer goodness as in Equation (3), we write it as: $\boldsymbol{G}_p = \boldsymbol{A}_p w_{p \rightsquigarrow r} \boldsymbol{x}_{p \rightsquigarrow r}$, where $w_{p \rightsquigarrow r}$ is the *path* weight and $\boldsymbol{x}_{p \rightsquigarrow r}$ is the binary random variable that depends on the end-to-end path loss rate. The mean of $\boldsymbol{x}$ is: $\overline{\boldsymbol{x}}_p = 1 - \overline{\boldsymbol{l}}_{p \rightsquigarrow r}$, where $\overline{\boldsymbol{l}}_{p \rightsquigarrow r}$ is the average end-to-end path loss rate. Computing the path weight is much easier in this case and is given by:

$$w_{p \rightsquigarrow r} = \begin{cases} 1, & R_p \leq b_{p \rightsquigarrow r} \\ \frac{R_p - b_{p \rightsquigarrow r}}{R_p}, & \text{otherwise} \end{cases} \tag{6}$$

Using this formulation, the expected rate maximization problem can be solved in a way similar to the one in Section 3.2.

Example. The parameters in this example are the same as in the example in Section 3.2. Thus, the possible active sets are also the same. The end-to-end selection utilizes the availability of peers and the path available bandwidth to calculate the expected rate. For example, the expected rate of the set $\{P_3, P_5, P_6\}$ is $1 \times .8 + 1 \times .8 + 1 \times .9 = 2.5$. The corresponding expected rate of all possible sets are 1.4, 2.5, 2.4, 1.9, 2.1, 2.0, 1.5, 2.0, 1.5, 2.4, 1.9, 1.4, 1.8, and 2.5, respectively. The maximum expected rate is 2.5, which is supplied by peer sets $\{P_3, P_5, P_6\}$ and $\{P_1, P_2, P_3, P_5\}$. Either of them can be taken, but we prefer the set with fewer peers to reduce the overhead of maintaining multiple concurrent connections.

4 Rate and Data Assignment in CollectCast

The previous section detailed how CollectCast selects the best active peers set to render good quality. This section describes how CollectCast coordinates the active peers by assigning the appropriate rate and data portion to each. The assignment is based on the offered rate of each active peer and the current loss rate in the network. Before presenting the assignment methods, we first explain the role of FEC in CollectCast.

Forward Error Correction (FEC) in CollectCast. We use erasure codes (also known as FEC in the network community) to tolerate packet losses due to network fluctuations and limited peers reliability. The media file is divided into equal-length data segments. Each segment has a size of Δ original packets and is protected using FEC separately. Several FEC techniques such as Reed-Solomon codes and Tornado codes [4] can be used. We use Tornado codes because they are faster to encode/decode, albeit with little decoding inefficiency [4]. We use the notation $FEC(\alpha)$ to indicate that the system can tolerate up to $(\alpha - 1)\%$ packet loss rate. For instance, $FEC(1.25)$ means that a data segment will be successfully reconstructed even if 25% of the sent packets were lost. α is the parameter that defines the current (packet) loss tolerance level in the system. α has two bounds: α_u, α_l, which are the maximum and minimum loss tolerance levels, respectively. These bounds impact the selection of active peers determined by solving the maximization problem (Section 3.2) because the bounds (R_l, R_u) in the constraints (5) are computed as: $R_u = \alpha_u R_0$ and $R_l = \alpha_l R_0$.

Data segments stored at peers are pre-encoded using $FEC(\alpha_u)$. A segment of Δ packets is encoded into $\Delta/(2-\alpha_u)$ packets. For instance, $FEC(1.25)$ on a segment of size 120 packets results in a 160 encoded packets, from which any 120 can reconstruct the original segment. Even though data segments are pre-encoded with α_u, we do not send at aggregated streaming rate of $\alpha_u R_0$ all the time. Rather, we send at $\alpha R_0, \alpha_l \leq \alpha \leq \alpha_u$. α is estimated based on the *current* expected aggregated loss rate $\overline{\boldsymbol{L}}_{\sum}$ using:

$$\alpha = \max(\alpha_l, 1 + \min(\alpha_u, 1 + \overline{\boldsymbol{L}}_{\sum})). \tag{7}$$

$\overline{\boldsymbol{L}}_{\sum}$ is determined as $\overline{\boldsymbol{L}}_{\sum} = \sum_{p \in \mathbb{P}^{actv}} \overline{\boldsymbol{l}}_{p \rightsquigarrow r} R_p / \sum_{p \in \mathbb{P}^{actv}} R_p$, where $\overline{\boldsymbol{l}}_{p \rightsquigarrow r}$ is the expected loss rate on the path $p \rightsquigarrow r$.

Rate assignment. After computing the appropriate aggregate rate (αR_0), each peer p is assigned an actual sending rate $\widehat{R}_p$ proportional to its offered rate:

$$\widehat{R}_p = \frac{\alpha R_0}{\sum_{x \in \mathbb{P}^{actv}} R_x} R_p. \tag{8}$$

Data assignment. The active peers collectively send the media file segment by segment: they all cooperate in sending the first segment, then the second one, and so on. Note that, since the active peers send at rate αR_0, they send only $\Delta/(2 - \alpha)$ packets out of the stored $\Delta/(2 - \alpha_u)$ packets. Each peer p is assigned a number of packets D_p to send in proportion to its actual streaming rate:

$$D_p = \left\lceil \frac{\Delta}{(2-\alpha)} \frac{\widehat{R}_p}{\alpha R_0} \right\rceil. \tag{9}$$

Example. Let $\alpha_l = 1.0625$, and $\alpha_u = 1.25$. Assume that the media file is divided into segments each with 120 packets. Encoding with $FEC(\alpha_u = 1.25)$, each encoded segment will have 160 packets. Suppose

that the current active set has three peers P_1, P_2, P_3 with offered rates $R_{P_1} = R_0/2, R_{P_2} = R_0/4, R_{P_3} = R_0/2$, respectively. Assume that the current estimated α is 1.125. Therefore, the assigned rates are: $\widehat{R}_{P_1} = 0.45, \widehat{R}_{P_2} = 0.225, \widehat{R}_{P_3} = 0.45$. The number of packets that need to be sent is 138, and the data assignment is: $D_{P_1} = 55, D_{P_2} = 28, D_{P_1} = 55$. Peer P_1 sends packets with sequence numbers from 1 to 55, peer P_2 from 56 to 83, and peer P_3 from 84 to 138.

Discussion. Packet losses in the Internet is known to be bursty, which has a negative effect on the FEC techniques: during a loss burst, the number of lost packets may exceed what FEC can recover. However, authors of [21] have shown that streaming from multiple senders, as in our case, alleviates the effect of loss burstness on FEC. In our usage of FEC, the number of redundant packets sent is proportional to the current loss rate. If loss rate is low (which is a typical case), only a small number of extra packets will be sent, saving network bandwidth. Finally, we note that the data is pre-encoded. Therefore, senders will not have to encode them on the fly. The receiver decodes them on the fly. Tornado codes are quite fast (order of millisecond for decoding), especially when the segment size is small.

5 Monitoring and Adaptation in CollectCast

Once the active peers are selected (Section 3) and each peer is assigned a streaming rate and data portion (Section 4), the streaming session begins. During a long streaming session the environment may change: peers may fail or network paths may become congested. To maintain good streaming quality on the receiver side, CollectCast needs to adapt to these changes. During the session, the receiver collects statistics on the loss rate and streaming rate contributed from each sending peer. These statistics are used to update the goodness topology, which is then used to *adjust* the active set.

Peer failure. A peer failure is detected in two ways: (1) from the TCP control channel established between the receiver and each of the sending peer (e.g., connection reset), and (2) if the rate coming from this peer is degraded. Once a failure is detected, the active set is adjusted by replacing the failed peer with new one(s). We choose the replacement peers using the topology-aware selection (Section 3.2), provided that the currently good peers are part of the new active set. This may not yield a globally optimal solution, but it is more practical for two reasons. First, the newly chosen set can be totally different from the old one, which will require tearing down all of the old connections and establishing new ones. Second, notice that the topology is partially updated, since for the standby peers, we use the information gathered at the beginning of the streaming session. Thus, it is better to keep peers that are currently doing well. After determining the new active set, the receiver sends a control packet to each peer in the set. The control packets contain the rate and data assignment, computed as explained in Section 4, for each peer.

Network fluctuations. The receiver procedure (Figure 2) makes a *switching* decision after receiving each segment of the media file. A segment is in the order of few seconds. Switching means one of two actions: (1) assigning new rates for the currently active peer set, or (2) adjusting the active set by adding or replacing peers. After receiving a segment, the receiver computes $\gamma = (R_{\sum} - R_0)/R_0$, where $R_{\sum}$ is the aggregate rate measured during the last segment. A value of $\gamma < 0$ means that the network is dropping more than the current loss tolerance level α allows. In this case, the receiver tries to increase α to reach the desired R_0. It computes a new value for α using the updated topology. If the new α exceeds the upper bound α_u, a new active set is selected using the topology-aware selection. Otherwise, a new rate and data assignment is computed using the new α. If γ is positive but less than a threshold (e.g., 0.1), we do nothing: the current setting is good to achieve the target rate with a reasonable FEC overhead. If γ is larger than the threshold, a decrease in α is appropriate. A new smaller α is computed and used to assign rate and data to peers.

6 Topology Inference and Labeling in CollectCast

The topology-aware selection algorithm of CollectCast (Section 3.2) relies on the goodness topology, which is a transformed version of the topology inferred and labeled through end-to-end probing techniques. In this section, we describe our approach to inferring and labeling an *approximate* topology just sufficient for peer selection. Discovering the interior characteristics of the network by probing only from its end points is called *network tomography* [11]. Our approach is a mix of a number of modified versions of known techniques. Our modifications significantly reduce the overhead and lead to a much shorter convergence time. We first construct the logical topology, and then we annotate it with the available bandwidth and loss rate. More details can be found in [16].

Building the logical topology. This is a straightforward step in which a tool like traceroute is used to build the physical topology. Traceroute is performed in parallel from all the candidate peers to the receiver. Then, consecutive links with no branching points are merged together into one segment, resulting in the logical topology. We note that some routers do not support traceroute. This, however, does not severely harm the technique because we are not interested in the exact topology, but in the shared segments among peers.

Annotating the topology with available bandwidth. Let us first precisely define the end-to-end *available* bandwidth of a path. As spelled out by [17], it is the *maximum* rate that the path can provide to a flow, without reducing the rate of other traffic. The link with the minimum available bandwidth (i.e., the tight link) determines the path available bandwidth. Measuring the path available bandwidth is costly: one should keep increasing the probing traffic rate till at least it reaches (probably exceeds) the available bandwidth on the tight link. Measuring the available bandwidth on individual path segments is even more costly. Our approach trades-off the *unnecessary* accuracy of available bandwidth for far less overhead. It accomplishes this through three ways: (1) instead

of measuring the path available bandwidth, we test whether a path can accommodate the aggregated rate from peers sharing this path. This rate is at most R_0. R_0 is typically less than 1 Mb/s, (2) we conservatively label all segments of a path with the value of its tightest segment, and (3) we construct the probing packets from the *actual* data (i.e., data from the media file that will be sent anyway).

Jain and Dovrolis [17] show that the one-way delay differences of a periodic packet stream is a good indication of the available path bandwidth between two nodes. The idea is that if the streaming rate is higher than the available bandwidth, the one-way delay difference will show a trend of increase. This is because packets will be queued at the tight link. On the other hand, if the streaming rate is lower than the available bandwidth, the one-way delay difference will be zero. Then, to measure the bandwidth, the sender sends a stream of packets with a specific rate. The receiver measures the trend in the delay difference and decides whether the next stream rate should be increased or decreased by a factor of 2. The procedure continues till the available bandwidth is estimated within the desired range of accuracy. We make two adaptations to the basic procedure. First, we set the initial stream rate as the minimum possible offered rate (R_p^{min}) from a peer. And we terminate whenever the stream rate reaches the minimum of R_0 and the aggregate rate from peers sharing the path. Second, since one peer may not be able to send at rate R_0, we *coordinate* the probing from multiple peers to get the same effect as probing from one sender.

To illustrate, consider measuring the bandwidth in the topology shown in Figure 4. Let us estimate the bandwidth on the path segment $5 \rightarrow 3$. Peer P_5 sends a stream of packets (say 100 packets) with rate $R_0/8$. The receiver P_r notices that the delay differences are 0. Then P_5 increases its rate to $R/4$. Still no increasing trend in the delay differences but P_5 can not increase its rate anymore. Now P_r triggers P_6 to start sending at $R_0/4$ while P_5 is still sending making the aggregate rate crossing $5 \rightarrow 3$ to be $R_0/2$. P_r measures the delay differences for the packet stream coming from P_5, that is, the stream coming from P_6 is considered as cross traffic to reduce the available bandwidth seen by P_5. P_6 keeps increasing its rate till it reaches its maximum ($R_0/2$) or P_r notices increasing delay differences. If the former happens, segment $5 \rightarrow 3$ will assumed to have an available bandwidth of $0.75R_0$, even though it might have much more available bandwidth. In the latter case, the exact available bandwidth on $5 \rightarrow 3$ will be measured, which in this example is $0.5R_0$. The available bandwidth on $4 \rightarrow 3$ and $2 \rightarrow 1$ can be measured in a similar way. To measure the available bandwidth on $3 \rightarrow 1$, P_r will coordinate the sending from P_3, P_4, P_5, P_6. A final note: suppose that the available bandwidth on $3 \rightarrow 1$ is less than that on $5 \rightarrow 3$, say $R_0/4$. In this case, the technique will underestimate the available bandwidth on $5 \rightarrow 3$ because P_r will see increasing delay differences due to the tight link $3 \rightarrow 1$. This conservative estimation will make the expected rate computed from Equation (4) even worse for set of peers that has a tight shared segment, helping the selection algorithm to avoid them as a solution.

Annotating the topology with loss rate. Instead of explicitly probing for segment-wise loss rates, we leverage the information obtained during available bandwidth measurements. The receiver assigns the sending rate to each

of the sending peers. It also determines which data packets should be sent by each peer. Therefore, it is easy to determine the loss rates on individual end-to-end paths. To compute the segment-wise loss rates, we use the recently proposed *Bayesian inference using Gibbs sampling* method [23]. The method models the network tomography (for segment-wise loss rates) as a Bayesian inference problem. Then, using the measured data and an assumed initial distribution for the segment losses, the method iteratively computes the posterior distribution of the segment losses [23].

Overhead estimation. We consider two types of overhead: processing and communication. The communication overhead is due to the probing packets. However, as noted above, we send actual data packets as probes. Thus, effectively, we do not introduce communication overhead. The receiver, though, needs a larger buffer (in the order of seconds) to store these data packets for later use. The processing overhead is mainly due to topology inference and peer selection. This is not much of a concern, given that the topology will typically be very small (10 to 20 nodes). We note that building the topology and determining the best active set will increase the start up delay, which is the initial time before starting playing back the media file. However, it is still in the order of seconds. Finally, the need for updating the topology will be infrequent, since the active set is expected to last for a relatively long period. This is because: (1) peers in this set are carefully chosen and will likely have high availability (i.e., low probability of failures), and (2) several Internet measurement studies (see for example [39]) have shown a fairly good stability in path properties such as loss, delay, and throughput. Through extensive measurements, authors of [39] conclude that loss, delay, and throughput properties exhibit a constancy on at least time scales of minutes.

Discussion. Ideally, CollectCast will leverage some public Internet measurement facilities, if they are widely deployed. CollectCast can query the measurement facility about the network conditions of the paths connecting the candidate peers with the receiver. The measurement facility will be utilized by many users and applications. Therefore, more accurate measurements can be performed and the overhead will be amortized over all applications. Recently, Internet measurement facilities have started to appear in the literature, see for example [33, 19].

7 TCP-Friendliness and Congestion Control in CollectCast

CollectCast employs UDP to transport data packets from supplying peers to the receiving peer. UDP is characterized as an *unresponsive* protocol because it does not react to congestion in the network. Therefore, applications using UDP protocol may compete unfairly with those that use responsive protocols such as TCP. Competing unfairly means that the UDP applications take larger share of the link bandwidth than TCP-compliant application. This concern is a bit alleviated in CollectCast because, although CollectCast sends using UDP, the sending rate is *upper-bounded* by the offered of the supplying peer. This offered rate is a fraction of the required streaming

rate, since multiple peers cooperate to provide the full rate. Thus, CollectCast will not grab a significant portion of the link bandwidth.

To further increase the friendliness of CollectCast to responsive protocols, we extend its functionalities as follows. For every peer p in the candidate set, CollectCast: (1) computes the TCP-friendly transmission rate ($R_p^{(TCP)}$), and (2) uses the *minimum* of a peer's offered rate (R_p) and the TCP-friendly rate ($R_p^{(TCP)}$) in the selection algorithm (Section 3) and in the data and rate assignment algorithms (Section 4). The TCP-friendly rate is the rate that a compliant TCP sender would send at under the current network conditions [15, 22]. This can be computed using the formula given in the RFC 3448 [15]:[4]

$$R_p^{(TCP)} = \frac{s}{T_{p \rightsquigarrow r}\sqrt{\frac{2}{3l_{p \rightsquigarrow r}}} + 12T_{p \rightsquigarrow r}\sqrt{\frac{3}{8l_{p \rightsquigarrow r}}}l_{p \rightsquigarrow r}(1 + 32l_{p \rightsquigarrow r}^2)}, \tag{10}$$

where: s is the packet size in bytes, which is fixed at 1 KB in CollectCast; $T_{p \rightsquigarrow r}$ is the average round trip time between the sending peer p and the receiving peer r; and $l_{p \rightsquigarrow r}$ is the average end-to-end loss rate between p and r.

This extension to CollectCast does not impose additional overhead. $T_{p \rightsquigarrow r}$ and $l_{p \rightsquigarrow r}$ are initially computed during the topology inference phase (from the probing traffic) and continuously updated throughout the streaming session for the active peers. These values are computed by the receiver and sent to the senders in control packets whenever a change in the rate/data assignments or sender switching is needed. The average loss rate $l_{p \rightsquigarrow r}$ is computed using the statistical method described in RFC 3448, Section 5 [15]. This method assigns weights to the n most recent samples in order to yield smooth changes in the measured loss rate. To measure the average round trip time $T_{p \rightsquigarrow r}$, the receiver time stamps the control packets sent to the senders. Each sender uses the timestamp as a reference and echoes it back with the first data packet. In successive packets, the sender increases the reference timestamp by the inter-packet sending time. When the receiver gets a packet, it computes $T_{p \rightsquigarrow r}$ as the difference between the current time and the timestamp of the packet. Note that, no clock synchronization is needed because the sender is using the timestamp of the receiver as a reference.

8 Evaluation

In this section, we evaluate the performance of CollectCast using extensive simulations. We first present the setup and parameters used in the simulation. Then, we compare the performance of the topology-aware selection (the selection technique used in CollectCast) versus the performance of the end-to-end and random selection techniques. The performance metrics are the aggregated streaming rate and packet loss rate at the receiver. Finally, we assess the impact of peer availability on the size of the candidate peer set, estimate the average size

[4] We follow the simplifying recommendations devised by the authors of [15] in setting the retransmission time out (RTO) as 4 times the round trip time and the number of packets acknowledged by a single TCP ACK to 1 (i.e., TCP with no delayed ACK).

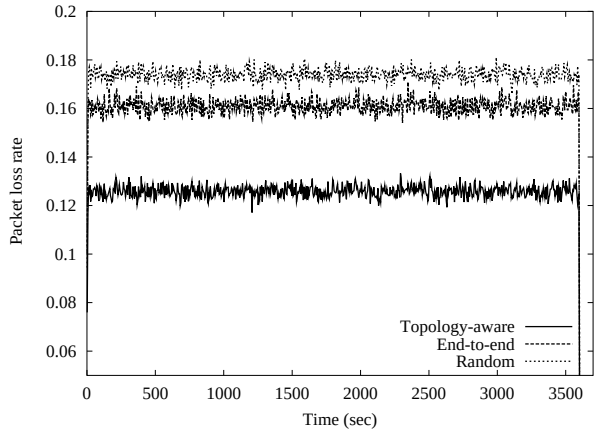


Figure 8: Aggregated loss rate perceived by the receiver: no peer failures.

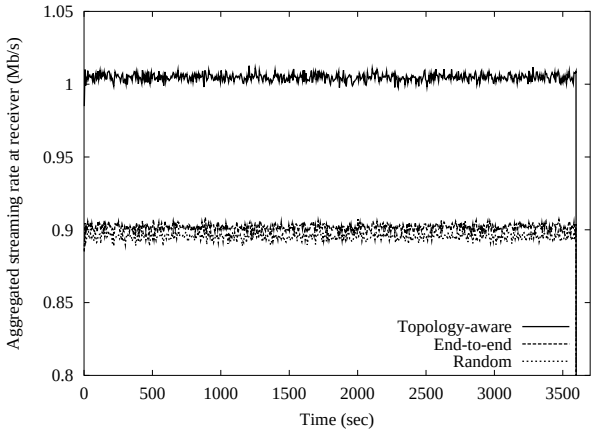


Figure 9: Aggregated streaming rate at the receiver: no peer failures.

of the active peer set during the streaming session, and estimate the expected load on the sending peers.

8.1 Simulation Setup

Simulation topology. We use a hierarchical topology in the simulation. I has three levels. The highest level is composed of transit domains, which represent large Internet Service Providers (ISPs). Stub domains; which represent small ISPs, campus networks, moderate-size enterprise networks, and similar networks; are attached to the transit domains on the second level. Some links may exist among stub domains. At the lowest level, the end hosts (peers) are connected to stub routers. The first two levels are generated using the GT-ITM tool [5]. We then, probabilistically add hosts to stub routers. Each experiment was run on several different topologies. The topologies used in the experiments have, on average, 600 routers and 1,000 hosts (peers).

Simulation parameters. Imposing cross traffic over such a large topology is not feasible. Instead, we approximate the effect of cross traffic by: (1) attaching a stochastic loss model to the links, and (2) randomly setting the links bandwidth to capture the available bandwidth on them. We use the two-state Markov loss model (aka Gilbert model), which was shown to model the Internet packet losses with a reasonable accuracy [38, 18]. In this model, the loss process is modeled as a Markov chain with two states: good and bad. In the good state, the probability of losing a packet is very small and typically assumed to be zero. In the bad state, the probability of losing packets is assumed to be 1.0. The model has two parameters, which are the transition probabilities between the good and bad states.

The available bandwidth on each link is chosen uniformly at random in the range $[0.25R_0, 1.5R_0]$. Peers' parameters are chosen to reflect the diversity in the P2P community [32]. The availability of peers ($\boldsymbol{A}_p$) is distributed uniformly in the range $[0.1, 0.9]$. The offered rate (R_p) is also distributed uniformly in the range $[0.125R_0, 0.5R_0]$. No peer can support more the $R_0/2$ and many of them provide a small fraction of R_0. The

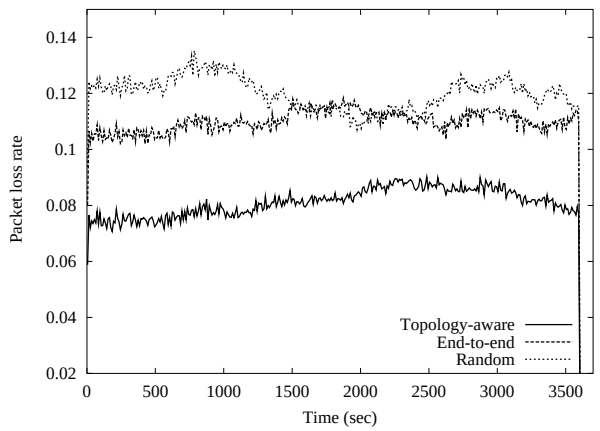


Figure 10: Aggregated loss rate perceived by the receiver: with peer failures.

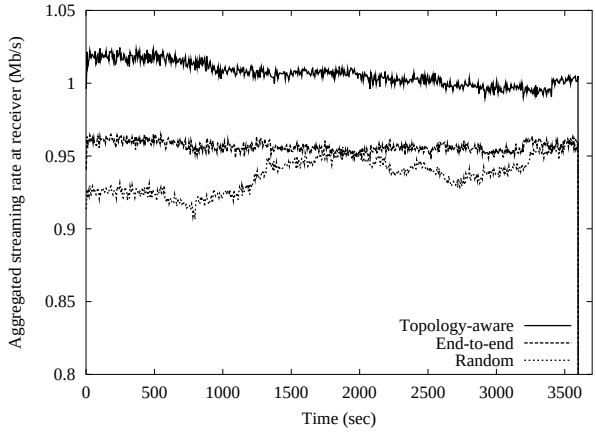


Figure 11: Aggregated streaming rate at the receiver: with peer failures.

streaming session lasts for 60 minutes and the streaming rate R_0 is 1 Mb/s. Every experiment is performed 100 times with different seeds, and the results are averaged over all runs.

8.2 Performance of the Topology-Aware Selection

This section demonstrates the importance of peer selection. It compares the performance of the topology-aware selection (the selection technique used in CollectCast) versus the performance of the end-to-end and random selection techniques. The performance metrics are the aggregated streaming rate and packet loss rate at the receiver. Two scenarios are presented. In the first scenario, we do not simulate peer failures, while in the second scenario we do simulate peer failures and switching.

We simulate a streaming session as follows. First, we randomly select a number of candidate peers (e.g., 20 peers) and a receiver from the the 1,000-peer community. Then, we select the active peer set using either the random, end-to-end, or topology-aware selection (Section 3). Each session is run three times with the same parameters, albeit each run with a different peer selection algorithm. Peers in the active set start streaming till a switching is needed. The loss tolerance level α_u is set to 1.2. We are interested in measuring two metrics: the aggregated loss rate and the aggregated streaming rate perceived by the receiving peer. These two metrics are important since they determines the media playback quality.

Results with no peer failures. Figure 8 depicts the aggregate loss rate seen by the receiver for the three selection techniques. The topology-aware selection achieves lower loss rate (13%) than those of end-to-end (17%) and random (18%) selection. The aggregated loss rate is high in this experiment because we set the available bandwidth on the links in the range $[0.25, 1.5]$ Mb/s. We do that to stress the selection techniques. The aggregated rate perceived by the receiver is shown in Figure 9. The topology-aware technique yields a steady aggregated rate of 1.0 Mb/s, which achieves full playback quality. The end-to-end technique performs better

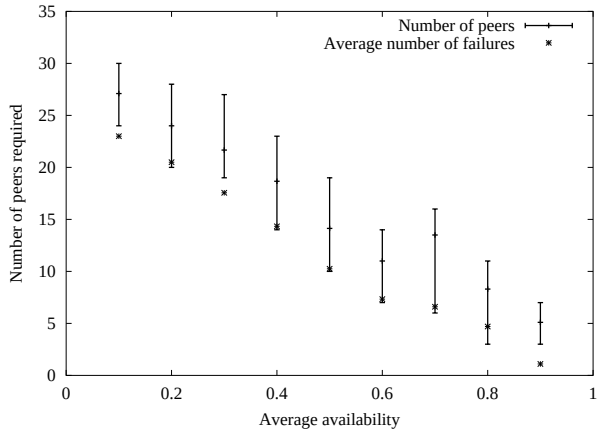


Figure 12: Size of the candidate peers set required for different average peer availability values. The mid-point is the mean, the lower point is the minimum, and the top point is maximum number of peers required in the candidate set.

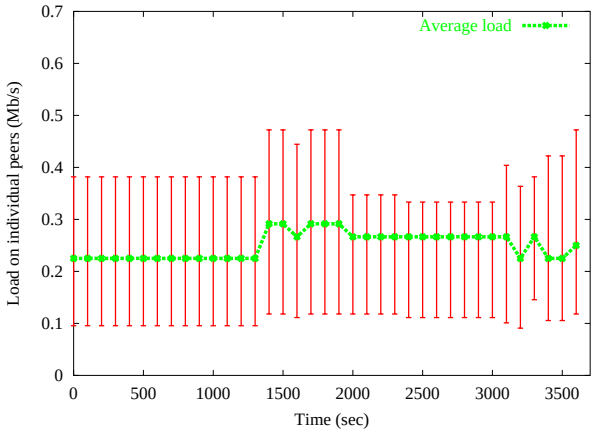


Figure 13: Load on individual peers in presence of failures. Mid-point is the mean load, while the lower and upper points are the minimum and maximum load, respectively.

than the random technique. However, neither of them can achieve full playback rate. This shows the importance of supplying peer selection under the same peer and network conditions. Similar results have also been obtained under other topologies and different loss rate and available bandwidth.

Results with peer failures. During the streaming session, a peer may fail with a probability that is inversely proportional to its availability. We simulate peer failures as follows. We schedule a fixed number of *failure trials* at random times throughout the streaming session. At each failure trial, a peer is selected randomly from the active set and we fail it probabilistically according to its availability: we generate a random number between 0 and 1. If this number is greater than the peer's availability, the peer is failed. Otherwise, the peer remains active and the session continues normally till the next failure trial. The intuition behind this failing method is that if we have many failure trials, each peer will get enough trials to be tested. The fraction of the 'no-failure' trials will approximately be its availability.

Figures 10 and 11 show the aggregated loss rate and the aggregated streaming rate, respectively, in the presence of peer failures. The topology-aware selection still performs better than the other two techniques, achieving a lower loss rate and maintaining full playback quality. Note that, in Figure 11, the aggregated rate is slowly decreasing as the session progresses. This is because as the time elapses, more peers fail and the selection technique is left with fewer peers in the standby set to choose from. This suggests that if we expect many peer failures, the candidate set should be large enough in order to maintain full playback quality, and the size of the candidate set should be chosen properly.

8.3 Candidate Set, Active Set, and Load on Peers

In this section, we study three aspects of CollectCast. First, we assess the impact of peer availability on the size of the candidate peer set. The size of the candidate peer set is an important parameter because it allows us to configure the P2P lookup substrate to return the appropriate number of peers. Choosing the size of candidate peer set arbitrarily may yield poor performance. If the size is too small, CollectCast may run of peers during the streaming session because of peer failures. In this case, a new request is issued to the P2P lookup substrate to return more peers, which may cause long period of disruption. On the other hand, if the size is too large, the overhead imposed during the construction of the topology will be higher and the selection algorithm may take unnecessarily long time to determine the active set. Second, we estimate the average size of the active peer set during the streaming session. This indicates the average number of connections that a receiving peer may need to maintain concurrently. Third, we estimate the expected load on the sending peers in terms of how much rate each sending peer contributes to the streaming session.

Impact of peer availability on the size of the candidate set. In this experiment, we estimate the size of the candidate set for different values of peers availability. We vary the average availability of peers from 0.1 to 0.9. A total of 25 failure trials are scheduled during each streaming session. If a failure trial is successful (i.e., we fail a peer), a replacement peer (or peers) will be chosen. We run the simulation 10 times for each value of peer availability and count the total number of peers that are needed to complete the session. Figure 12 shows the impact of peer availability on the size of candidate set. The figure shows the average number of successful failure trials (out of 25) and the minimum, mean, and maximum number of peers required in the candidate set as the average availability grows from 0.1 to 0.9, over the 10 simulation runs. For example, for an average peer availability of 0.6, we need an average of 11 peers in the candidate set, and a maximum of 14 will guarantee that we will not run of out peers in the candidate set. Figure 12 shows that as the availability increases, the number of peers needed in the candidate set decreases. We are deriving a more rigorous and generic relation between the size if candidate set and peer availability based on streaming session duration, peer failure model, and network failure model.

Size of the active set. The receiving peer establishes concurrent connections with all peers in the active set. Each connection adds overhead on the receiver: more buffers are allocated and more control packets are sent. Using the same parameters as in Section 8.1, we conducted several experiments to estimate the average size of the active set. As shown in Figure 14, we find that the average number of active peers is fairly small, less than four most of the time and it does not depend on the availability.

Load on individual peers. Peers are not dedicated server machines. Therefore, it is important to limit the load (in terms of the streaming rate) on peers. We sample the load on each active peer during many streaming sessions that allow peer failures. We take a sample every 100 seconds. Figure 13 shows that the average load on individual peers is between 0.22 Mb/s and 0.3 Mb/s. In very few times, a peer is assigned a 0.45 M/s, provided

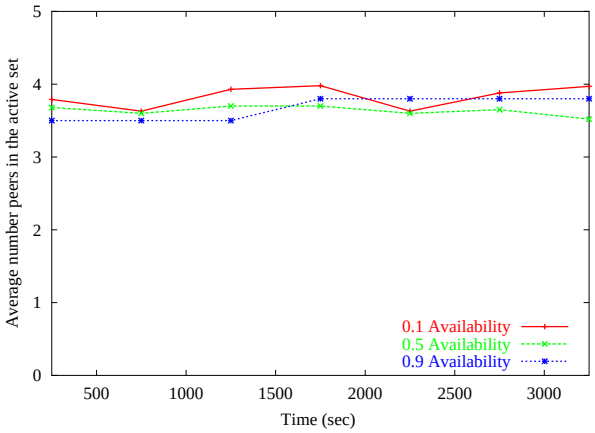


Figure 14: Average number of peers in the active set for different average peer availability.

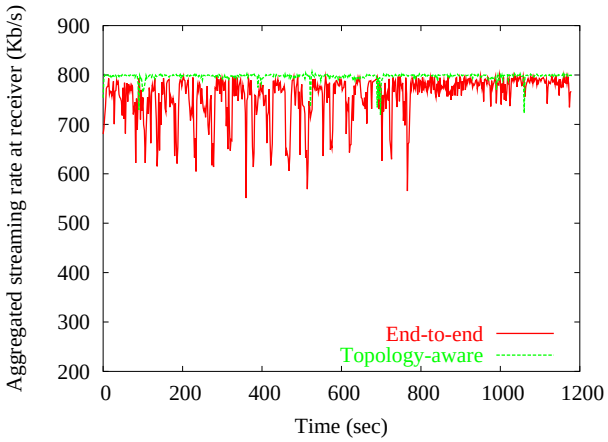


Figure 15: Effect of the careful peer selection on the aggregated received rate.

Table 1: The MPEG-4 movie traces used in the experiments.

Movie title	**Average rate Kb/s**	**Peak rate Kb/s**	**Size Mbyte**	**Streaming rate R_0 Kb/s**
Star Wars IV	287.21	1874.00	43.08	400.00
The Firm	364.72	2020.40	54.71	400.00
Aladdin Cartoon	402.90	2559.80	60.44	400.00
From Dusk Till Dawn	576.12	3106.00	86.42	800.00

that it can support it, i.e., it offered rate is greater than 0.5 Mb/s. In the first 1,200 seconds the average is small because during that period there were no failures. As we encounter more failures, and thus smaller candidate set and fewer options, the average increases and becomes closer to the maximum.

9 PROMISE and Experiments on PlanetLab

To assess the performance of CollectCast in real environments, we have implemented a P2P media streaming system on top of CollectCast. We call this system PROMISE. PROMISE has been tested in both local and wide area environments. In the implementation, we use Pastry (code obtained from [14]) as the P2P lookup substrate. We have modified Pastry to support multiple peer lookup. The code runs as an agent on each participating peer. To test the code in the wide area environment, we have installed PROMISE agents on 15 nodes of the PlanetLab test bed [26]. The nodes chosen for the experiments are distributed over different geographic locations.

We have conducted an extensive experimental study to assess the performance of PROMISE from several angles. In this sections, we present four sets of results. The first set presents the packet-level performance, which considers the aggregated rate measured at the receiver and how it changes with the time. The second set addresses the frame-level performance. This focuses more on the perceived quality quantified in terms of the number of frames that either miss their deadlines or lost. The third set studies the the impact of changing the system parameters on the quality. In the fourth set, we show how PROMISE handles peer failure and switching.

9.1 Packet-Level Performance

In this set of experiments, we focus on the *raw* aggregated received rate measured by the receiver. The setup is as follows. The receiver is located at the UC Berkeley peer. The remaining 14 peers constitute the set of candidate peers. We construct and annotate the topology connecting the candidate peers with the receiving peer. We construct the topology using the `tracepath` tool, which is similar to the `traceroute` tool but it does not require superuser privileges. We measure the available bandwidth using `pathload` [25]. After annotating the topology, we choose the active peer set using two selection techniques: topology-aware and end-to-end. We compare the aggregated rate achieved by the peers chosen by each of the two selection techniques. The streaming session lasts 20 minutes. The playback rate R_0 is 800 Kb/s. The dynamic peer switching as well as the FEC encoding are turned off. We repeat the streaming session five times and compute the average aggregated received rate. The results shown in Figure 15 demonstrates the potential gain achieved from the topology-aware selection of peers. The aggregated rate from peers selected by the end-to-end technique varies widely and sometimes drops below 600 Kb/s. Whereas, the aggregated rate from peers carefully chosen by the topology-aware technique is smooth and rarely drops below 750 Kb/s. The reason is that the end-to-end technique selected two peers (one at Caltech and the other at UCSD) that share a tight segment, which could not support the aggregated rate from both of them. The topology-aware technique avoided that segment and chose a better active set: it has two peers, one at Rice and one at UCSD. This confirms with our simulation results in Section 8.

9.2 Frame-Level Performance and Initial Buffering

We study the quality of playback of the streamed movies. We quantify the quality by the number of frames that either: (i) miss their playback deadlines or (ii) are lost. We differentiate among the two cases because a larger initial buffering time could mitigate the first case, while it does not affect the second one (unless if we employ a retransmission technique). Moreover, higher values for α (loss tolerance level) may recover lost frames but it has a little impact on the delayed frames. In this set of experiments, we study the impact of the initial buffering on the quality. We also compare the buffer size required by the topology-aware and the end-to-end selection techniques.

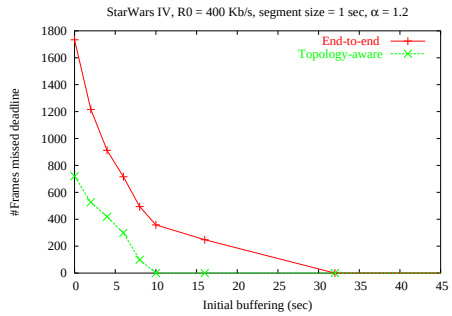
We use video traces of several movies encoded using MPEG-4. The traces were obtained from [35]. We use the verbose versions of the traces. Each row of the trace file has four entries: frame number, frame type (I, P, or B), frame playout time, and frame length in bytes. The frame playout time is relative to the first frame playout time, which is set to zero. The movie titles and some statistics about them are listed in Table 1. We stream only the first 20 minutes of each movie, that is, we stream 30,000 frames of each movie because all movies have a frame rate of 25 frames per second. The setup of these experiments is similar to the setup of the previous set of experiments, except that the FEC encoding is enabled. We set $\alpha = 1.2$ and the segment size equals 1 second. We record the arrival time of each single packet. After the termination of the streaming session, we determine the

number of frames that would have missed their deadlines for a specific initial buffering time. To decide whether a frame f misses its deadline, we compare two values: $f_{deadline}$ and f_{avail}. If $f_{deadline}$ is greater than f_{avail}, f misses its deadline. $f_{deadline}$ is the sum of the frame playout time (read from the trace file) and the initial buffering time. f_{avail} is the time at which all packets constituting f are successfully reconstructed by the FEC decoder and are available in the buffer. We determine which packets the frame f occupies by using the frame length field in the trace file.

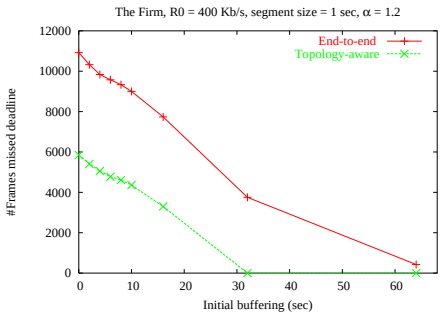
Figure 16 shows the results for four different movies: Star Wars IV, The Firm, Aladdin Cartoon, and From Dusk Till Dawn. For each movie, we repeat the session five times and plot the average. The first observation is that peers selected by the topology-aware technique require much less initial buffering in the all four cases. To ensure full quality, i.e., no frame misses its deadline, the topology-aware technique requires, on the average, less than half of the initial buffering required by the end-to-end technique. The second observation is that the total number of frames that miss their deadlines depend on the movie characteristics and the streaming rate R_0. For example, in Figure 16.a, the initial buffering needed to ensure the full quality is fairly small (about 10 seconds) for the topology-aware selection. Also, the number of frames that missed their deadlines is relatively small for buffering less than 10 seconds. This is because the average and peak rates of the Star Wars IV movie are only 287.21 Kb/s and 1874.00 Kb/s, respectively, and we stream at R_0 = 400 Kb/s. In contrast, we need a larger initial buffering in the case of The Firm (Figure 16.b) and Aladdin Cartoon (Figure 16.c) because the average and peak rates are higher in these two cases. This implies that selecting the appropriate streaming rate for each movie has a direct impact on the quality. Rate smoothing techniques can be used to estimate the streaming rate for each movie. Another approach is to leverage the characteristics of the cooperative P2P environment: peers that received the movie in the past may share their experiences with the currently requesting peer.

9.3 Impact of Changing System Parameters

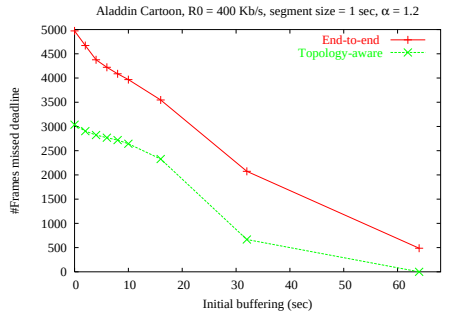
We inspect the effect of the two main system parameters, the loss tolerance level α and the segment size, on the quality. We also assess the tradeoffs and overhead associated with various values of these parameters. In the first experiment (Figures 17 and 18), we fix the segment size at one second and we vary α from 1.0 to 1.8. We calculate the number of segments that can not be decoded by FEC. FEC fails to reconstruct a segment if more than $(\alpha - 1)\%$ of the packets are lost or corrupted. For each undecodable segment, we mark all of its packets as lost and count the number of frames that use any of these lost packets. We consider these frames as lost frames, since we do not employ any error concealment or frame interpolation techniques. Figure 17 compares the number of undecodable segments (left Y-axis) and the number of lost frames (right Y-axis) resulted from the topology-aware and the end-to-end selection techniques for different values of α. The number of undecodable segments in the end-to-end technique is about six times larger than that in the topology-aware technique. Higher loss tolerance levels, although desirable, come at a higher price: more redundant traffic is sent as shown in Figure



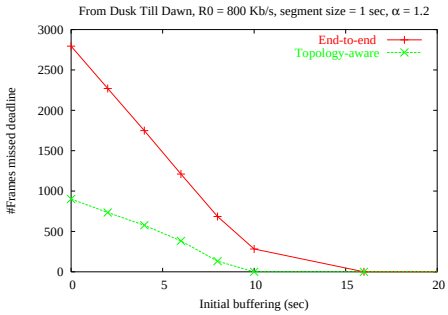
(a) Star Wars IV.



(b) The Firm.



(c) Aladdin Cartoon.



(d) From Dusk Till Dawn.

Figure 16: Frame-level performance: initial buffering needed to ensure full quality. The topology-aware selection requires much smaller initial buffering than the end-to-end selection to ensure that all frames meet their deadlines. Traces from four different movies are used in the experiments.

18. The results in Figure 17 and Figure 18 implies that the topology-aware technique is able to reconstruct all segments with a moderate redundant traffic (α is in the range [1.1–1.2]).

In the second experiment (Figures 19 and 20), we vary the segment size from 0.1 to 16 seconds. We fix the initial buffering time at a specific value, and count the number of frames that missed their deadlines. We conduct five streaming sessions and report the average. Then, we repeat the whole experiment for a different initial buffering time. We use only the topology-aware selection technique and we fix α at 1.2. The results are shown in Figure 19. The general observation is that increasing the segment size has a negative impact on the quality. This is because it takes FEC more time to reconstruct a larger segment than a smaller one. FEC needs to wait for at least $(2 - \alpha)\Delta$ packets to arrive in order to decode a segment of size Δ packets. Larger Δ means more packets need to arrive. Furthermore, the decoding time (CPU cycles) of the Tornado codes used in CollectCast is linear in the segment size. We notice that decreasing the segment size below one second has a marginal positive impact on the quality. This is because a portion of the gain we get from fast decoding of small segments is lost due to the more frequent invocations of the FEC decoding routine. As shown in Figure 20, small segments impose more communication overhead on the system. This overhead is due to sending control packets to the senders every segment. The control packets specify the rate at which each sender should send at and the portion of the data that should be sent. The results of this experiment indicate that a segment of size from one to two seconds would strike a balance between the quality and the overhead imposed.

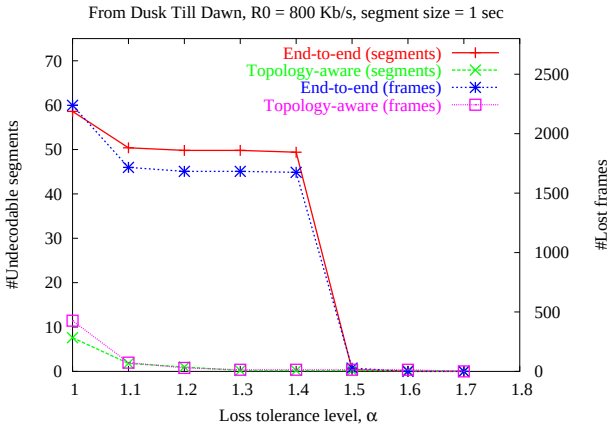


Figure 17: The impact of the loss tolerance level α on the number of undecodable segments and the lost frames.

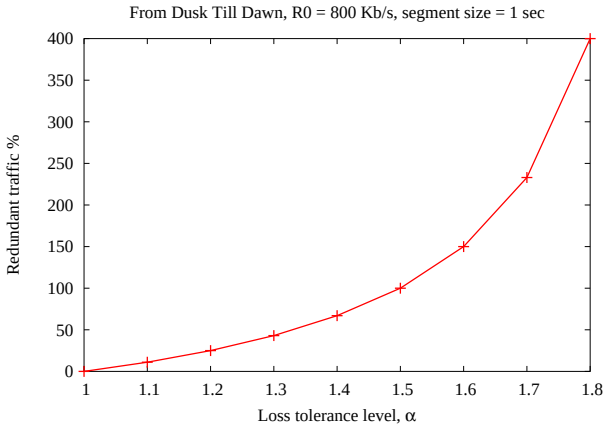


Figure 18: The amount of redundant traffic sent by the senders for various values of the loss tolerance level α.

9.4 Peer Failure and Switching

This experiment is conducted on heterogenous nodes outside the PlanetLab test bed to demonstrate code portability. We assess the monitoring and adaptation component of CollectCast. We intentionally fail peers and let CollectCast detect and react to the failure. The receiver is located at Purdue University. Six candidate peers were chosen for this streaming session: purdue1 and purdue2 at Purdue University but in two different subnets, uconn at University of Connecticut, gatech at Georgia Institute of Technology, uiuc at University of Illinois, and toronto at University of Toronto. The active set initially has four peers: purdue1, purdue2, uconn, and gatech. The aggregate streaming rate is 450 Kb/s. Figure 21 shows the results from multiple failure-prone peers serving a streaming session. After 385 seconds, we fail purdue2. CollectCast detects the failure and purdue2 is replaced by uiuc. The switching is fast and it does not significantly affect the aggregated rate. Another failure is scheduled at time 780 with a similar quick response from CollectCast.

10 Related Work

In the last few years, the P2P paradigm has received tremendous attention from researchers. Two main categories of research can be identified: research on protocols and algorithms (such as searching and replication), and research on building P2P systems. The first category aims at building scalable and efficient P2P infrastructure (substrate), which could be used for systems in the second category. Lookup (or routing) protocols such as CAN [27], Chord [34], and Pastry [30] guarantee locating the requested object within a logarithmic number of steps, if the object exists in the system. However, network locality has not been amply exploited (except in case of Pastry). Examples of P2P systems include CFS [12] on top of Chord [34], and PAST [31] on top of Pastry [30].

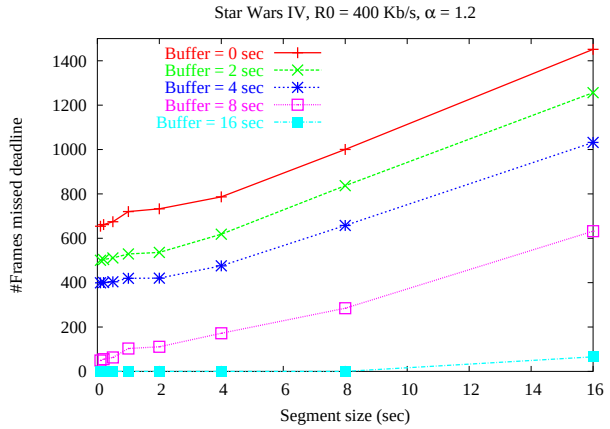


Figure 19: The impact of the segment size on the quality for different values of the initial buffering.

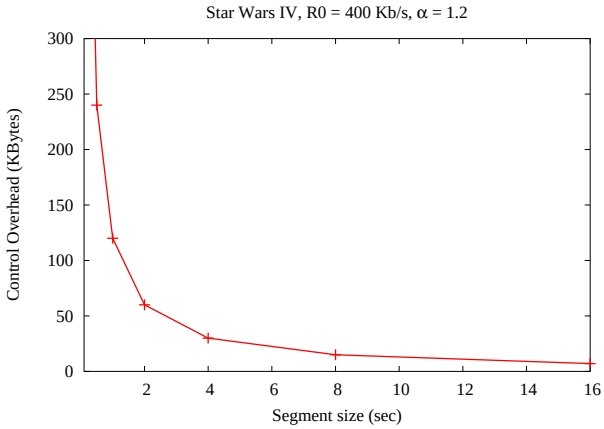


Figure 20: The communication overhead imposed by sending control packets for different segment sizes.

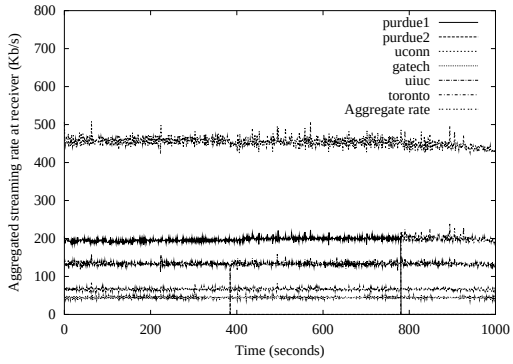


Figure 21: Streaming from multiple peers. Two supplying peers failed at times 385 and 780. CollectCast detects and replaces the failed peers.

Another example is Pixie [29]: a P2P content exchange architecture. Pixie aggregates requests from multiple peers and multicasts content to the requesting peers. These systems do not target media streaming. Therefore, unlike CollectCast, they do not consider real-time and sending rate requirements for P2P data transmission.

Application level multicast (ALM) is proposed to overcome the limited deployment of IP multicast. Each ALM-based system has its own protocol for building and maintaining the multicast tree. For example, both NICE [1] and Zigzag [36] adopt hierarchical distribution trees and therefore scale to a large number of peers. Narada [9], on the other hand, targets small scale multi-sender multi-receiver applications. Narada maintains and optimizes a *mesh* that interconnects peers. The optimized mesh yields good performance but it imposes maintenance overhead. SpreadIt [13] constructs a distribution tree rooted at the sender for a live media streaming session. A new receiver joins by traversing the tree starting at the root till it reaches a node with sufficient remaining capacity. CoopNet [24] supports both live and on-demand streaming. It employs multi-description coding and constructs multiple distribution trees (one tree for each description) spanning all participants. SplitStream [7] provides a cooperative infrastructure that can be used to distribute large files (e.g., software updates) as well as

streaming media. SplitStream is built on top of Scribe [8], a scalable publish-subscribe system that employs Pastry [30] as the lookup substrate. The content in SplitStream is divided into several *stripes*, each is distributed by a separate tree. Different from these systems, PROMISE is proposed for the streaming of media data from multiple senders to one receiver. And CollectCast is another P2P service complementing the ALM service.

Many P2P data sharing and distribution systems implicitly assume that a sending peer is capable of supporting one or more receiving peers. However, it has been shown that peers are heterogeneous in their capability and/or willingness to contribute resources to other peers [32]. Few systems have considered the problem of selecting multiple supplying peers (senders) for a receiver, based on peer heterogeneity as well as network topology and conditions. CollectCast addresses this problem.

The distributed video streaming framework [20, 21] shows the feasibility and benefits of streaming from multiple servers to a single receiver. The receiver uses a rate allocation algorithm to specify the sending rate for each server in order to minimize the total packet loss. This specification is based on estimating the end-to-end loss rate and available bandwidth between the receiver and each server. However, the framework is not explicitly designed for P2P environments. Therefore, it does not address the selection and dynamic switching of senders.

Finally, Rodrigues and Biersack [28] show that parallel download of a large file from multiple replicated servers achieves significantly shorter download time. The subsets of a file supplied by each server are dynamically adjusted based on the network conditions and the server load. This work targets bulk file transfer, not real-time media streaming. Moreover, it does not consider sender selection nor does it leverage network tomography techniques.

11 Conclusion and Future Work

This paper presents a novel and comprehensive P2P media streaming service, CollectCast. The most salient features of CollectCast include: (1) it accounts for peer heterogeneity, reliability, and limited capacity, (2) it matches a requesting peer with a set of supplying peers that are likely to achieve the best streaming quality, (3) it dynamically adapts to network fluctuations and peer failure, and (4) it performs (2) and (3) by inferring and leveraging the underlying network conditions. Our simulations demonstrate that significant gain in the streaming quality can be achieved by CollectCast even in the presence of peer failures. The simulations also show that CollectCast does not burden the participating peers: we show that on average a sending peer contributes less than a quarter of the required streaming rate. In addition, we have implemented a P2P media streaming system (called PROMISE) on top of CollectCast. Results from testing PROMISE on the PlanetLab test bed confirm that streaming from multiple failure-prone peers in a dynamic P2P environment is indeed feasible. Specifically, we show that the full quality can be maintained in the presence of failures and losses. Furthermore, the results obtained from streaming several MPEG-4 movies indicate that applications built on top of CollectCast can achieve better

performance in two angles: packet-level and frame-level. In the packet-level performance, the aggregated rate is much smoother in streaming sessions that employ CollectCast than those that do not use it. In the frame-level performance, CollectCast reduces the number of frames that miss their deadlines by about 50% under the same initial buffering time.

CollectCast can be extended beyond the physical network characteristics and streaming applications. For example, CollectCast may take peers' social properties such as credibility and trustworthiness into consideration. One can imagine a graph showing the topology formed by the candidate suppliers and the receiver, but the links are labeled with trust-related metrics. This will enable security-sensitive applications to choose the best peers that will supply the most trusted data or service.

References

[1] S. Banerjee, B. Bhattacharjee, C. Kommareddy, and G. Varghese. Scalable application layer multicast. In *Proc. of ACM SIGCOMM'02*, pages 205–220, Pittsburgh, PA, USA, August 2002.

[2] M. Bawa, H. Deshpande, and H. Garcia-Molina. Transience of peers and streaming media. *First Workshop on Hot Topics in Networks (HotNets 2002)*, October 2002.

[3] A. Bestavros, J. Byers, and K. Harfoush. Inference and labeling of metric-induced network topologies. In *Proc. of IEEE INFOCOM'02*, New York, NY, USA, June 2002.

[4] B. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proc. ACM SIGCOMM'98*, pages 56–67, Vancouver, British Columbia, August 1998.

[5] K. Calvert, M. Doar, and E. Zegura. Modeling Internet topology. In *IEEE Communications Magazine*, pages 35:160–163, 1997.

[6] K. Calvert, J. Griffioen, B. Mullins, A. Sehgal, and S. Wen. Concast: Design and Implementation of an Active Network Service. *IEEE Journal on Selected Area in Communications*, 19(3):426–437, March 2001.

[7] M. Castro, A. Druschel, P. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth content distribution in a cooperative environment. In *Proc. of 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003.

[8] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8):1489–1499, October 2002.

[9] Y. Chu, S. Rao, S. Seshan, and H. Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(8):1456–1471, October 2002.

[10] M. Coates, R. Castro, and R. Nowak. Maximum likelihood network topology identification from edge-based unicast measurements. In *Proc. ACM SIGMETRICS 2002*, Marina Del Rey, CA, USA, June 2002.

[11] M. Coates, R. Hero, A. Nowak, and B. Yu. Internet tomography. *IEEE Signal Processing Magazine*, 19(3), 2002.

[12] F. Dabek, M. Kaashoek, D. Karger, D. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, October 2001.

[13] H. Deshpande, M. Bawa, and H. Garcia-Molina. Streaming live media over peer-to-peer network. Technical report, Stanford University, 2001.

[14] Free pastry home page. http://www.cs.rice.edu/CS/Systems/Pastry.

[15] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP friendly rate control (TFRC): protocol specification. Request for Comment, RFC 3448, January 2003.

[16] M. Hefeeda, A. Habib, B. Boyan, D. Xu, and B. Bhargava. PROMISE: peer-to-peer media streaming using CollectCast. Technical report, CS-TR 03-016, Purdue University, August 2003. Extended version.

[17] M. Jain and C. Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput. In *Proc. of ACM SIGCOMM'02*, pages 295–308, Pittsburgh, PA, USA, August 2002.

[18] V. Markovski, F. Xue, and L. Trajkovic. Simulation and analysis of packet loss in user datagram protocol transfers. *The Journal of Supercomputing*, 20(2):175–196, 2001.

[19] A. Nakao, L. Peterson, and A. Bavier. A routing underlay for overlay networks. In *Proc. ACM SIGCOMM'03*, Karlsruhe, Germany, August 2003.

[20] T. Nguyen and A. Zakhor. Distributed video streaming over Internet. In *Proc. of Multimedia Computing and Networking (MMCN02)*, San Jose, CA, USA, January 2002.

[21] T. Nguyen and A. Zakhor. Distributed video streaming with forward error correction. In *Proc. Int'l Packetvideo Workshop (PV'02)*, Pittsburgh PA, USA, April 2002.

[22] J. Padhye, V. Firoiu, D. Towsley, and J. Kuros. Modeling tcp throughput: A simple model and its empirical validation. In *Proc. of ACM SIGCOMM'98*, pages 303–314, Vancouver, Canada, August 1998.

[23] V. Padmanabhan, L. Qiu, and H. Wang. Server-based inference of Internet link lossiness. In *Proc. of IEEE INFOCOM'03*, San Francisco, CA, USA, April 2003.

[24] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai. Distributing streaming media content using cooperative networking. In *Proc. of NOSSDAV'02*, Miami Beach ,FL, USA, May 2002.

[25] Pathload home page. http://www.cc.gatech.edu/fac/Constantinos.Dovrolis/pathload.html/.

[26] Planetlab home page. http://www.planet-lab.org/.

[27] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of ACM SIGCOMM'01*, San Diego, CA, USA, August 2001.

[28] P. Rodriguez and E. Biersack. Dynamic parallel access to replicated content in the Internet. *IEEE/ACM Transactions on Networking*, 10(4):455–465, August 2002.

[29] S. Rollins and K. Almeroth. Pixie: A jukebox architecture to support efficient peer content exchange. In *Proc. of ACM Multimedia*, Juan Les Pins, France, December 2002.

[30] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.

[31] A. Rowstron and P. Druschel. Storage management in past, a large-scale, persistent peer-to-peer storage utility. In *Proc. of 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[32] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of Multimedia Computing and Networking (MMCN02)*, San Jose, CA, USA, January 2002.

[33] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public Internet measurement facility. In *Proc. 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Seattle, Washington, USA, March 2003.

[34] I. Stoica, R. Morris, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of ACM SIGCOMM'01*, San Diego, CA, USA, August 2001.

[35] MPEG-4 movie traces. http://www-tkn.ee.tu-berlin.de/research/trace/ltvt.html.

[36] D. Tran, K. Hua, and T. Do. Zigzag: An efficient peer-to-peer scheme for media streaming. In *Proc. of IEEE INFOCOM'03*, San Francisco, CA, USA, April 2003.

[37] D. Xu, M. Hefeeda, S. Hambrusch, and B. Bhargava. On peer-to-peer media streaming. In *Proc. of IEEE ICDCS'02*, Vienna, Austria, July 2002.

[38] M. Yajnik, S. Moon, J. Kurose, and D. Towsley. Measurement and modeling of the temporal dependence in packet loss. In *Proc. of IEEE INFOCOM'99*, pages 345–352, York, NY, USA, March 1999.

[39] Y. Zhang, N. Duffield, V. Paxon, and S. Shenker. On the constancy of Internet path properties. In *Proc. of ACM SIGCOMM Internet Measurement Workshop*, San Francisco, CA, USA, November 2001.