Coordinated Multi-streaming for 3D Tele-immersion

David E. Ott and Ketan Mayer-Patel
Department of Computer Science
University of North Carolina at Chapel Hill

ABSTRACT

This paper looks at the problem of multi-streaming in 3D tele-immersion and describes how a protocol architecture called CP (for Coordination Protocol) can be used to coordinate video frame transport between application clusters. CP provides application endpoints with information about current network conditions, and an open architecture for implementing application-specific coordination schemes. The scheme described in this paper apportions available bandwidth among flows such that frame transport synchrony, important for 3D reconstruction performance, is significantly enhanced.

Results demonstrating the effectiveness of CP in increasing multi-stream coordination, while at the same time maintaining aggregate congestion responsiveness, are obtained from a FreeBSD/Linux implementation and a live experimental network. Results underscore the importance of consistency in network information across flows for realizing dramatic improvements in frame arrival synchrony.

Categories and Subject Descriptors: C.2.2 Computer Communication Networks: Network Protocols [applications].
General Terms: Design, Algorithms, Performance, Experimentation.
Keywords: Network protocols, distributed applications, flow coordination.

1. INTRODUCTION

The goal of tele-immersion is to enable users in physically remote spaces to interact with one another in a shared space that mixes both local and remote realities, and allows participants to share a mutual sense of presence. In the 3D Tele-immersion (3DTI) [9, 19] system, for instance, a user wears polarized glasses and a head tracker as a view-dependent scene is rendered in real-time on a large stereoscopic display in 3D. Ideally, there exists a seamless continuum between the user's experience of local and remote space within the application.

Two important components of the 3DTI architecture are *scene acquisition* and *3D reconstruction*. The former is comprised of an array of digital cameras and computing hosts set up to capture a remote physical scene from a wide variety of camera angles. The cameras are calibrated and registered to a *world coordinate system* and are designed to capture images in synchrony. The images are then streamed to a remote location where the 3D reconstruction system uses pixel correspondence and camera calibration information to extract depth values on a per pixel basis. The resulting view-independent *depth streams* can then be rendered from any viewpoint depending on the user's head position. [11]

Figure 1: 3D Tele-immersion.

In this paper, we are interested in the problem of coordinated multi-streaming between scene acquisition and 3D reconstruction components of the 3DTI architecture. Specifically, we are interested in providing reliable transport of *frame ensembles* (a set of n video frames captured from n cameras at the same instant in time) such that aggregate streaming is

- Responsive to network congestion.
- Highly synchronous with respect to frame arrivals.

Congestion responsiveness is important not only to prevent unfairness to competing flows and the possibility of congestion collapse[7], but to minimize unnecessary loss and retransmissions. 3D reconstruction places a high demand on data integrity to be effective, and hence it is a basic requirement in 3DTI that data transport be reliable. Frame synchrony is the notion that frames within the same ensemble are received by the reconstruction subsystem at the same time. A low degree of frame synchrony will result in *stalling* as the 3D reconstruction pipeline waits for remaining pixel data to arrive, a highly undesirable effect for 3DTI as a real-time, interactive application.

The problem of coordinated multi-streaming is significant because it represents an instance of a much wider class of distributed multimedia applications that we refer to as *cluster-to-cluster applications*. Unlike end-to-end applications where the endpoints of communication are single hosts, a cluster-to-cluster (C-to-C) application uses collections of computing and communication devices. The many flows typically employed in a C-to-C application transfer data between endpoints on one cluster and endpoints on another. To be effective, a C-to-C application must carefully coordinate the use of network resources across these flows to properly reflect application-level goals and priorities.

Figure 2 illustrates the general architecture of a C-to-C application. In 3DTI, for example, one cluster is comprised of cameras connected to computers capturing video and the other cluster is comprised of the distributed reconstruction system that renders a 3D view of the captured environment. Some identifying characteristics of this architecture include:

- A common intermediary path called the *cluster-to-cluster*

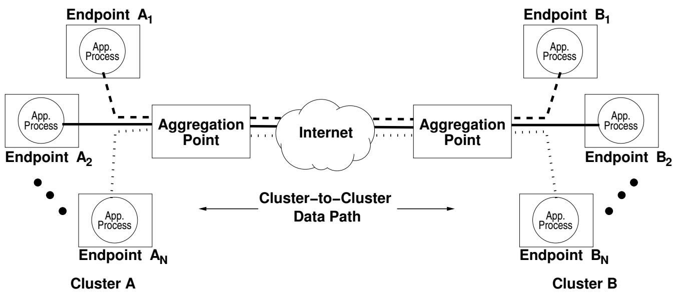

Figure 2: C-to-C application model.

data path. This is the path over the wide-area that flows passing between clusters will share.

- A natural *aggregation point* through which all data leaving one cluster will converge to the same forwarding host. We refer to this aggregation point as an *AP*. Typically the AP is the first-hop gateway router connecting a cluster to the rest of the Internet. In cluster-to-cluster applications, two APs exist, one in front of each cluster on either side of the C-to-C data path.
- Independent, but semantically related flows of data that are part of the same C-to-C application.
- Complex adaptation requirements as the priority and bandwidth requirements of flows within the application change dynamically.

In general, the network *within* a cluster is under local administrative control and can be provisioned to comfortably support the needs of the application. In contrast, the path *between* clusters is shared with other Internet flows and typically cannot be provisioned. As such, it is the primary source of changing network conditions including congestion.

As we consider ever more complex multimedia applications such as tele-immersion that exhibit a cluster-to-cluster architecture, the lack of mechanisms to manage aggregate bundles of related flows becomes increasingly problematic. Network-level protocols like IP are concerned only with the next hop while transport-level protocols like TCP are concerned with end-to-end semantics such as reliability. A void exists between these two levels for addressing the concerns of related flows that share only a portion of their end-to-end paths.

The problem to be solved in cluster-to-cluster applications is that of *flow coordination*. Application flows share a common intermediary path between clusters, and yet employ transport protocols that *operate in isolation from one another*. As a result, related flows of the same application may compete with one another when network resources become limited instead of cooperating to use available bandwidth in application-controlled ways.

Our approach to the problem of coordinated multi-streaming in 3DTI is one that highlights a generalized architecture for solving flow coordination problems in C-to-C applications of all kinds. We call this architecture the *Coordination Protocol* or *CP*. CP provides application endpoints with information on current conditions on the C-to-C data path, including the aggregate bandwidth that is available to the application *as a whole*. At the same time, it serves as a repository of state information that can be exchanged between flows in an application-defined manner. The result is an open architecture that can be exploited by any C-to-C application to realize flow coordination and aggregate congestion responsiveness in ways specific to the application's problem domain.

In this paper, we show how CP can be applied to the problem of multi-streaming in 3DTI. The scheme described dynamically apportions available bandwidth among flows such that frame transport synchrony, important for 3D reconstruction performance, is significantly enhanced while preserving end-to-end reliability semantics. Results demonstrating the effectiveness of CP in increasing multi-stream coordination and at the same time maintaining aggregate congestion responsiveness are obtained from a FreeBSD/Linux implementation and a live experimental network.

The organization of this paper is as follows. In Section 2, we summarize the Coordination Protocol architecture. Section 3 discusses the problem of multi-streaming in 3DTI and how CP can be applied to increase frame arrival synchronization. We present results in Section 4 showing that CP significantly enhances application performance. In Section 5, we mention related work. Finally, Section 6 summarizes the paper.

2. COORDINATION PROTOCOL

In this section, we describe the Coordination Protocol (CP). Our focus here will be on CP services rather than internal mechanisms and implementation detail. For more information on aggregate congestion control mechanisms used by CP, the reader is referred to [14].

2.1 Overview

The CP protocol architecture is designed with two goals in mind:

- To inform endpoints of network conditions over the cluster-to-cluster data path, including aggregate bandwidth available to the application as a whole, and
- To provide a lightweight infrastructure for exchanging state among flows and allowing an application to implement its own flow coordination scheme.

To realize these goals, CP makes use of a shim header inserted in C-to-C application packets. Ideally, the position of this header would be between the network layer (IP) and transport layer (TCP, UDP, etc.). This makes it transparent to IP routers on the C-to-C forwarding path and preserves end-to-end transport-level protocol semantics. Our UDP-based implementation, however, places the CP header in the first several bytes of UDP application data, thus obviating the need for endpoint OS changes and making the protocol more deployable. In either case, each flow of a C-to-C application employs a transport-level protocol that makes use of CP information in various ways.

The main CP mechanism is actually implemented at the aggregation point (AP). This may be the cluster's first hop router, or a forwarding agent in front of the first hop router through which all application traffic must pass. The AP is part of the cluster's local computing environment and, as such, is under local administrative control.

The CP header of packets belonging to the same C-to-C application are processed by the AP during packet forwarding. Essentially, the AP uses information in the CP header to maintain a state table. Flows deposit information into the state table of their local AP as packets traverse from an application endpoint through the AP and on toward the remote cluster. Packets traveling in the reverse direction pick up entries from the state table and report them to the transport-level protocol layered above CP and/or the application.

In addition, the two APs conspire to measure characteristics of the C-to-C data path such as round trip time, loss, available bandwidth, etc. These measurements are made using all packets of all flows belonging to the same C-to-C application. These values are also inserted into the state table. Figure 3 illustrates the header and its contents at different points on the network path.

Report information is received by an application endpoint on a per packet basis. This information can take several forms including: information on current network conditions on the C-to-C data path (round trip time, loss, available bandwidth), information on peer flows (number of flows, aggregate bandwidth usage), and/or application-specific information exchanged among flows using a format and semantics defined by the application. An endpoint uses a subset of available

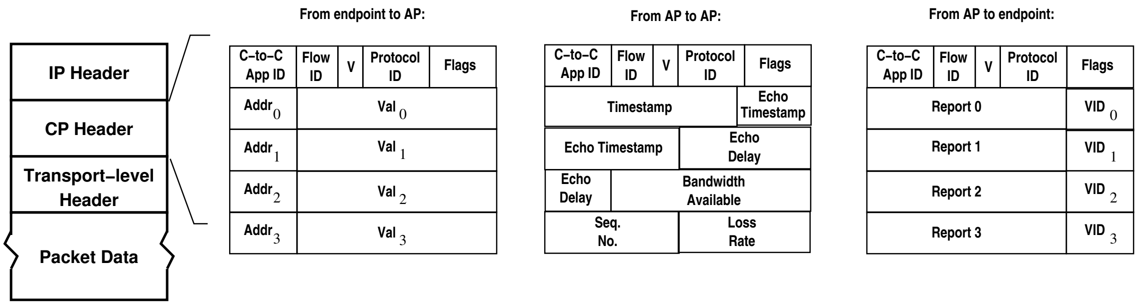

Figure 3: CP packet structure as it traverses the cluster-to-cluster data path.

information to make send rate and other adjustments (e.g., encoding strategy) to meet application-defined goals for network resource allocation and other coordination tasks.

While our overview of CP here is brief, an important point should be emphasized. CP's role is to provide information useful to application endpoints in implementing their own self-designed coordination schemes. In a sense, it is merely an information service piggybacked on packets that traverse the cluster-to-cluster data path. As such, aggregation points do no buffering, scheduling, shaping, or policing of application flows. Instead, coordination is implemented *by the application* which must configure endpoints to respond to CP information with appropriate send rate adjustments that reflect the higher objectives of the application.

2.2 AP State Tables

An AP creates a *state table* for each C-to-C application currently in service that acts as a repository for network and flow information, as well as application-specific information shared between flows in the C-to-C application.

The organization of a state table is as follows:

- The table is a two dimensional grid of cells each of which can be addressed by an *address* and an *offset*. (We will use the notation $address.offset$ when referring to particular cells.)
- There are 256 addresses divided into four types: *report pointers*, *network statistics*, *flow statistics*, and *general purpose* addresses.
- For each address, 256 *offsets* are defined. The value and semantics of the particular cell located by the offset depend on the address context.

Each cell in the table contains a 24-bit value. Our current implementation uses four bytes per cell to align memory access with word boundaries, making the state table a total of 256 KB in size. Even with a number of concurrent C-to-C applications, tables can easily fit into AP memory.

An endpoint may read any location (*address.offset*) in the table by using the report address mechanism described below. In contrast, an endpoint may only write specific offsets of the report and application addresses; network and flow statistic addresses are assigned by the AP and are read-only. The state table is illustrated in Figure 4.

2.2.1 Setting Cells of the State Table

The CP header of outgoing packets can be used to set the value of up to 4 cells in the state table. When an outbound packet (i.e., a packet leaving a cluster on its way toward the other cluster) arrives at the AP, the CP header includes the following information:

- The flow id (fid) of the specific flow to which this packet belongs. Each flow of the application is assigned a unique fid in the range $[0, 63]$. How this flow id assignment is made is an orthogonal issue which is not important to our discussion and can be achieved in a number of different ways.
- Four "operation" fields which are used to set the value of specific cells in the state table. The operation field is comprised of two parts. The first is an 8-bit address ($Addr_i$) and the second is a 24-bit value (Val_i). The i subscript is in the range $[0, 3]$ and simply corresponds to the index of the 4 operation fields in the header. Figure 3 illustrates this structure.

When an AP receives an outbound packet, each operation field is interpreted in the following way. The cell to be assigned is uniquely identified by $Addr_i.fid$. The value of that cell is assigned Val_i. In this manner, each flow is uniquely able to assign one of the first 64 cells associated with that address.

Although the address specified in the operation field is in the range $[0, 255]$, not all of these addresses are valid for writing (i.e., some of the addresses are read-only). Similarly, since a flow id is restricted to the range $[0, 63]$, in fact only 64 of the offsets associated with a particular writable address can be set. As previously mentioned, the address space is divided into four address types. The mapping between address range and type is illustrated in Figure 4. The semantics of a cell value at a particular offset depends on the specific address type and is described in the following subsections.

2.2.2 Report Pointers

Four of the writable addresses are known as *report pointers*. Using the mechanism described above, each flow is able to write a 24-bit value into $R_j.fid$ where R_j is one of the 4 report pointers (i.e., $R1$, $R2$, $R3$, and $R4$ in Figure 4) and fid is the flow id. The value of these 4 cells (per flow) control how the AP processes *inbound* packets (i.e., packets arriving from the remote cluster) of a particular flow.

When an inbound packet arrives, the AP looks up the value of $R_j.fid$ for each of the four report addresses. The 24-bit value of the cell is interpreted in the following way. The first 8 bits are interpreted as a state table address ($addr$). The second 8 bits are interpreted as an offset (off) for that address. The final 8 bits are interpreted as a validation token (vid). The AP then copies into the CP header the 24-bit value located at ($addr.off$) concatenated with the 8-bit validation token vid. This is done for each of the four report fields.

Thus, outbound packets of a flow are used to write a value into each of four report pointers, $R1$ through $R4$. These configure the AP to report values in the state table using inbound packets. The validation token has no meaning to the AP *per se*, but can be used by the application to help disambiguate between different reports.

2.2.3 Network Statistics

One of the addresses in the table is known as the network statistics address (NET). This is a read-only address. The offsets of this address correspond to different network statistics about the C-to-C data path as measured by APs across the aggregate of all flows in the C-to-C application including:

- Round trip time ($NET.rtt$)
- Loss ($NET.loss$)
- Bandwidth available ($NET.bw$)

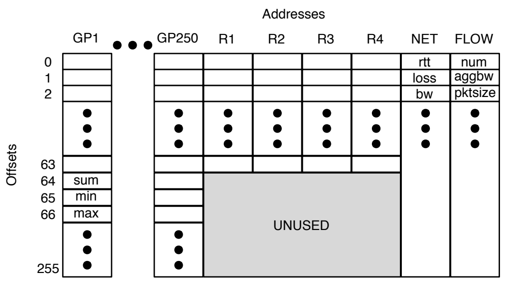

Figure 4: CP state table maintained at each AP.

This list is far from complete, merely giving the statistics that come into play when we describe how we use CP mechanisms to solve the synchronous frame ensemble transfer problem. Bandwidth available provides an estimate of the bandwidth available to a single TCP-compatible flow given the current round trip time, packet loss rate, average packet size, etc. How this estimate is calculated, and how the value can be scaled to n application flows is described in [14].

2.2.4 Flow Statistics

While statistics characterizing the C-to-C data path are available through the NET address, statistics characterizing the application flows themselves are provided by offsets of the flow statistics address $FLOW$. Offsets of this address include information about:

- Number of active flows ($FLOW.num$)
- Throughput ($FLOW.tput$)
- Average packet size ($FLOW.pktsize$)

Again, this is merely a partial list enumerating the offsets of this address that we currently use within the 3DTI system. Other offsets correspond to other types of statistics. Up to 256 different statistics can be provided.

2.2.5 General Purpose Addresses

The general purpose addresses (i.e., $GP1$ through $GP250$) in Figure 4 give a cluster-to-cluster application a set of tools for sharing information in a way that facilitates coordination among flows. For example, general purpose addresses may be used to implement floor control, dynamic priorities, consensus algorithms, dynamic bandwidth allocation, etc. General purpose addresses may also be useful in implementing coordination tasks among endpoints not directly related to networking.

Offsets for each general purpose address are divided into two groups: assignable *flow offsets* and read-only *aggregate function offsets*. We have already discussed how the offsets equal to each flow id can be written by outbound packets of the corresponding flow. These are the flow offsets. While this accounts for the first 64 offsets of each of general purpose address, the remaining 196 offsets are used to report aggregate functions of these first 64 flows offsets. Some examples are:

- Statistical offsets for functions such as sum, min, max, range, mean, and standard deviation.
- Logical offsets for functions such as AND, OR, and XOR.
- Pointer offsets. For example, the offset of the minimum value, the offset of the maximum value, etc.
- Usage offsets. For example, the number of assigned flow offsets or the most recently assigned offset.

Operations are implemented using lazy evaluation for efficiency. Which operations to include is an area of active research. Flow offsets are treated as soft state and time out if not refreshed.

2.3 Implementing Flow Coordination

While CP provides network and flow information, and provides facilities for exchanging information, it is up to the cluster-to-cluster application to exploit these services to achieve coordination among flows. Typically this involves a distributed adaptation scheme in which current network conditions and application state dictate the send rate modifications on individual hosts.

An important point to note here is that aggregate bandwidth available to the application as a whole (equal to CP's bandwidth available estimate for a single flow times the number of active flows in the application) may be distributed across endpoints in any manner. That is, it is *not* necessarily the case that a given application flow receives exactly $1/n$ of the aggregate bandwidth in an n-flow application. In fact, an application may apportion bandwidth across endpoints in any manner as long as the aggregate bandwidth level is not exceeded. We believe this to be a powerful feature of our protocol architecture.

Here we provide a couple brief examples to illustrate. Examples are "miniature" in the sense that realistic C-to-C applications are likely to have many more flows and networking requirements that are more complex and change dynamically.

Example 1. Flows A, B, and C are always part of the same cluster-to-cluster application, but flows D and E join and leave intermittently. Each requests $NET.bw$ reports to inform them of the estimated bandwidth available to a single application flow. In addition, they request $FLOW.num$ reports that tell them how many flows are currently part of the application. Since the application is configured to run at no more than 3 Mbps, each flow sends at the rate $R = min(3Mbps/FLOW.num, Net.bw)$.

Example 2. Flow A is a control flow. Flows B and C are data flows. All flows request $NET.bw$ and $GP1.fid(A)$ which informs them of the value flow A has assigned to general purpose address 1 at the offset equal to its flow id. When running, the application has two states defined by the value flow A has assigned to $GP1.fid(A)$: NORMAL ($GP1.fid(A) = 0$) which indicates normal running mode, and UPDATE ($GP1.fid(A) = 1$) which indicates that a large amount of control information is being exchanged to update the state of the application. During NORMAL, A sends at the rate $R = (3 * NET.bw) * .1$ while B and C each send at no more than $R = (3 * NET.bw) * .45$. During UPDATE, A sends at the rate $R = (3 * NET.bw) * .9$ while B and C each send at no more than $R = (3 * NET.bw) * .05$.

These simple examples help to illustrate some of the advantages of the CP state table mechanism. Flows can be tightly coordinated and exchange state information on a per packet basis. Distributed local decisions can be made in informed ways that result in the appropriate global behavior using AP state table information piggybacked on packets that are already being sent and received as part of the application. Aggregate measures of application performance that can be effectively gathered only at the AP and not at any one end host are made available to the application. AP performance is not a bottleneck because the amount of work done for each forwarded packet is limited to simple accounting and on-demand state table updates.

3. COORDINATED MULTI-STREAMING

The design and implementation of CP was inspired by the multi-streaming requirements of applications like tele-immersion. As such, we have been collaborating 3D Tele-immersion (3DTI) developers to showcase how CP can help with some of their most difficult data transport challenges. The rest of this paper explores ways in which CP was employed within this context and the resulting improvements in application performance. The central problem faced by the developers of 3DTI was that of asynchrony among multiple video flows.

The scene acquisition subsystem in 3DTI is charged with capturing video frames simultaneously on multiple cameras and streaming them to the 3D reconstruction engine at a remote location. The problem of synchronized frame capture is solved using a single *triggering* mechanism across all cameras. Triggering can be handled periodically or in a synchronous blocking manner in which subsequent frames are triggered only when current frames have been consumed. The triggering mechanism itself can be hardware-based (a shared 1394 Firewire bus) or network-based using message passing.

3DTI uses synchronous blocking and message passing to trigger simultaneous frame capture across all hosts. A *master-slave* configuration is used in which each camera is attached to a separate Linux host (i.e., slave) that waits for a triggering message to be broadcast by a trigger host (i.e., master). Once a message has been received, a frame is captured and written to the socket layer which handles reliable streaming to an endpoint on the remote reconstruction subsystem. As soon as the write call returns (i.e., the frame can be accommodated in the socket-layer send buffer), a message is sent to the trigger host notifying it that the capture host is ready to capture again. When a message has been received for all hosts, the trigger host broadcasts a new trigger message and the process repeats.

Some key issues in multi-streaming video frames in this context include

- Send buffer size,
- choice of transport protocol,
- aggregate responsiveness to network congestion,
- bandwidth utilization, and
- synchronization across flows.

In the original 3DTI design, TCP was chosen to be the transport-level protocol for each video stream. TCP, while not typically known as a streaming protocol, was an attractive choice to the 3DTI developers for several reasons. First, it provided in-order, reliable data delivery semantics which, as mentioned in Section 1, is an important requirement in this problem domain. Second, it is congestion responsive. Use of TCP for multi-streaming in 3DTI insures that C-to-C traffic as an aggregate is congestion responsive by virtue of the fact that individual flows are congestion responsive. The original developers had hoped that by using relatively large capacity networks (i.e., Abilene), performance would not be an issue.

The resulting application performance, however, was poor, but not necessarily because of bandwidth constraints. Instead, the uncoordinated operation of multiple TCP flows between the acquisition and reconstruction clusters resulted in large end-to-end latencies and asynchronous delivery of frames by different flows. By adding CP mechanisms to the architecture and developing a CP-based, reliable transport-level protocol, we demonstrate how a small bit of coordination between peer flows of a C-to-C application can go a long way toward achieving application-wide networking goals.

3.0.1 Multi-streaming with TCP

The major disadvantage of TCP in the multi-streaming problem context, is that individual flows operate independently of peer flows within the same application. Each TCP stream independently detects congestion and responds to loss events using its well-know algorithms for increasing and decreasing congestion window size. While the result is a congestion responsive aggregate, differences in congestion detection can easily result in a high degree of asynchrony as some flows detect multiple congestion events and respond accordingly, while other flows encounter fewer or no congestion events and maintain a congestion window that is, on average during the streaming interval, larger. The result, for equal size frames across all capture endpoints, is that some flows may end up streaming frames belonging to the same ensemble more quickly *at the expense* of peer flows that gave up bandwidth in the process.

The problem is heightened when video frames are of unequal size. This might be the case when individual capture hosts apply compression as part of the capture process. A flow with more data to send might, in some cases, encounter more congestion events and, as a result, back off more than a flow with less data to send. The result is a high probability of stalling as some flows finish streaming their frame and wait for the remaining flows to complete before the next frame trigger can proceed.

The problem of stalling can be mitigated, of course, by increasing socket-level send buffering, but at the expense of increasing end-to-end delay which is highly undesirable since 3DTI is an interactive, real-time application. What is needed, we argue, is an *appropriate amount of buffering*: large enough to maintain a full data pipeline at all times, but small enough to minimize unnecessary end-to-end delay. Maintaining this balance requires information about conditions on the C-to-C data path, however, something that TCP cannot provide.

3.0.2 Multi-streaming with CP-RUDP

To address these problems, we deployed CP-enabled software routers in front of each cluster to act as APs. Then we developed a new UDP-based protocol called CP-RUDP and deployed it on each endpoint host in the application. CP-RUDP is an application-level transport protocol for experimenting with send rate modifications using CP information in the context of multi-stream coordination. Essentially, it provides the same in-order, reliable delivery semantics as TCP, but with the twist that reliability has been completely decoupled from congestion control. This is because the CP layer beneath can now provide the congestion control information needed for adjusting send rate in appropriate ways. In addition, CP-RUDP is a rate-based protocol, while TCP is a window-based protocol.

In the context of 3DTI, our work focused on two areas:

- Better bandwidth distribution for increased frame arrival synchrony.
- Adjusting sender-side buffering to maximize utilization but minimize delay.

To accomplish the first goal, we rely on an important property of the CP state table: *consistency of information across endpoints*. Because the APs are now in charge of measuring network characteristics of the C-to-C data path for the application as a whole, individual flows can employ that information and make rate adjustments confident that peer flows of the same application are getting the same information and are also appropriately responding. In particular, endpoints see the same bandwidth availability estimates, round trip time measurements, loss rate statistics, and other network-based statistics.

With this in mind, we found the most effective coordination algorithm for 3DTI's multi-streaming problem to be the application of two relatively simple strategies.

- Each endpoint in the application sends at *exactly* the rate given by the $Net.bw$ report value. The value, as described in [14], incorporates loss and round trip time measurement information on the cluster-to-cluster data path and uses a TCP modeling equation to calculate the instantaneous congestion-responsive sending rate for a single flow [6, 8].
- Each endpoint uses an adaptive send buffer scheme in which buffer size, B, is continually updated using the expression $B = 1.5 * (Net.bw * Net.rtt)$. In other words, the send buffer size is set to constantly be 1.5 times the bandwidth delay product. By using the bandwidth delay product, we insure that good network utilization is effectively maintained at all times. The 1.5 multiplicative factor is simply a heuristic that insures some additional buffer space for retransmission data and data that is waiting to be acknowledged.

Experimental results demonstrating the effectiveness of this scheme are presented in the following section.

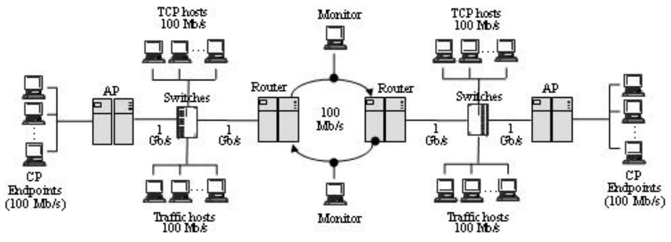

Figure 5: Experimental network setup.

4. RESULTS

In this section, we present experimental results demonstrating the effectiveness of flow coordination to the problem of multi-streaming in 3DTI. Included is a description of our experimental setup and performance metrics. Our goal is to compare multi-streaming performance between TCP, a reliable, congestion responsive but uncoordinated transport protocol, and CP-RUDP, an equivalently reliable, congestion responsive transport protocol but with the added feature that it supports flow coordination. Our results show a dramatic improvement in synchronization while maintaining a bandwidth utilization that does not exceed that of TCP. They also underscore the tremendous benefit of information consistency across flows as provided by the CP architecture.

4.1 Experimental Setup

Our experimental network setup is shown in Figure 5. CP hosts and their local AP on each side of the network represent two clusters that are part of the same C-to-C application and exchange data with one another. Each endpoint sends and receives data on a 100 Mb/s link to its local AP, a FreeBSD router that has been CP-enabled as described above. Aggreate C-to-C traffic leaves the AP on a 1 Gb/s uplink. At the center of our testbed are two routers connected using two 100 Mb/s Fast Ethernet links. This creates a bottleneck link, and by configuring traffic from opposite directions to use separate links, emulates the full-duplex behavior seen on wide-area network links.

In order to calibrate the fairness of application flows to TCP flows sharing the same bottleneck link, we use two sets of hosts (labeled "TCP hosts" in Figure 5) and the well-known utility *iperf* [1]. Iperf flows are long-lived TCP flows that compete with application flows on the same bottleneck throughout our experiment. The normalized flowshare metric described in Section 4.2 then provides a way of quantifying the results.

Also sharing the bottleneck link for many experiments are background flows between traffic hosts on each end of the network. These hosts are used to generate Web traffic at various load levels and their associated patterns of bursty packet loss. More is said about these flows in Section 4.4.

Finally, network monitoring during experiments is done in two ways. First, *tcpdump* is used to capture TCP/IP headers from packets traversing the bottleneck, and then later filtered and processed for detailed performance data. Second, a software tool is used in conjunction with *ALTQ* [10] extensions to FreeBSD to monitor queue size, packet forwarding events, and packet drop events on the outbound interface of the bottleneck routers. The resulting log information provides packet loss rates with great accuracy.

4.2 Performance Metrics

Define *frame ensemble* to be a set of n frames captured by n different frame acquisition hosts at the same instant in time. A frame ensemble is generated after each triggering event as described in Section 3.

Overall, our goal is to compare the level of synchrony in frame arrivals within the same frame ensemble. With this in mind, we define the metric *completion asynchrony* for frame ensemble i as follows. Within any given frame ensemble i, there is some receiving host that receives frame i in its entirety first. Let's call this host $H_{f,i}$ and the time of completion $c_{f,i}$. There's another host that receives frame i in its entirety last (i.e., after all other hosts have already received frame i). Call this host $H_{l,i}$ and the time of completion $c_{l,i}$. Completion asynchrony C_i is defined as the time interval between frame completion events $c_{l,i}$ and $c_{f,i}$. Intuitively, it reflects how staggered frame transfers are across all application flows in receiver-based terms.

$$C_i = c_{l,i} - c_{f,i} \tag{1}$$

An important metric to the application as a whole is the *frame ensemble transfer rate* which we define as the number of complete frame ensemble arrivals f over time interval p.

$$R = \frac{f}{p} \tag{2}$$

Finally, we wish compare the bandwidth taken by flows in the application to that of TCP iperf flows competing over the same bottleneck link. Define *average flowshare* (F) to be the mean aggregate throughput divided by the number of flows. The *normalized flowshare* is then the average flowshare among a subset of flows, for example CP-RUDP flows ($F_{CP-RUDP}$), divided by the average flowshare for all flows (F_{all}). (All flows here refers to CP-RUDP flows and competing TCP iperf flows, but not background traffic flows.)

$$F_{CP-RUDP} = \frac{F_{CP-RUDP}}{F_{all}} \tag{3}$$

1.0 represents an ideal fair share. A value greater than 1.0 indicates that CP-RUDP flows on an average have received more than their fair share, while for less than 1.0 the reverse is true.

4.3 Loss Experiments

In this section we compare the performance of TCP multi-streaming with that of CP-RUDP under conditions of varying packet loss rates. In each experiment, TCP's send buffer is necessarily configured to be large in order to maximize utilization in the face of frame asynchrony and insure that good frame ensemble rates are maintained.

To generate controlled loss, we used the *dummynet* [16] traffic shaping utility found in FreeBSD 4.5. *Dummynet* provides support for classifying packets and dividing them into flows. A pipe abstraction is then applied that emulates link characteristics including bandwidth, propagation delay, queue size, and packet loss rate. We enabled dummynet on bottleneck routers and configured it to produce loss at various rates.

Runs lasted for 10 minutes during which the initial 5 minutes were spent on ramp-up and stabilization, and the subsequent 5 minutes were used to collect performance data. Trials with longer stabilization and run intervals did not show significantly different results.

Completion asynchrony results in Figure 7 show a dramatic difference between TCP and CP-RUDP. Values for TCP show frame arrival asynchrony to be 5 to 20 times larger than CP-RUDP.

Figure 7 shows CP-RUDP getting a slightly better frame ensemble transfer rate for various loss rates, but not by much. Likewise, Figure 8 shows CP-RUDP getting slightly more bandwidth, but not by much.

In summary, the benefit of CP-RUDP to 3DTI multi-streaming is a substantial increase in synchronization while maintaining appropriate utilization and fairness to TCP peer flows.

4.4 Load Experiments

While testing CP performance under various dummynet loss conditions is instructive, a random loss model is wholly unrealistic. In reality, losses induced by drop tail queues in Internet routers are bursty and correlated. To better capture

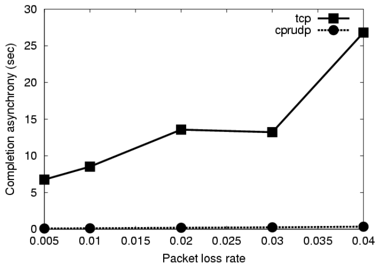

Figure 6: Completion asynchrony vs. packet loss.

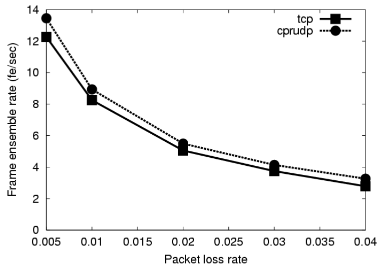

Figure 7: Frame ensemble transfer rate vs. packet loss.

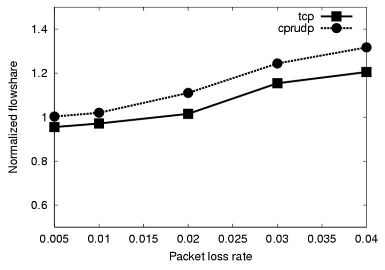

Figure 8: Normalized flowshare vs. packet loss.

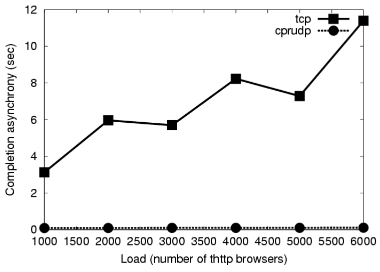

Figure 9: Completion asynchrony vs. *thttp* load.

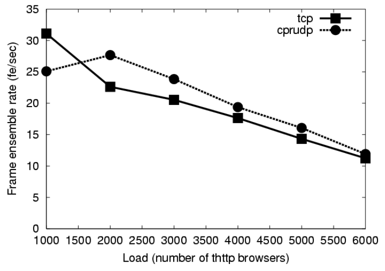

Figure 10: Frame ensemble transfer rate vs. *thttp* load.

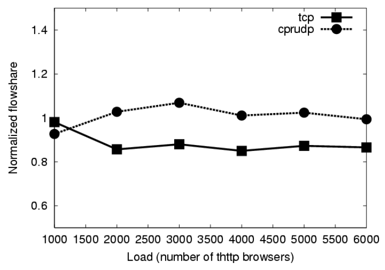

Figure 11: Normalized flowshare vs. *thttp* load.

this dynamic, we tested TCP and CP-RUDP performance against various background traffic workloads using a Web traffic generator known as *thttp*.

Thttp uses empirical distributions from [17] to emulate the behavior of Web browsers and the traffic that browsers and servers generate on the Internet. Distributions are sampled to determine the number and size of HTTP requests for a given page, the size of a response, the amount of "think time" before a new page is requested, etc. A single instance of *thttp* may be configured to emulate hundreds of Web browsers and significant levels of TCP traffic with real-world characteristics. Among these characteristics are heavy-tailed distributions in flow ON and OFF times, and significant long range dependence in packet arrival processes at network routers.

We ran four *thttp* servers and four clients on each set of traffic hosts seen in Figure 5. Emulated Web traffic was given a 20 minute ramp-up interval and competed with TCP and CP-RUDP flows on the bottleneck link in both directions. We varied the number of browsers emulated from 1000 to 6000. Resulting loss rates are between .005 and .05 as measured at bottleneck router queues.

Figure 9 completion asynchrony results once again illustrate the vast improvement CP-RUDP represents over TCP in terms of frame arrival synchrony.

Figure 10 shows that frame ensemble rates are once again very similar between CP-RUDP and TCP, indicating similar throughput for both. Normalized flowshare results in Figure 11 show CP-RUDP achieving almost perfect fairness with competing TCP flows, while TCP application flows receive a little less than their fair share due to stalling.

4.5 Fixed Frame Size Experiments

In this set of experiments, we chose a background *thttp* load of 6000 browsers and looked at the effect of frame size on transfer asynchrony.

Figure 12 completion asynchrony results show CP-RUDP to be a vast improvement over TCP in terms of frame arrival synchrony. Furthermore, CP-RUDP asynchrony values remain consistently low across all frame sizes.

Frame ensemble rates in Figure 10 and normalized flowshare results in Figure 11 repeat the pattern discussed in the previous sections.

4.6 Variable Frame Size Experiments

Finally, we look at the effect of variable frame size on transfer asynchrony. Here, variable frame size refers to differences in frame size within the same frame ensemble. This situation might occur, for example, when compression is applied by individual endpoints on frames that differ somewhat in image content.

To generate variable sized frames, we have each sender sample a normal distribution to determine frame size at each capture event. The mean frame size of the run and a standard deviation value of 25% are used as input parameters to the distribution.

Figure 9 completion asynchrony results once again show a dramatic and consistent improvement over TCP. Frame ensemble rates in Figure 10 and normalized flowshare results in Figure 11 once again repeat the pattern discussed in previous plots.

5. RELATED WORK

The initial concept of the Coordination Protocol (CP) was sketched in [12], and early simulation results presented in [13]. [14] presents work on CP's aggregate congestion control mechanisms, including *bandwidth filtered loss detection.* The work presented in this paper, in contrast to previous work, looks at applying CP to the problem of multi-streaming in 3D tele-immersion, emphasizing CP mechanisms for state sharing and experimental results from a real implementation.

The ideas behind CP were primarily inspired by the Congestion Manager (CM) architecture developed by Balakrishnan [3]. CM provides a framework for different transport-level protocols on the same host to share information on network conditions. CM is an excellent example of a coordination mechanism, but operates only when flows share the entire end-to-end path.

Pradhan et al. propose a way of aggregating TCP connections sharing the same traversal path in order to share congestion control information [15]. Their scheme takes a TCP connection and divides it into two separate ("implicit") TCP connections: a "local subconnection" and a "remote subconnection." This scheme, however, breaks the end-to-end semantics of the transport protocol (i.e., TCP).

Active networking, first proposed by [18] allows custom programs to be installed within the network. Since their original

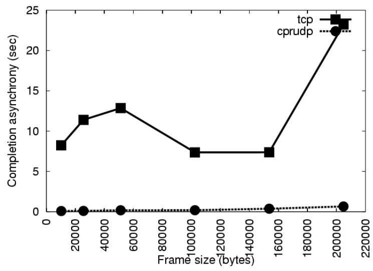

Figure 12: Completion asynchrony vs. frame size.

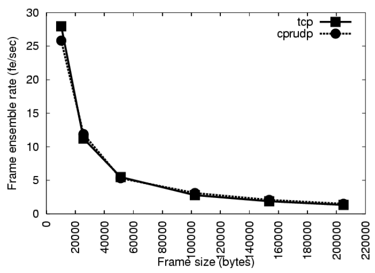

Figure 13: Frame ensemble transfer rate vs. frame size.

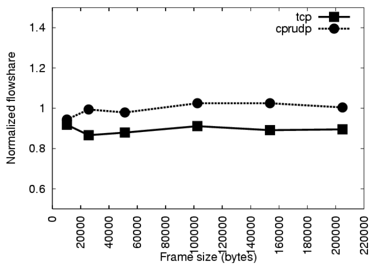

Figure 14: Normalized flowshare vs. frame size.

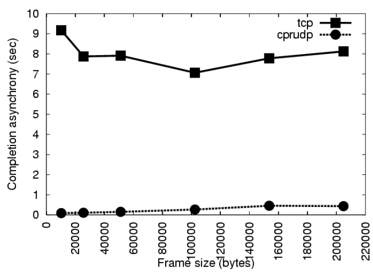

Figure 15: Completion asynchrony vs. mean frame size.

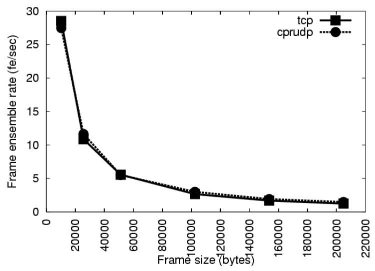

Figure 16: Frame ensemble transfer rate vs. mean frame size.

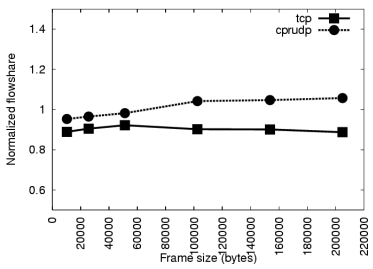

Figure 17: Normalized flowshare vs. mean frame size.

conception, a variety of active networking systems have been built ([20, 2, 5], for instance). They are often thought of as a way to implement new and customized network services. In fact, CP could be implemented within an active networking framework. Active networking, however, mandates changes to routers along the entire network path. This severely hinders deployment. CP requires changes only at the endpoints and at the aggregation points.

[4] describes a lightweight scheme that allows IP packets to manipulate small amounts of temporary state at routers using an ephemeral state store (ESS). The CP state table extends these ideas by including measures of network conditions, flow information, and aggregate functions of application-deposited state.

6. SUMMARY

In this paper, we motivate the need for coordination among peer flows of a broad class of futuristic multimedia applications that we call cluster-to-cluster (C-to-C) applications. These applications involve multi-stream communication between clusters of computing resources across which the application is distributed. One such application at the focus of our attention is a tele-immersion system called 3DTI.

To address the multi-stream coordination issues of C-to-C applications, we have developed a protocol architecture called the Coordination Protocol (CP) that measures network conditions between clusters across all application flows and provides them with a generalized state sharing mechanism. The result is an open architecture for implementing flow coordination in application-defined ways.

In particular, we have used CP to develop a new reliable streaming protocol called CP-RUDP that addresses the synchronization requirements found in 3DTI. We presented results showing how CP-RUDP is able to dramatically improve multistream synchronization within the context of this application.

7. REFERENCES

[1] http://dast.nlanr.net/Projects/Iperf/.

[2] D.S. Alexander et al. Active bridging. *Proceedings of SIGCOMM'97*, pages 101–111, September 1997.

[3] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An integrated congestion management architecture for internet hosts. *Proceedings of ACM SIGCOMM*, September 1999.

[4] Kenneth L. Calvert, James Griffioen, and Su Wen. Lightweight network support for scalable end-to-end services. In *Proceedings of ACM SIGCOMM*, August 2002.

[5] D. Decasper et al. Router plugins: A software architecture for next generation routers. *Proceedings of SIGCOMM'98*, pages 229–240, September 1998.

[6] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. *Proceedings of ACM SIGCOMM*, pages 43–56, 2000.

[7] Sally Floyd and Kevin R. Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, 1999.

[8] M. Handley, S. Floyd, J. Padhye, and J. Widmer. *RFC 3448: TCP Friendly Rate Control (TFRC): Protocol Specification.* Internet Engineering Task Force, January 2003.

[9] N. Kelshikar, X. Zabulis, J. Mulligan, K. Daniilidis, V. Sawant, S. Sinha, T. Sparks, S. Larsen, H. Towles, K. Mayer-Patel, H. Fuchs, J. Urbanic, K. Benninger, R. Reddy, and G. Huntoon. Real-time terascale impementation of tele-immersion. *International Conference on Computation Science*, June 2003.

[10] C. Kenjiro. A framework for alternate queueing: Towards traffic management by PC-UNIX based routers. In *USENIX 1998*, pages 247–258, June 1998.

[11] Sang-Uok Kum, Ketan Mayer-Patel, and Henry Fuchs. Real-time compression for dynamic 3D environments. *ACM Multimedia 2003*, November 2003.

[12] D. Ott and K. Mayer-Patel. Transport-level protocol coordination in cluster-to-cluster applications. *Proceedings of the Interactive Distributed Multimedia Systems Workshop (IDMS)*, 2001.

[13] D. Ott and K. Mayer-Patel. A mechanism for TCP-friendly transport-level protocol coordination. *USENIX 2002*, June 2002.

[14] D. Ott, T. Sparks, and K. Mayer-Patel. Aggregate congestion control for distributed multimedia applications. *Proceedings of IEEE INFOCOM '04*, March 2004.

[15] P. Pradhan, T. Chiueh, and A. Neogi. Aggregate TCP congestion control using multiple network probing. *Proc. of IEEE ICDCS 2000*, 2000.

[16] Luigi Rizzo. http://info.iet.unipi.it/ luigi/ip_dummynet/.

[17] F.D. Smith, F. Hernandez Campos, K. Jeffay, and D. Ott. What TCP/IP protocol headers can tell us about the web. In *ACM SIGMETRICS*, pages 245–256, June 2001.

[18] D. L. Tennenhouse and D. Wetherall. Towards an active network architecture. *Multimedia Computing and Networking*, January 1996.

[19] Herman Towles, Sang-Uok Kum, Travis Sparks, Sudipta Sinha, Scott Larsen, and Nathan Beddes. Transport and rendering challenges of multi-stream 3D tele-immersion data. *NSF Lake Tahoe Workstop on Collaborative Virtual Reality and Visualization (CVRV 2003)*, October 2003.

[20] David Wetherall. Active network vision and reality: lessons from a capsule-based system. *Operating Systems Review*, 34(5):64–79, December 1999.