# Accelerating Peer-to-Peer Networks for Video Streaming Using Multipoint-to-Point Communication

*Hung-Yun Hsieh and Raghupathy Sivakumar*
*Georgia Institute of Technology*

## ABSTRACT

Existing transport layer protocols such as TCP and UDP are designed specifically for point-to-point communication. The increased popularity of peer-to-peer networking has brought changes in the Internet that provided users with potentially multiple replicated sources for content retrieval. However, applications that leverage such parallelism have thus far been limited to non-real-time file downloads. In this article we consider the problem of multipoint-to-point video streaming over peer-to-peer networks. We present a transport layer protocol called R$^2$CP that effectively enables real-time multipoint-to-point video streaming. R$^2$CP is a receiver-driven multistate transport protocol. It requires no coordination between multiple sources, accommodates flexible application layer reliability semantics, uses TCP-friendly congestion control, and delivers to the video stream the aggregate of the bandwidths available on the individual paths. Simulation results show great performance benefits using R$^2$CP in peer-to-peer networks.

## INTRODUCTION

Over the last few years, the area of peer-to-peer overlay networks has attracted considerable attention. The notion of end users collaborating to support a richer set of network services with no network infrastructure support has been quite well received. This is not surprising given the understandably slow rate of deployment for any service that requires changes to the Internet infrastructure.

One interesting communication paradigm that has in particular come into the limelight with the advent of peer-to-peer networking is multipoint-to-point communication. A single "client" (requesting peer) can use multiple "servers" (supplying peers) to access the desired content, and gain from the resulting parallelization of the access. Multipoint-to-point communication has become particularly relevant in the context of peer-to-peer networks because of the following reasons:

- Since peers act as content servers, it is more likely that for any given request there are multiple sources available to serve the same content. Moreover, the availability of the content increases as its popularity increases. This is very different from the conventional client-server model where the hosts that serve the content are limited to those allocated by the content provider.
- In a peer-to-peer network, the *uplink* (outbound) from a peer server is not guaranteed to be sufficiently provisioned. Consider, for example, a peer using an asymmetric digital subscriber line (ADSL) connection for network access. Although the downstream bandwidth can be on the order of a few megabits per second, the upstream bandwidth is typically an order of magnitude smaller. Hence, it is very likely that when a peer client accesses the content, the bottleneck really lies at the peer server end.

Unsurprisingly, several peer-to-peer applications such as Kazaa have started using multiple connections to replicated sources for expediting data transfer. These applications, however, are limited to non-real-time downloads and cannot be used for real-time streaming, despite the fact that the most popular type of files shared in peer-to-peer networks is multimedia content such as music and video [1]. In this work we focus on the problem of enabling real-time video streaming over peer-to-peer networks using multipoint-to-point communication. The goal is to allow users to start watching the video after a short preroll delay (a few seconds), without having to wait (for potentially hours) until the content is completely downloaded on the local disk. The key challenges that render related work ineffective in supporting multipoint-to-point communication for real-time video streaming

*This work was funded in part by the National Science Foundation under grant ANI-0117840.*

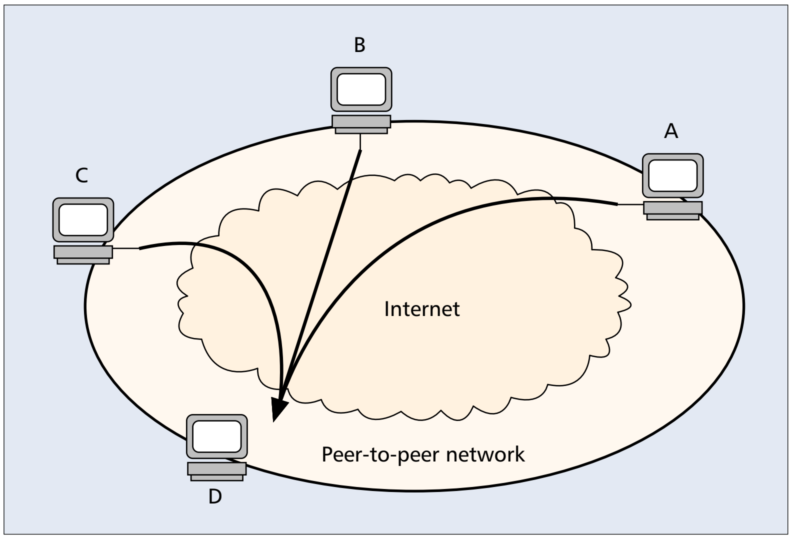

**Figure 1.** *Multipoint-to-point communication.*

over peer-to-peer networks stem from the unique requirements of the target application and environment:

**Video streaming:** While it is possible for solutions targeting non-real-time applications to fetch different pieces of the accessed content independently and then perform offline resequencing on the disk [2, 3], such solutions cannot be used for real-time video streaming where ordering of content (in-sequence delivery) is necessary. When a finite resequencing buffer is used in such approaches, the very fact that individual paths carry dependent information that needs to be delivered in a globally sequenced manner can result in head-of-line blocking and buffer overflow, causing the aggregate throughput to be throttled by the slowest path [4].

**Peer-to-peer networks:** While several approaches proposed for real-time streaming in content delivery networks leverage the existence of multiple content servers [5, 6], such solutions cannot be used for peer-to-peer networks due to the "non-server-like" behaviors of peers acting as sources, such as heterogeneous capacity and transient availability. Moreover, many peers may not be encountered more than once; hence, any approach that relies on the design of optimal server placement or content distribution is not applicable to peer-to-peer networks.

In this article we present a transport layer protocol called Radial Reception Control Protocol ($R^2$CP) that enables effective multipoint-to-point video streaming over peer-to-peer networks.[1] Briefly, $R^2$CP is a purely receiver-driven multistate transport protocol. It requires no explicit coordination between multiple sources, consumes minimal computing resources at the sources, operates in a TCP-friendly fashion for individual paths, seamlessly accommodates flexible application layer reliability semantics, and effectively delivers the aggregate of the bandwidths available on individual paths. While the cornerstones of the $R^2$CP design are in fact also applicable to multipoint-to-point non-real-time content download, we restrict the focus of all our discussions in this article to real-time video streaming. We show through packet-level simulations that $R^2$CP achieves effective multipoint-to-point real-time video streaming.

The rest of the article is organized as follows. We first elaborate on the problem and the key research challenges in supporting multipoint-to-point video streaming over peer-to-peer networks. We then describe the $R^2$CP protocol, including its software architecture and protocol operations. Finally, we present performance results for $R^2$CP, and conclude the article.

*[1] The $R^2$CP protocol was proposed in [7] for non-real-time data transfer on mobile hosts with heterogeneous wireless interfaces. In this article we show that the principal design elements of $R^2$CP with appropriate modifications allow it to achieve effective multipoint-to-point video streaming over peer-to-peer networks.*

## The Problem

In this section we elaborate on the problem considered in this article. We first discuss the goal and the key challenges. We then show that existing approaches cannot effectively address these challenges, thus motivating a new approach to tackle the problem.

### Goal

The use of peer-to-peer applications has so far been limited to non-real-time file downloads. Any file transferred using such applications needs to be completely downloaded on the local disk before it can be used. However, the most popular type of files shared in peer-to-peer networks is multimedia content such as music or video [1]. If a user with a 1.5 Mb/s ADSL connection is to download a 90 min VCD movie encoded at 1.38 Mb/s (standard rate for 352 × 240 pixel resolution VCD movies) in such a *use-after-download* mode, it will take the user at least 83 min before he/she can start watching the movie. Such a long waiting delay can be avoided if the user is provided with the ability to *stream video while downloading*. In this article we aim to enable such real-time video streaming applications for users in peer-to-peer networks.

Unlike in conventional streaming applications where the servers are provided through the content delivery networks, in peer-to-peer streaming the host supplying the video clip typically does not fit the high-bandwidth low-latency profile of a server [1]. For peers with an ADSL connection, for example, the uplink data rate is limited to less than 512 kb/s (or 256 kb/s for typical cable modem users). As mentioned earlier, if a user is to stream a VCD movie, the average streaming rate should be at least 1.38 Mb/s — about three times the maximum outbound rate from any supplying peer using an ADSL connection. Conventional approaches that open a single unicast connection between the streaming server and the client thus will fail to provide users in peer-to-peer networks with the ability to play back while downloading. Notwithstanding such limitations, interestingly the peer-to-peer network is also characterized by a high degree of content replication due to individual peers acting as both clients and servers. For any content query, existing peer-to-peer lookup protocols can efficiently locate multiple peers with the desired content. Therefore, in this article we target a solution where the requesting peer uses a multipoint-to-point connection, as shown in Fig. 1, to simultaneously stream content from multiple supplying peers for achieving the desired playout rate.

## Challenges

An important issue in realizing the benefits of multipoint-to-point communication is the ability to aggregate the bandwidths available along multiple paths (pipes) from individual sources to the destination. We identify two key challenges involved in achieving multipoint-to-point video streaming over peer-to-peer networks.

**Peer heterogeneity:** The characteristics of the peers participating in peer-to-peer networks exhibit a very high degree of heterogeneity. For example, the authors in [1] find that the downstream bottleneck bandwidth in the peer-to-peer network measured varies from less than 1 Mb/s (bottom 20 percent) to more than 10 Mb/s (top 20 percent), while that of the upstream link varies from less than 100 kb/s to more than 5 Mb/s. Moreover, the average path latency varies from less than 70 ms (lower 20 percent) to more than 280 ms (higher 20 percent). Since real-time streaming applications typically use a finite playout buffer to allow for a small preroll delay, it is critical that packets arrive in the order they will be consumed by the application to minimize losses due to buffer overflow and deadline expiry. It is a nontrivial packet scheduling problem to minimize out-of-order arrivals and packets missing deadlines when the video is streamed across multiple paths with potentially mismatched bandwidths and delays (and fluctuations thereof due to network congestion or user activity).

**Peer transience:** While servers in content delivery networks are required to be highly available to serve content, this is not the case for peers voluntarily participating in content sharing. The authors in [1] show that most sessions in a peer-to-peer network are relatively short, with the median approximately equal to 60 min. A similar study [8] shows that around 60 percent of the hosts in the network keep active for no longer than 10 min. Therefore, it is very likely that during content retrieval the sharing peers could disconnect from the network or abort the connection. Any streaming protocol thus needs to handle peer transience such that the dynamic departures of sources have minimal impact on the performance perceived at the requesting peer.

## Related Work

As mentioned earlier, although existing peer-to-peer applications like Kazaa can use multiple sources to speed up downloads, they are not applicable to real-time video streaming that uses a finite buffer and requires in-sequence delivery. Similarly, related work such as [2, 3] is designed for non-real-time file downloads, and hence cannot be used to provide the desired solution.

In [5] the authors propose an approach that uses multiple description coding (MDC) for video streaming from multiple servers in content delivery networks. Multiple complementary descriptions for a video stream are created and distributed across the edge servers (surrogates) in the network. The authors consider the problems of server coloring (code distributions), server placement, and server selection for optimizing client performance. The proposed approach, however, is not applicable to peer-to-peer networks, where the distribution of the peers participating in streaming is not static, and the path characteristics are not known a priori. The overheads incurred in applying MDC across paths with mismatched data rates (e.g., the need to estimate data rates for achieving unbalanced encoding or quality adaptation) can further make such an approach undesirable in peer-to-peer networks. In [6] the authors propose an approach that streams video from multiple distributed video servers. This approach requires periodic synchronization between servers in terms of sending rates and sending sequences. However, synchronization introduces overheads in terms of packet duplicates or losses. Moreover, since servers are in control of the streaming process, such an approach will suffer from significant connection disruptions or abortion in peer-to-peer networks due to dynamic peer departures.

In [9] the authors consider peer-to-peer streaming using multiple sources. They focus on the problem of data assignment when the bandwidth available at each source is predetermined. The proposed solution is later extended to a new system in [10]. In the proposed approach, the receiver maintains with each source a UDP connection for streaming, and a TCP connection for sending control information such as rate and data assignment. The receiver periodically monitors (probes) the status of peers and connections. With knowledge of the offered rate and loss rate along each path, the receiver computes the ideal streaming rates and data portions that should be served by individual sources, and updates the assignment whenever network fluctuations and peer failures are detected. While the proposed approach addresses peer heterogeneity and transience, it requires the service provided by lower layers for measuring the offered rate, loss rate, and round-trip time (for calculating the TCP-friendly rate). Moreover, the use of separate TCP connections for sending assignment information incurs additional overhead whenever updates are necessary due to network fluctuations. In this article we propose an approach that allows the receiver to stream from multiple sources by maintaining a multipoint-to-point connection, without the need to use explicit network measurements and control channels.

*Periodic synchronization introduces overheads in terms of packet duplicates or losses. Moreover, since servers are in control of the streaming process, such a sender-centric approach will suffer from connection disruptions or abortion in peer-to-peer networks due to dynamic peer departures.*

## The R$^2$CP Protocol

We now present the R$^2$CP protocol that addresses the challenges identified above and enables effective multipoint-to-point video streaming over peer-to-peer networks.

### An Architectural Overview

R$^2$CP is a receiver-driven multistate transport protocol that supports both unicast (one source) and multipoint-to-point (multiple sources) connections. As we show in Fig. 2, an R$^2$CP connection with $k$ sources can be decomposed into the following two components:

- $k$ RCP pipes that connect individual senders (sources) to the receiver (destination)
- An R$^2$CP engine that coordinates multiple RCP pipes at the receiver

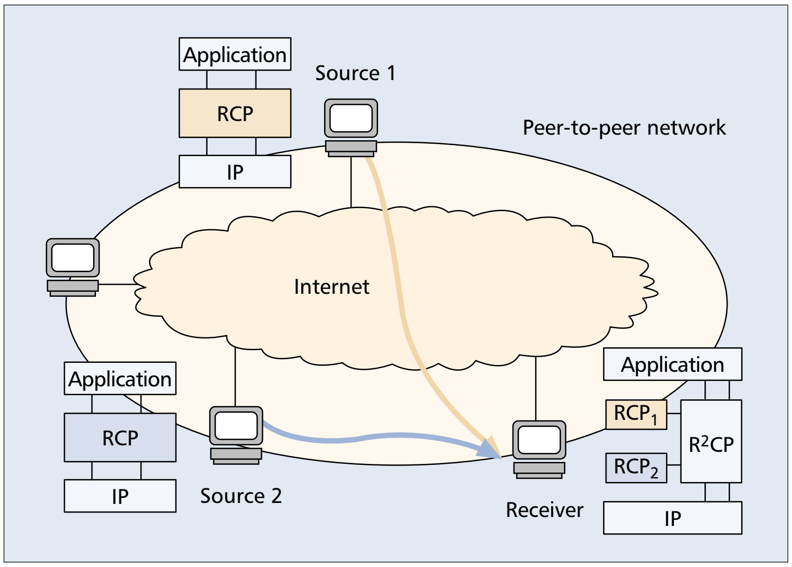

**Figure 2.** *R$^2$CP for multipoint-to-point communication.*

The R$^2$CP engine resides only at the receiver, and is transparent to the senders as far as the protocol handshake along each RCP pipe is concerned. RCP by design is a TCP clone that reuses most of the transport mechanisms in TCP, including a window-based congestion control mechanism such as *streaming-friendly* binomial congestion control [11]. The difference, however, is that RCP transposes the intelligence of TCP from the sender to the receiver such that the RCP receiver drives the operations of the protocol, while the RCP sender merely responds to the instructions (requests) sent by the receiver. The transposition of functionalities allows the receiver to be in charge of the connection progression along each pipe, including congestion control and loss recovery,[2] and hence facilitates the coordination task performed by the R$^2$CP engine. Note that using sender-centric protocols such as TCP for individual pipes is not a scalable option, as the geographically distributed sources would have to be explicitly coordinated to support the same semantics.

The receiver in an R$^2$CP connection is the primary seat of intelligence for most protocol functionalities, including congestion control, loss recovery, and packet scheduling. R$^2$CP decouples the protocol functionalities associated with individual paths from those pertaining to the aggregate connection. The per-connection and per-pipe functionalities are handled by the R$^2$CP engine and RCP, respectively. For example, congestion control estimates the available bandwidth of the underlying path, and hence is performed by RCP on a per-pipe basis. On the other hand, packet scheduling pertains to the aggregate connection, and hence is handled by the R$^2$CP engine. In this way individual RCPs track the characteristics of the underlying paths and control *how much data to request* from each source, while the R$^2$CP engine tracks the progression of the connection and controls *which data to request* from each source.

The design of the two-tier architecture together with the decoupling of functionalities provides R$^2$CP with the following benefits in supporting multipoint-to-point video streaming over peer-to-peer networks:

- Since congestion control is performed on a per-pipe basis, any existing congestion control algorithms can be used for each pipe without suffering from performance degradation due to operating over multiple heterogeneous paths.
- Since the R$^2$CP engine is the only entity responsible for scheduling requests of application data, any reliability semantics required by the application can be supported through interfaces with the R$^2$CP engine (e.g., using application-level framing [12]) without interfering with the operations (e.g., congestion control) performed along each pipe. We note that the purely unreliable service provided by UDP is insufficient for video streaming [13].
- Since the R$^2$CP engine is responsible for the aggregate connection, and its operation is internal to the receiver, the shutdown of any RCP source (due to, say, transience of the supplying peer) can have minimal impact on the progression of the connection.

In the following, we discuss how R$^2$CP achieves the decoupling of functionalities through *dynamic binding* of application data.

## Dynamic Binding

Whereas the R$^2$CP engine at the receiver handles the socket buffer and maintains the global sequence number, individual RCPs maintain local sequence numbers for their protocol operations.[3] Hence, even though the congestion control mechanism used in RCP is designed for in-sequence delivery (recall that RCP is a TCP clone where loss recovery and congestion control are coupled), the R$^2$CP engine can retrieve noncontiguous data from each source (to minimize out-of-order arrivals as we discuss in packet scheduling) without triggering the adverse reactions in RCP. The mappings (bindings) between the local and global sequence numbers are maintained by the R$^2$CP engine in the `binding` data structure. As we show in Fig. 3, the R$^2$CP engine is a wrapper around RCP, serving the read/write from the application and IP layers. Any local sequence number used by RCP will be converted to the global sequence number by the R$^2$CP engine before transmissions, and vice versa. (Note that individual RCP senders process the requests from the receiver based on the global sequence number.) From the perspective of the R$^2$CP engine, the `binding` data structure allows it to control which application data should be assigned to (requested from) which RCP pipe depending on its transmission schedule, while still ensuring a contiguous sequence number space for each RCP to perform congestion control.

The binding between the application data and the local sequence number is, however, not static for the following two reasons:

- RCP supports the fully reliable semantics as TCP, and hence uses retransmissions for loss recovery. However, if the R$^2$CP engine deems it unnecessary to recover the lost data (e.g., due to deadline expiry), new

[2] *We refer interested readers to [7] for detailed operations of RCP, including the duality between RCP and TCP, the change from* ***data-acknowledgment*** *to* ***request-data*** *protocol handshake, and its TCP friendliness.*

[3] *For clarity of presentation, in this article we assume that data is packetized in the unit of the maximum segment size (MSS), and hence the sequence number refers to the packet sequence number. We also assume in this article that MSS is the same across all RCP pipes in the connection, although this is not a requirement for the actual protocol operations.*

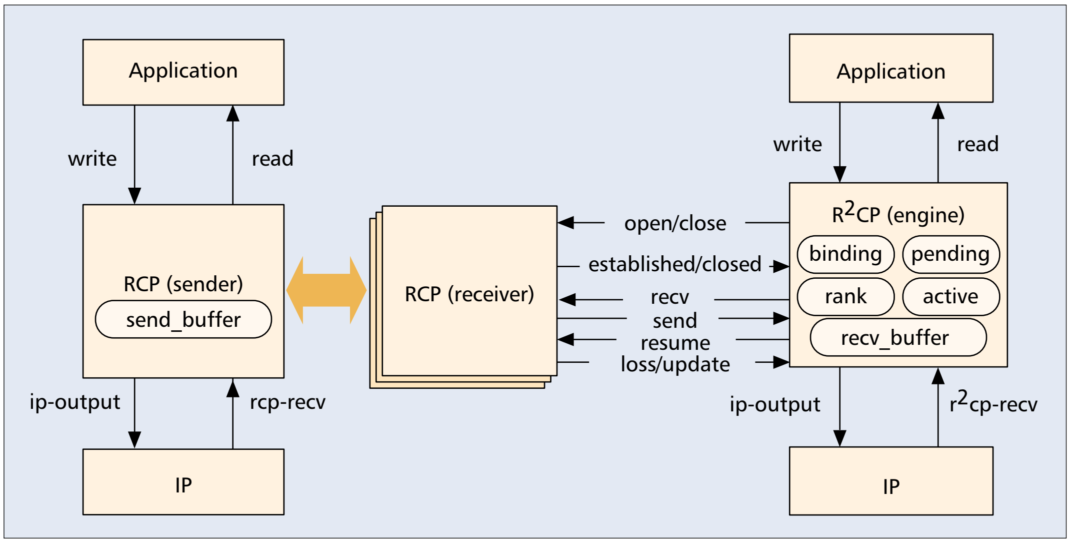

**Figure 3.** *R²CP architecture.*

*The R²CP engine uses the arrivals of data to clock the transmissions of new requests, and schedules a request along a path only when the concerned RCP pipe has space in its congestion window for requests.*

application data will be bound to the retransmission requests. Even if the application data needs to be retransmitted, it can be sent through another pipe with, say, a shorter round-trip time depending on the schedule.

- If an RCP pipe is closed (e.g., due to peer departures), the application data bound to the pipe will be reassigned to other pipes for recovery (whenever appropriate).

Such dynamic binding between the application data and RCP local sequence numbers reduces the role of individual RCPs to providing transmission slots that track the available bandwidth along each path. The R²CP engine can flexibly utilize the transmission slots for packet scheduling, as discussed in the following.

## Packet Scheduling

While individual RCP pipes provide transmission slots for data requests from the sources, a key task that needs to be performed by the R²CP engine is to decide *which* data to request for each transmission slot; that is, to schedule requests of application data. The objective is to minimize the number of packet losses at the receiver due to either buffer overflow or deadline expiry. We observe that *an optimal schedule (that minimizes the number of packet losses) for streaming packets with non-decreasing deadlines to a buffer limited receiver is one that ensures in-sequence arrival of packets at the receiver*. Intuitively, in-sequence delivery minimizes the chances of buffer overflow at the receiver. Moreover, since the deadlines associated with packets of increasing sequence numbers are non-decreasing, in-sequence delivery also minimizes the chances of packets missing the deadlines.

To minimize out-of-order arrivals, the R²CP engine needs information regarding the bandwidths and latencies of individual pipes. Since congestion control is a process of bandwidth estimation, changes in bandwidth and latency are directly reflected through the changes in the size of the congestion window. Hence, the progression of the congestion window in RCP provides an effective way for the R²CP engine to track the available bandwidth along each pipe. The R²CP engine uses the arrivals of data to clock the transmissions of new requests, and schedules a request along a path only when the concerned RCP pipe has space in its congestion window for requests. Since the request (REQ) in RCP has the dual role of the acknowledgment (ACK) in TCP for congestion control [7], the use of such a fine-grained packet scheduling allows R²CP to closely track the bandwidth fluctuations *without incurring extra overheads compared to TCP*.

While one simple way to schedule a request is to assign the next global sequence number to the new request, such first-come-first-served (FCFS) scheduling will cause out-of-order arrivals due to mismatched latencies along different paths. The assignment instead should be based on the potential order (rank) of data packets arrivals. The R²CP engine maintains a `rank` data structure for finding the rank of the new request. Specifically, whenever the R²CP engine sends out a request for sequence number $i$ through pipe $j$, an entry is added to the `rank` data structure with a timestamp of $S = T_i + RTT_j$, where $T_i$ is the time when the request is sent, and $RTT_j$ is the round-trip time of pipe $j$. The timestamp reflects the time when a new request will be issued in response to the arrival of the requested data. When the R²CP engine receives the *send*() call from pipe $k$ at time $T$, it locates the rank $r$ of the request as

$$r = 1 + \sum_{n=1}^{N} \left\lfloor \frac{T + RTT_k - S_n}{RTT_{p(n)}} \right\rfloor^{+},$$

where $N$ is the total number of entries in `rank`, $S_n$ is the timestamp at entry $n$, $p(n)$ is the pipe that sent out the request at entry $n$, and $\lfloor x \rfloor^+$ is the maximum of zero and the largest integer less than or equal to $x$. In other words, the R²CP engine finds the number of data segments that will be requested after $T$ (due to arrivals of pending requests) but arrive before $T + RTT_k$.

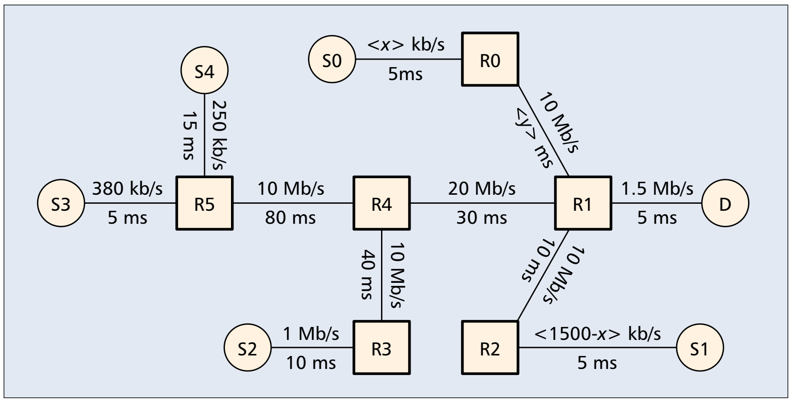

**Figure 4.** *Network topology.*

Once the rank $r$ is identified, the R$^2$CP engine binds the $r$th data in line to the new request, and adds an entry for the request in the `rank` data structure as usual. The entry is deleted when the corresponding data arrives.

### Protocol Operations

We have so far presented the software architecture of R$^2$CP, and its dynamic binding and packet scheduling operations. In this section we present the protocol operations of R$^2$CP using Fig 3.

When pipe $p$ that has space in its congestion window uses the *send*() call (with local sequence number $l$ as the parameter) to request for transmission at time $T$, R$^2$CP locates the rank $r$ of the request using the `rank` data structure as we discussed earlier. The $r$th data to request in the `pending` data structure is identified as the segment with sequence number $g$. The R$^2$CP engine then creates a placeholder (e.g., an `skbuf` without data [14]) in `recv_buffer` expecting the arrival of data segment $g$. It adds for data $g$ an entry $(p, l)$ in the `binding` data structure and an entry $T + RTT_p$ in the `rank` data structure. It finally sends out the request packet to the IP layer using $g$ as the sequence number and then removes $g$ from the pending data structure. The highest sequence number $h$ for data that has been delivered to the application is also included in the request packet.

When the request packet arrives at the sending end of pipe $p$, the sender finds in its `send_buffer` the data with sequence number $g$, and sends out the data packet to the IP layer. The sender then purges data in its buffer with sequence numbers lower than $h$. Since the sender simply echoes whatever data is requested from the receiver, out-of-order or non-contiguous transmissions are possible at individual senders.

When data segment $g$ arrives, the R$^2$CP engine enqueues the data to its placeholder in `recv_buffer`, finds the corresponding RCP pipe $p$ and the local sequence number $l$ based on the `binding` data structure, and passes $l$ to the concerned RCP pipe using the *recv*() call. It then deletes the corresponding entries in the `binding` and `rank` data structures. The concerned RCP pipe updates its states (e.g., congestion window and the next local sequence number to request) and determines whether it can send more requests or not based on the size of its congestion window. If it can generate more requests, it uses the *send*() call with the next local sequence number for transmission request as before. If any loss is detected via three out-of-order arrivals or timeouts [7], the RCP pipe uses the *loss*() call to notify the R$^2$CP engine, which then *unbinds* the global sequence numbers corresponding to the lost packets by deleting the corresponding entries in the `binding` and `rank` data structures. Depending on the reliability semantics required by the application, the unbound sequence numbers will be re-inserted to the `pending` data structure waiting for retransmissions, or simply discarded. If the RCP pipe has a new estimate for RTT, it uses the *update*() call to notify the R$^2$CP engine, which then appropriately updates the `rank` data structure.

Whenever the R$^2$CP engine receives the *send*() call from an RCP pipe but does not have space in its `recv_buffer` to receive more application data (due to, say, application backlog), it returns with `FREEZE` and puts the corresponding RCP in the `active` data structure. Later, if space opens up in its `recv_buffer` (due to, say, application read), the R$^2$CP engine uses the *resume*() call to *de-freeze* pipes in the `active` data structure. Resumed pipes will as usual issue requests through the *send*() call depending on individual states.

The *open*() and *close*() calls are used by the R$^2$CP engine to add or delete RCP pipes in the connection during connection setup and teardown, or even in the middle of the connection. Since the RCP pipes are responsible only for providing transmission slots in the R$^2$CP connection, it is clear that adding or deleting RCP pipes has minimal impact on the operations of R$^2$CP. As mentioned above, whenever an RCP pipe is closed during the progression of the connection, all global sequence numbers bound to the concerned pipe will be unbounded and re-inserted (whenever appropriate) to the `pending` data structure for retransmissions.

## Performance Evaluation

In this section we present the evaluation results for R$^2$CP. We first explain the simulation model based on the *ns*-2 network simulator. We then show the performance of R$^2$CP in achieving effective video streaming using network simulation.

### The Simulation Model

We use the network topology shown in Fig. 4 to evaluate the performance of R$^2$CP. The network topology consists of six routers and six access nodes (peers) connected using duplex links with bandwidths and propagation delays as shown in the figure (note that while all access links have asymmetric bandwidths, the figure only shows the bandwidth in the direction of the multipoint-to-point connection). Node $D$ is the peer requesting the video clip, and nodes $S0$ to $S4$ are peers returned by the peer-to-peer lookup protocol that have the desired content. We assume that the target streaming rate is 1376 kb/s (the standard file size of a VCD movie is 172 kbytes/min). The initial playout delay is set to 5 s.

We first study the performance of R$^2$CP in aggregating the bandwidths available along multiple pipes with mismatched bandwidths and latencies. We assume the content is transferred from two supplying peers (*S*0 and *S*1). We first vary the bandwidth *x* of the bottleneck link between *S*0 and *D* from 200 to 750 kb/s, while that between *S*1 and *D* varies from 1300 to 750 kb/s accordingly (the bandwidth ratio between the two pipes thus varies from 1 to about 6). We then vary the propagation delay *y* on link *R*0–*R*1 from 10 to 150 ms such that the latency ratio between the two pipes varies from 1 to 8. Finally, we introduce bandwidth and delay fluctuations to the two pipes using on/off UDP background traffic traversing from S0 to *R*1 and from *S*1 to *R*1, respectively. We use the Pareto traffic source for both background flows, where the data rate during the burst time from *S*0 is set to 200 kb/s, and that from *S*1 is set to 500 kb/s. The mean burst time is 0.1*t* s, the mean idle time is 0.2*t* s, and the shape parameter is 1.5. We vary *t* from 1 to 100 to observe the scalability of $R^2$CP with the degree of fluctuations in background traffic.

Thereafter, we study the performance of $R^2$CP in achieving multipoint-to-point video streaming. We set the bandwidth of link *S*0–*R*0 to 128 kb/s and the delay of link *R*0–*R*1 to 20 ms, and consider a more complicated scenario where the desired content is streamed from four heterogeneous peers (*S*0, *S*2, *S*3, and *S*4) with bandwidth/delay mismatches and fluctuations due to background traffic. In addition to the target multipoint-to-point connection, we introduce the following background traffic while the streaming takes place:

1) *File download at S*1: A TCP flow from *S*2 to S1 using FTP as the traffic source.

2) *Web browsing at S*3: An on/off UDP flow from *S*3 to *R*0 using the Pareto traffic source with a mean burst time of 1 s, a mean idle time of 2 s, a data rate during burst time of 200 kb/s, and a shape parameter of 1.5 (for emulating the request traffic in the uplink).

3) *Web browsing at S*4: An on/off UDP flow from *S*4 to *R*4, similar to 2) but with the following parameters: 0.5 s, 1 s, 100 kb/s, and 1.5.

4) *Backbone long-lived flows*: Five TCP flows between *R*1 and *R*5 (bidirectional).

5) *Backbone short-lived flows*: 40 on/off UDP flows between *R*1 and *R*5 (in both directions).

All pipes in the multipoint-to-point connection use binomial congestion control [11] with the following parameters: $k = 1.0$, $l = 0$, $\alpha = 1.0$, $\beta = 0.66$. We use TCP/Sack for all TCP flows. The simulation is run for 600 s, and all data points are averaged over 15 runs when randomness is introduced.

## Simulation Results

Figure 5 shows the performance of $R^2$CP to effectively aggregate the bandwidths available along pipes in the multipoint-to-point connection. We compare the performance of $R^2$CP against that of the following approaches:

**Ideal:** The ideal performance for bandwidth aggregation is equal to the throughput sum of individual pipes when independent TCP flows are used (no head-of-line blocking).

**Multiple sockets:** An approach that opens multiple TCP sockets and requests data from

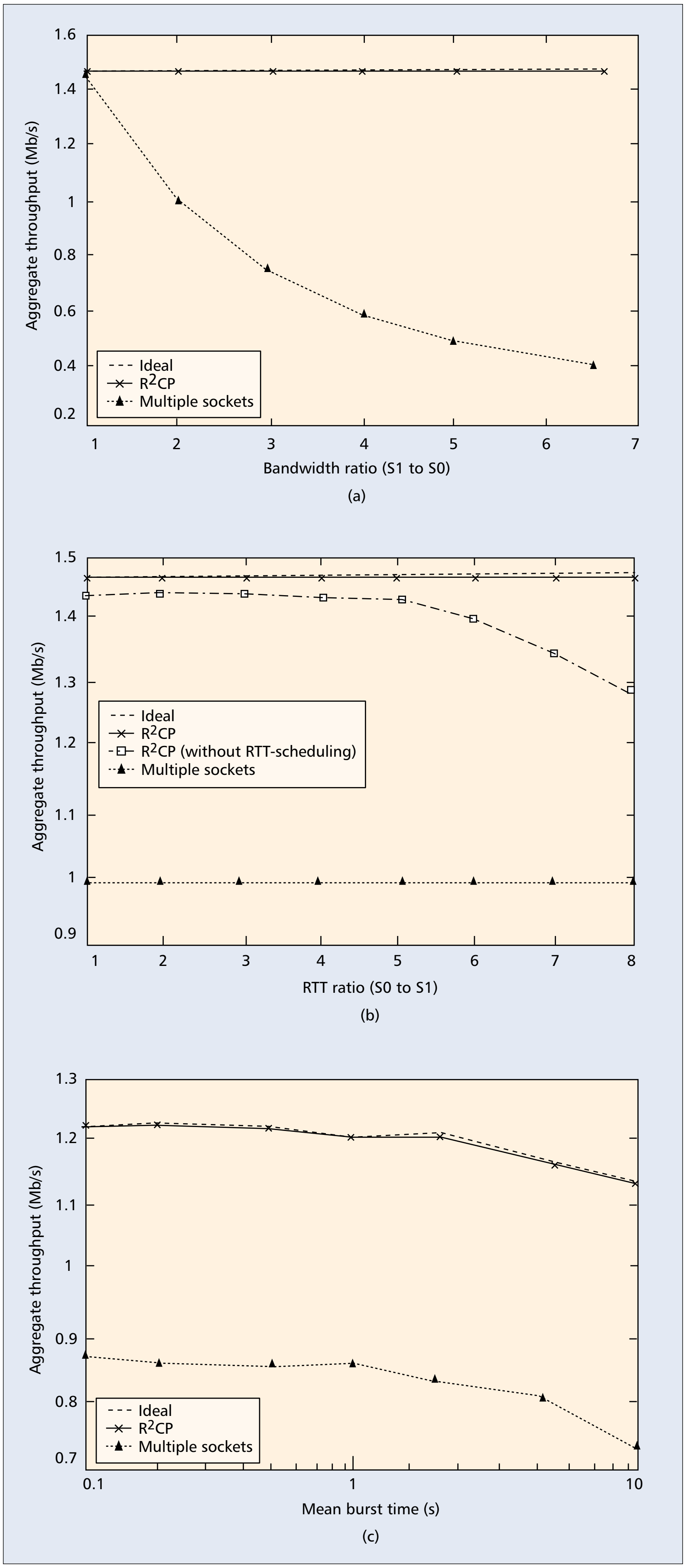

**Figure 5.** *Scalability with peer heterogeneity: a) rate differential; b) delay differential; c) traffic fluctuations.*

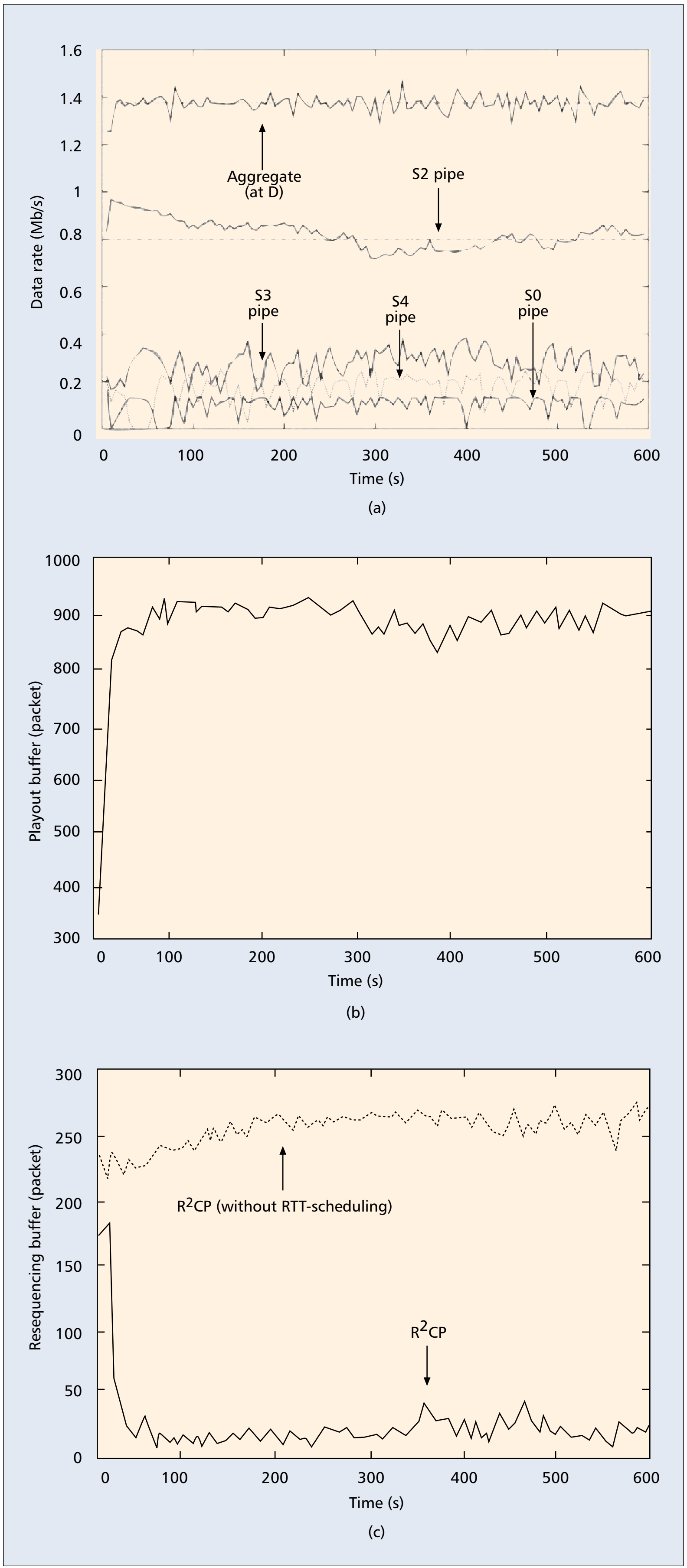

**Figure 6.** *Multipoint-to-point video streaming: a) instantaneous data rate; b) playout buffer occupancy; c) resequencing buffer occupancy.*

each socket in a round-robin fashion. The application uses a finite resequencing buffer to achieve in-sequence delivery. The goal is to show the impact of peer heterogeneity on a pure application layer approach (without transport layer support) that has no knowledge of the network characteristics.

**R$^2$CP without RTT-scheduling:** A simplified version of R$^2$CP that uses FCFS scheduling for assigning packets to individual pipes. Essentially, the R$^2$CP engine simply assigns the next packet in the `pending` data structure to the next pipe that requests to send, without taking into consideration its round-trip time.

It is clear from Fig. 5a that the multisocket approach does not scale when the bandwidths of the two pipes are different. This is due to the application being unaware of the bandwidths available along the paths traversed by individual pipes, which otherwise would require the use of a sophisticated bandwidth estimation and probing mechanisms implemented at the application. R$^2$CP, on the other hand, achieves the ideal performance even when the bandwidth ratio of the two pipes goes beyond 6 (using the same buffer space as in the multisocket approach), by reusing the congestion control mechanism implemented along each pipe for effective striping. As we described earlier, a key element of the R$^2$CP engine is to maintain the `rank` data structure for packet scheduling such that packets requested through different pipes arrive in sequence. We observe in Fig. 5b that such packet scheduling indeed allows R$^2$CP to effectively uses the bandwidths available along different pipes with large RTT mismatches.[4] Finally, we find in Fig. 5c that the performance of R$^2$CP to effectively aggregate the available bandwidths remains valid under bandwidth and delay fluctuations due to background traffic.

We now show the performance of R$^2$CP to use the available bandwidths along individual pipes for achieving effective video streaming. As described above, the connection consists of four supplying peers with bandwidth/delay mismatches and fluctuations due to background traffic. The target streaming rate is 1376 kb/s. Figure 6a shows the instantaneous goodput (receive rate in 5-s bins) observed at the receiver (node $D$), along with the desired streaming rate and the send rates from individual pipes. We can make the following observations from the figure:

- Because of the on/off UDP flows, the individual send rates of the $S3$ and $S4$ pipes show large fluctuation, despite the use of the binomial congestion control scheme. For example, the maximum fluctuation on the $S3$ pipe is 232 kb/s, and that on the S4 pipe is 224 kb/s.
- When competing with regular TCP flows,

[4] *However, we note that when the latency ratio between component pipes is small, pure congestion-window-based packet scheduling also achieves reasonable performance. The reason is that the progression of the congestion window (which clocks the request for individual pipes) is determined by the bandwidth-delay product of the underlying path, rather than the bandwidth alone. It hence can be used as a lightweight protocol for small latency ratio.*

the binomial congestion control used on the S2 pipe allows it to achieve a relatively smooth send rate. However, a period of long-term unfairness for approximately 150 s can persist due to its slowly-responsive nature to network congestion.

•The aggregate receive rate observed at the receiver manages to stay around the target streaming rate despite the large fluctuations on the *S*3 and *S*4 pipes, and the long-term unfairness on the *S*2 pipe. This is because the fluctuations on individual pipes can potentially be absorbed by other pipes in a multipoint-to-point connection, as evident from the "forced" fluctuations on the *S*0 pipe that are complementary to the fluctuations on the *S*3 and *S*4 pipes; note that the bandwidth of the bottleneck link on the S0 pipe does not exhibit any fluctuations (see the simulation model).

The performance of R$^2$CP in achieving effective video streaming can also be observed from Fig. 6b, where we show the occupancy of the playout buffer. We observe that after the initial ramp-up due to the playout delay, the maximum variation in the playout buffer is only around 100 packets, meaning a relatively stable aggregate transfer rate from multiple sources to the destination despite the fluctuations along individual paths. Finally, we show the size of the resequencing buffer (`recv_buff`) at the R$^2$CP engine. R$^2$CP delivers packets to the application (which are then stored in the application buffer pending playout), and purges packets that are in sequence or determined to be lost by individual pipes (through duplicates or timeouts). The size of the resequencing buffer is calculated upon arrival of new packets. From Fig. 6c we can see the packet scheduling algorithm used by R$^2$CP greatly reduces out-of-order arrivals from multiple pipes, and avoids the requirement for a large resequencing buffer used by the multipoint-to-point connection.

## SUMMARY

In this article we investigate the problem of multipoint-to-point video streaming over peer-to-peer networks. We present a transport layer protocol called R$^2$CP that effectively enables real-time multipoint-to-point video streaming from heterogeneous peers showing large bandwidth and delay mismatches along respective end-to-end paths. R$^2$CP is receiver-driven, requires no coordination between multiple sources, accommodates flexible application layer reliability semantics, and uses TCP-friendly congestion control. Simulation results show that R$^2$CP can effectively achieve multipoint-to-point streaming over peer-to-peer networks.

*R$^2$CP is receiver driven, requires no coordination between the multiple sources, accommodates flexible application layer reliability semantics, and uses TCP-friendly congestion control. Simulation results show that R2CP can effectively achieve multipoint-to-point streaming over peer-to-peer networks.*

## REFERENCES

[1] S. Saroiu, P. Gummadi, and S. Gribble, "A Measurement Study of Peer-to-Peer File Sharing Systems," *Proc. SPIE MMCN*, San Jose, CA, Jan. 2002.

[2] P. Rodriguez and E. Biersack, "Dynamic Parallel-Access to Replicated Content in the Internet," *IEEE/ACM Trans. Net.*, vol. 10,no. 4, Aug. 2002, pp. 455–64.

[3] J. Byers, M. Luby, and M. Mitzenmacher, "Accessing Multiple Mirror Sites in Parallel: Using Tornado Codes to Speed Up Downloads," *Proc. IEEE INFOCOM*, New York, NY, Mar. 1999.

[4] H.-Y. Hsieh and R. Sivakumar, "A Transport Layer Approach for Achieving Aggregate Bandwidths on Multi-Homed Mobile Hosts," *Proc. ACM MOBICOM*, Atlanta, GA, Sept. 2002.

[5] J. Apostolopoulos *et al.*, "On Multiple Description Streaming with Content Delivery Networks," *Proc. IEEE INFOCOM*, New York, NY, June 2002.

[6] T. Nguyen and A. Zakhor, "Distributed Video Streaming over Internet," *Proc. SPIE MMCN*, San Jose, CA, Jan. 2002.

[7] H.-Y. Hsieh *et al.*, "A receiver-centric Transport Protocol for Mobile Hosts with Heterogeneous Wireless Interfaces," *Proc. ACM MOBICOM*, San Diego, CA, Sept. 2003.

[8] S. Sen and J. Wang, "Analyzing Peer-to-Peer Traffic Across Large Networks," *Proc. ACM Internet Measurement Wksp.*, Marseille, France, Nov. 2002.

[9] D. Xu *et al.*, "On peer-to-peer Media Streaming," *Proc. IEEE ICDCS*, Vienna, Austria, July 2002.

[10] M. Hefeeda *et al.*, "PROMISE: Peer-to-Peer Media Streaming Using Collect Cast," *Proc. ACM Multimedia*, Berkeley, CA, Nov. 2003.

[11] D. Bansal and H. Balakrishnan, "Binomial Congestion Control Algorithms," *Proc. IEEE INFOCOM*, Anchorage, AK, Apr. 2001.

[12] D. Clark and D. Tennenhouse, "Architectural Consideration for a New Generation of Protocols," *Proc. ACM SIGCOMM*, Philadelphia, PA, Sept. 1990.

[13] N. Feamster and H. Balakrishnan, "Packet Loss Recovery for Streaming Video," *Proc. Packet Video Wksp.*, Pittsburgh, PA, Apr. 2002.

[14] D. Bovet and M. Cesati, *Understanding the Linux Kernel*, Sebastopol, CA: O'Reilly & Associates, Dec. 2002.

## BIOGRAPHIES

HUNG-YUN HSIEH (hyhsieh@ece.gatech.edu) received B.S. and M.S. degrees in electrical engineering from National Taiwan University, Republic of China. He is currently a Ph.D. candidate in the School of Electrical and Computer Engineering at Georgia Institute of Technology. His research interests include wireless systems, mobile computing, and network protocols. His thesis research focuses on addressing network heterogeneity and bandwidth scarcity in next-generation wireless data networks.

RAGHUPATHY SIVAKUMAR (siva@ece.gatech.edu) received a B.E. degree in computer science from Anna University, India, in 1996, and Master's and doctoral degrees in computer science from the University of Illinois at Urbana-Champaign in 1998 and 2000, respectively. He joined the School of Electrical and Computer Engineering at Georgia Institute of Technology as an assistant professor in August 2000. His research interests are in wireless network protocols, mobile computing, and network quality of service.