Aggregate Congestion Control for Distributed Multimedia Applications

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

***Abstract*— We consider the problem of applying aggregate congestion control to a class of distributed multimedia applications known as *Cluster-to-Cluster (C-to-C) applications*. Flows in such an application share a common intermediary path that is the primary source of network delay and packet loss.**

Using the *Coordination Protocol (CP)* architecture, we show how aggregate congestion control can be achieved with the following properties:

- **Almost any rate-based, single-flow congestion control algorithm may be applied to make aggregate C-to-C traffic congestion responsive.**
- **C-to-C applications may use multiple flow bandwidth shares and still exhibit correct aggregate congestion responsiveness.**
- **C-to-C applications may implement complex application-specific adaptation schemes in which the behavior of individual flows is decoupled from the behavior of the congestion responsive aggregate flow.**

***Bandwidth filtered loss detection (BFLD)* is presented as a technique for making single-flow loss detection algorithms work when aggregate traffic uses multiple flowshares. The approach is evaluated using both *ns2* simulation and an experimental implementation in FreeBSD and Linux. Results demonstrate its success for a wide range of network conditions.**

I. Introduction

As multimedia applications of the future become increasingly diverse and sophisticated, so too will their networking needs. Where one or two data streams was sufficient, future applications will require many streams to handle an ever-growing number of media types and modes of interactivity. Where the endpoints of communication were once single computing hosts, future endpoints will be collections of communication and computing devices. Examples of such applications include distributed sensor arrays, tele-immersion [1], computer-supported collaborative workspaces (CSCW) [2], ubiquitous computing environments [3], and complex multi-stream, multimedia presentations [4].

This work is supported by the National Science Foundation ITR Program (Award #ANI-0219780)

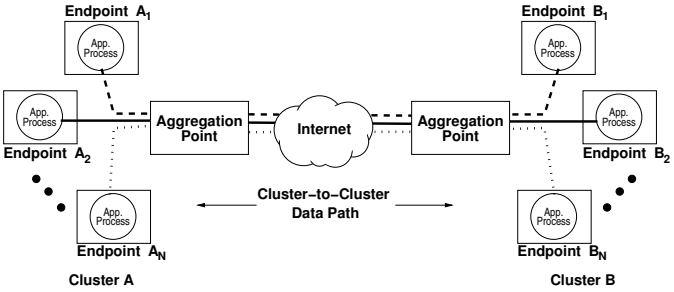

Fig. 1. C-to-C application model.

In this paper, we are interested in a class of distributed multimedia applications that we call *cluster-to-cluster (C-to-C) applications*. The hallmark of a C-to-C application is that it is distributed over many computing and communication devices, or *endpoints*, within some local environment and communicates with a set of endpoints located in some remote environment. Consider, as an example, a tele-immersion application. In such an application, tens of cameras are used to capture video data from a number of different angles and viewpoints. These video streams (along with other sensor information such as spatialized audio and 3D tracking information) are sent to a remote environment where they are consumed by a distributed set of processes which may, for example, be driving an immersive multi-projector 3D display.

C-to-C applications exhibit a number of interesting and important characteristics. Figure 1 illustrates the abstract C-to-C applicaiton model and some of its identifying characteristics, including:

- *A natural aggregation point.* Data communicated between clusters will typically pass through a natural *aggregation point (AP)* as data leaves one environment on route to the remote environment.
- *A common Internet path across flows.* While few flows in a C-to-C application share the exact same end-to-end path, all flows will share a common Internet path between clusters. While the network *within* a cluster can be provisioned to support the needs of the application, the path *between* clusters is shared with other Internet flows. We call this the *cluster-to-cluster data path*.
- *Independent, but semantically related flows of data.*

An application may need to priortize its many streams in a particular way, or divide complex media objects into multiple streams with specific temporal or spatial relationships. Furthermore, these relationships may be complex and dynamic.

- *Transport-level heterogeneity.* UDP- or RTP-based protocols, for example, might be used for streaming media while TCP is used to reliably transmit application control data.
- *Complex adaptation requirements.* Changes in available bandwidth require coordinated adaptation decisions that take into account the global objectives of the application, its current state, the nature of various flows, and relationship between flows.

An important issue for C-to-C applications is that of *congestion control*. While individual flows within the application may use a variety of transport-level protocols, including those without congestion control, it is essential that *aggregate application traffic* is congestion responsive [5].

While application traffic must be responsive to network congestion at an aggregate level, how this responsiveness is achieved should depend entirely on the application. Thus, the sending behavior of individual flows in response to congestion may vary widely according to an arbitrarily complex scheme defined by the application. For example, certain flows may halt sending altogether, while others make media encoding adjustments, and still others continue to send at their original sending rate. Important only is the effectiveness of the scheme in responding to changes in available bandwidth on an aggregate level.

This paper addresses the problem of applying congestion control to aggregate C-to-C application traffic. In particular, we are interested in leveraging existing single-flow congestion control schemes for C-to-C aggregate flows such that:

- *Cluster endpoints are informed of bandwidth available to the C-to-C application as a whole.*
- *Endpoints may respond to this information in application-defined ways.*
- *End-to-end semantics are preserved for each individual flow.*
- *Aggregate application traffic is congestion responsive.*

In addition, we believe that an aggregate congestion control scheme should support *multiple flowshares*. In other words, if we consider a single flowshare to be the bandwidth achieved by a single congestion-controlled flow (i.e., a single TCP connection), then a C-to-C application that involves multiple flows should receive multiple flowshares. Several approaches discussed in Section II ([6], [7], [8]) apply congestion control to aggregate flows such that the total bandwidth used is the equivalent of a single flowshare. We believe that this unduly restricts the bandwidth available to any given flow in a multi-flow application.

We propose applying congestion control to aggregate C-to-C application traffic such that an application with m flows may receive the *equivalent of* m *flowshares*. Furthermore, how available bandwidth is actually divided among flows is left entirely to the C-to-C application. For example, some application flows may take more than a single flowshare, while others take less. The decoupling of aggregate congestion control from individual flow behavior is a novel feature of our approach, and of tremendous utility to applications with diverse objectives and networking needs.

The main contributions of this paper are:

- *A protocol is described that supports global measurement of network conditions across all flows of a C-to-C application.* We call this protocol the *Coordination Protocol (CP)* because these measurements enable application endpoints to make coordinated adaptation decisions.
- *A method for applying rate-based, single-flow congestion control algorithms to aggregate C-to-C traffic is described and evaluated.* To illustrate, we implement and examine experimentally TFRC [9].
- *This method is extended to allow aggregate congestion control for the equivalent of* m *flowshares.* A new technique called *bandwidth filtered loss detection (BFLD)* is presented that allows bandwidth availability to be calculated correctly regardless of the aggregate sending rate.
- *We describe an implementation of our architecture using FreeBSD and Linux and evaluate its performance under various conditions.* Our results demonstrate the overall success of our approach.

The organization of this paper is as follows. Section II discusses various approaches to managing congestion control in flow aggregates. In Section III, we describe the *Coordination Protocol (CP)* and discuss how it supports the application of existing single-flow congestion control algorithms to the C-to-C application context. Simulation results evaluating these methods for a single flowshare is presented in Section IV. In Section V, we consider how this technique can be extended to support m flowshares. In Section VI, we describe our implementation of the CP and present performance evaluation results under various network conditions. Section VII summarizes this paper and discusses future directions.

II. Related Work

The problem of managing congestion control for flow aggregates has been addressed by a number of other researchers, most notably in the Congestion Manager (CM) work of Balakrishnan [6]. In this section, we discuss

several such approaches and assess their applicability to the C-to-C application context.

A. Flow Segmentation

One approach for applying congestion control to flow aggregates is to multiplex a single congestion responsive flow among individual flows sharing the same transmission path. In the C-to-C context, this could be done using a single flow between aggregation points, with an application- or transport-level multiplexer/demultiplexor at each AP. This approach is taken by [8] in their work on *TCP trunking* for connections that traverse a common backbone path.

Another variation of this approach known as *aggregated TCP (ATCP)* is presented in [7]. In this approach, multiple connections from a set of endpoints to a common remote endpoint are each divided into a *local subconnection* between an endpoint and its portal router and a shared *remote subconnection*

Whether executed at the application-level or transparently as in TCP-trunking, there are a number of problems with flow segmentation in the C-to-C context. First, the approach reduces aggregate application traffic to a single flowshare. We argue in Section I that limiting aggregate C-to-C application traffic to a single congestion responsive flow is unfairly restrictive in circumstances where the application employs numerous flows or is competing with numerous flows at the bottleneck link. Second, this approach fails to inform C-to-C application endpoints of aggregate networking performance. Without this information, application endpoints cannot fully exploit specific interstream adaptation schemes. Third, this approach may result in additional network delay as application packets are buffered at the trunk source waiting to be forwarded in a congestion controlled manner. Finally, end-to-end transport-level protocol semantics are not preserved for individual flows if communication is segmented into multiple connections (e.g., endpoint to AP, AP to AP, AP to endpoint).

B. Congestion Manager(CM)

The *congestion manager (CM)* architecture, proposed by Balakrishnan et al. in [6], provides a compelling solution to the problem of applying congestion control to aggregate traffic where flows share the same end-to-end path. Unlike the above schemes, CM emphasizes application control by informing flows of bandwidth available to them and avoiding the buffering of flow data during the forwarding process.

While the CM architecture proposes many useful concepts and mechanisms for managing congestion control for flow aggregates, we believe that it is not a good match for the C-to-C problem context as described in this paper.

First, CM's use of a flow scheduler to apportion bandwidth among flows is problematic. Because C-to-C applications can have complex schemes for accommodating ad hoc flow arrivals and departures, and for responding to changes in available bandwidth and changes in application state, we expect adaptation strategies to result in very dynamic rate adjustments for individual flows. Thus, characterizing each flow's rate requirements is difficult to do *a priori*. This kind of characterization is required with CM because individual flow rate requirements are reconciled within a hierarchical fair-service curve (HFSC) scheduler. The HFSC scheduler at the core of CM also serves to police the aggregate sending rate and ensures that the resulting traffic conforms to the calculated congestion controlled rate. Thus, while CM is able to take a set of individual flows that are well-characterized, and a set of static interflow priorities, and build a hierarchical schedule for bandwidth allocation, this approach is less suitable in the more dynamic C-to-C context.

Furthermore, CM is designed to multiplex a single congestion responsive flowshare among flows that have the same end-to-end path. Again, as in the multiplexing approach, it may be undesirable to constrain a C-to-C application to a single flowshare. Our solution allows aggregate C-to-C traffic to use multiple flowshares while remaining congestion responsive.

III. Coordination Protocol (CP)

In this section, we briefly describe our proposed solution, the Coordination Protocol (CP). Our focus here will be on CP mechanisms for aggregate congestion control. The reader is referred to [10], [11] for a more complete presentation of CP.

A. Overview

CP is implemented between the network layer (IP) and the transport layer (TCP, UDP, etc.). The network stacks of each cluster endpoint and their associated AP are modified to process CP packet headers, while all other nodes along the C-to-C data path require no special modifications. Figure 2 illustrates the CP architecture from a stack implementation point of view.

Using the CP header, a cluster AP identifies C-to-C application packets and attaches network probe information to each. The remote AP receives and processes

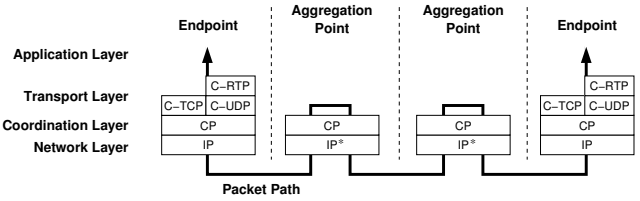

Fig. 2. CP network architecture.

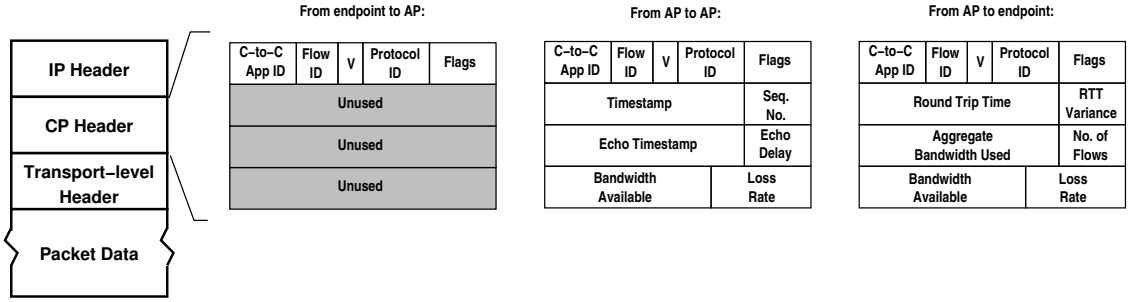

Fig. 3. CP packet structure.

this information. This exchange is bi-directional. By exchanging probe information in this manner, each AP builds a picture of current network conditions, including round-trip time (*RTT*) and loss rates for the application as a whole. This information, along with an estimated available bandwidth value, is passed to application endpoints using the CP header on a per-packet basis.

An AP uses aggregate measurements of RTT and loss to drive a rate-based congestion control algorithm (e.g., TFRC or RAP). Our design of CP allows a large class of congestion control algorithms to be used, bringing to bear the work of others instead of inventing new algorithms. The result of the congestion control algorithm is an ongoing aggregate sending rate calculation. This estimate predicts the bandwidth that would have be used by a single flow employing the same congestion control algorithm under similar network conditions.

When C-to-C endpoints receive this estimate, they respond by modifying their sending rate in an application-defined manner. A C-to-C application is free to employ any response scheme it wishes in order to realize an aggregate sending rate that reflects the bandwidth available to the application. In addition, the application need not limit itself to a single flowshare of bandwidth and may use up to m flowshares, where m is application-defined. Within this aggregate rate, applications are free to manage individual flows in any manner. In particular, individual flows may not be congestion responsive as long as application traffic as an aggregate is. In Section V we discuss how the use of multiple flowshares is realized in greater detail.

The benefits of this approach include:

- A *fast forwarding path* since traffic shaping and flow segmentation are avoided. APs do simple accounting across all application flows and a small number of calculations to obtain probe results.
- Application endpoints are informed of *aggregate bandwidth availability* on a per-packet basis.
- *Preserved end-to-end semantics* for transport-level protocols.
- *Complete application control* over the manner in which an aggregate congestion response is realized.
- Support for *multiple flowshares*.

B. Why A New Protocol Layer?

The decision to insert CP between the network and transport layers requires some justification. First, we note that placing CP below the transport-layer preserves the end-to-end semantics of individual transport-level protocols. Second, we argue that CP logically belongs in this position because managing the aggregate C-to-C application traffic is conceptually above the next-hop forwarding concerns of IP and below the end-to-end concerns of the transport layer. Third, application-layer handling of CP packets at the AP would affect forwarding performance.

We point out, however, that our decision is merely one of implementation. It is certainly possible to implement the mechanisms we describe at the application-level. Indeed, Section VI describes a hybrid UDP-based implementation using CP headers nested within UDP packet data and "deep" processing by kernel-level forwarding code at the APs.

C. CP Operation

Figure 3 shows a CP data packet. CP encapsulates transport-level packets by prepending a 16-byte header and, in turn, IP encapsulates CP packets. Each CP header contains an application identifier associating the packet with a C-to-C application, allowing the AP to identify which packets are part of an aggregate flow. The header also contains a version number and a flags field. The remaining contents of the CP header vary according to the changing role played by the header as it traverses the network path from source endpoint to destination endpoint.

The basic operation of CP is as follows:

- **As packets originate from source endpoints:** The endpoint stack places information in the CP header identifying the C-to-C application and flow.
- **As packets arrive at the local AP:** The AP processes the identification information arriving in the CP header. Bandwidth usage statistics and other state information associated with the C-to-C application are updated. Part of the CP header is overwritten, allowing the AP to communicate congestion probe information to the remote AP. As

the packet is forwarded to the remote AP, the header now contains timestamps used to measure RTT, a sequence number to detect losses, and loss rate and available bandwidth estimates.

- **As packets arrive at the remote AP:** The CP header is used to measure network delay and loss. Again, part of the CP header is overwritten, this time to communicate network condition information, aggregate bandwidth usage, and other aggregate measures of performance to the remote endpoint.
- **As packets arrive at the destination endpoint:** The endpoint stack processes network condition information from the CP header and makes it available to the transport-level protocol and the application.

D. Aggregate Congestion Control

Implementing aggregate congestion control in CP involves several mechanisms. The APs use fields in the CP header to measure RTT and detect loss. In addition, the APs maintain an average packet size calculation. This information is made available to the congestion control algorithm. The algorithm is expected to estimate the available bandwidth for a single flowshare. The estimate is maintained by the *receiving* AP. For example, in Figure 1, the AP for Cluster B maintains an estimate for available bandwidth from Cluster A to Cluster B and reports this estimate back to endpoints in Cluster A within the CP header of packets traveling back in the other direction. In the same manner, Cluster A maintains an estimate of available bandwidth from Cluster B to Cluster A.

To measure RTT, the AP's use a timestamp-based mechanism. An AP inserts a timestamp into each packet which is echoed along with the delay since that timestamp was received. When the echo is received by the original AP, a RTT sample is constructed as *RTT = current time - timestamp echo - echo delay*. The RTT sample is used to maintain a smoothed weighted average estimate of RTT and RTT variance.

To detect loss, each AP inserts a monotonically increasing sequence number in the CP header. At the receiving AP, losses are detected as a gap in the sequence number space. These losses are reported to the congestion control algorithm and a smoothed average loss rate is maintained.

CP can employ any rate-based congestion control algorithm that uses the current RTT, mean packet size, and individual loss events or loss rates as basic building blocks. We illustrate this in Section IV where our implementation of TFRC is described in some detail.

E. Transport-level Protocols

Transport-level protocols are built on top of CP. We have initially considered coordinated versions of TCP (C-TCP) and UDP (C-UDP) implemented using a modified socket API.

With C-UDP, the application is provided an interface to set the C-to-C application id and flow id, and get the latest estimated RTT, aggregate loss rate, and estimated available bandwidth. The application is responsible for adapting its packet send rate based on this information.

Our coordinated version of TCP (C-TCP) provides the same end-to-end semantics as TCP (i.e., a reliable byte stream), but relies on the underlying CP protocol to detect congestion and suggest an appropriate sending rate. The application can attenuate the suggested congestion-controlled rate by setting a scale factor.

F. Exploiting CP

A C-to-C application may configure its endpoints to respond to changes in bandwidth availability (as well as other information in the CP header) in any way it chooses and modify the configuration at will. For example, it need not be the case that each endpoint responds in a uniform manner, or even that all flows respond. An application may instead realize a congestion-controlled aggregate send rate by backing off or terminating some flows, but not others.

Likewise, how an application implements dynamic endpoint configuration is left entirely up to the application itself. Some applications may be statically configured from the onset. Others may employ a centralized control process which interprets changing network information and periodically sends configuration messages to each endpoint. Still others may employ a decentralized approach in which endpoints independently evaluate application and network state information and make send rate adjustments accordingly.

IV. Single Flowshares

In this section, we describe our implementation of CP in *ns2* [12] and discuss simulation results for a mock C-to-C application configured to send at an aggregate rate equivalent to a **single** flowshare. Our results show that CP performs well when compared to competing flows of the same protocol type.

A. CP-TFRC

We refer to our *ns2* implementation of the TFRC congestion control algorithm in CP as *CP-TFRC*. (Full details of the TFRC algorithm can be found in [13].) For CP-TFRC, a loss rate is calculated by constructing a loss history and identifying *loss events*. These events are then converted to a *loss event rate*. Smoothed RTT,

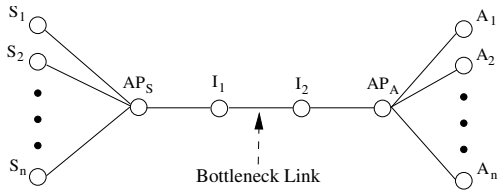

Fig. 4. Simulation testbed in ns2.

Parameter	Value
Packet size	1 K
ACK size	40 B
Bottleneck delay	50 ms
Bottleneck bandwidth	15 Mb/sec
Bottleneck queue length	300
Bottleneck queue type	RED
Simulation duration	180 sec

TABLE I
CONFIGURATION PARAMETERS.

loss event rate, and various other values are then used as inputs into the equation [13]:

$$X = \frac{s}{R\sqrt{\frac{2bp}{3}} + t_{RTO}(3\sqrt{\frac{3bp}{8}})p(1+32p^2)} \quad (1)$$

which calculates a TCP-compatible transmission rate X (bytes/sec) where s is the packet size (bytes), R is the round trip time (sec), p is the loss event rate, t_{RTO} is the TCP retransmission timeout (sec), and b is the number of packets acknowledged by a single TCP acknowledgement. Updates in bandwidth availability are made at a frequency of once every RTT. Bandwidth availability is estimated at the remote AP. The resulting bandwidth availability value is placed in the CP header on the reverse path, and simply forwarded by the local AP to application endpoints.

B. Configuration

Figure 4 shows our ns-2 simulation topology. Sending agents, labeled S_1 through S_n, transmit data to AP_S where it is forwarded through a bottleneck link to remote AP_A and ACK agents A_1 through A_n. For any given simulation, the bottleneck link between I_1 and I_2 is shared by CP flows transmitting between clusters and competing (i.e., non-CP) TFRC flows. Table I summarizes topology parameters. Links between ACK agents A_1 through A_n are assigned delay values that vary in order to allow some variation in RTT for different end-to-end flows.

Flows in our simulated C-to-C application are configured to take an equal portion of the current bandwidth available to the application. That is, if n C-to-C endpoints share bandwidth flowshare B, then each endpoint sends at a rate of B/n. More complex configurations are possible, and the reader is referred to [11] for further illustrations.

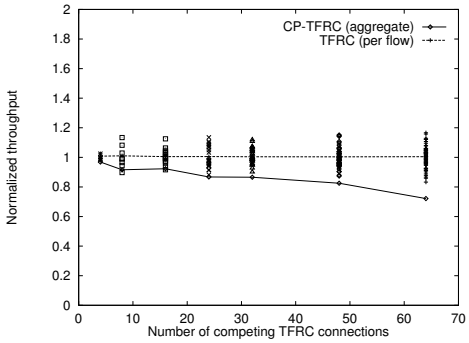

Fig. 5. TFRC versus CP-TFRC normalized throughput as the number of competing TFRC flows is varied.

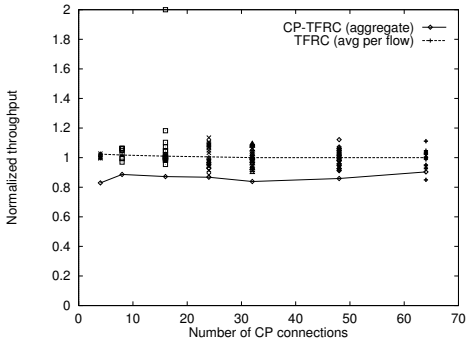

Fig. 6. TFRC versus CP-TFRC normalized throughput as the number of flows in the C-to-C aggregate is varied.

C. Evaluation

Our goal in this section is to compare aggregate CP-TFRC traffic using a single flowshare with competing TFRC flows sharing the same C-to-C data path. Our concern is not evaluating the properties (e.g., TCP-compatibility) of the TFRC congestion control scheme, but rather examining how closely C-to-C aggregate traffic conforms to TFRC bandwidth usage patterns. The question of how well CP-TFRC performs with respect to competing TCP flows is left to Section VI

In Figure 5, a mock C-to-C application consisting of 24 flows competes with a varying number of TFRC flows sharing the same cluster-to-cluster data path. Throughput values have been normalized so that a value of 1.0 represents a fair throughput level for a single flow.

The performance of TFRC flows is presented in two ways. First, normalized bandwidth of a single run is presented as a series of points representing the normalized bandwidth received by a each competing flow. These points illustrate the range in values realized within a trial. Second, a line connects points representing the *average* (mean) bandwidth received by competing TFRC flows across 20 different trials of the same configuration.

The CP-TFRC line connects points representing the *aggregate* bandwidth received by 24 CP flows averaged over 20 trials. For each each trial, this aggregate flow competes as only a single flowshare within the simulation. We see from this plot that as the number of

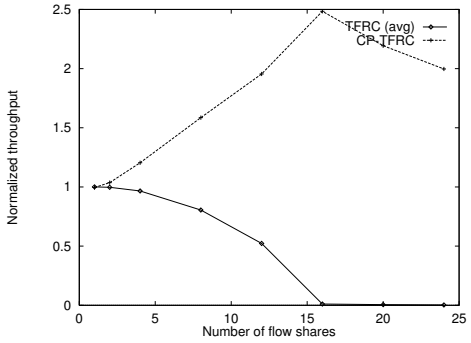

Fig. 7. Throughput for multiple flowshares (naive approach).

competing TFRC flows increases, C-to-C flows receive only slightly less than their fair share.

Figure 6 shows per-flow normalized throughput when the number of competing TFRC flows is held constant at 24, and the number of CP flows is increased, but still sharing a single flowshare. Again aggregate CP traffic received very close to its fair share of available bandwidth, with normalized values greater than 0.8 throughout.

V. Multiple Flowshares

In this section, we consider the problem of supporting multiple flowshares. While numerous approaches for applying aggregate congestion control using single flowshares have been suggested as reviewed in Section II, we are unaware of any approach that considers the multiple flowshare problem. The reason for this is that single-flow congestion control algorithms break when a sender fails to limit their sending rate to the rate calculated by the algorithm. Here we use simulation to show how this is the case for CP-TFRC. After discussing the problem in some detail, we present a new technique, *bandwidth filtered loss detection (BFLD)* and demonstrate its effectiveness in enabling multiple flowshares.

A. Naive Approach

Our goal in this section is to allow C-to-C applications to send the equivalent of m flowshares in aggregate traffic, where m is equal to the number of flows in the application. As mentioned in Section I, we believe that limiting a C-to-C application to a single flowshare may unfairly limit bandwidth for an application that would otherwise employ multiple independent flows.

A *naive approach* for realizing multiple flowshares is simply to have each C-to-C application endpoint multiply the estimated bandwidth availability value B by a factor of m. Thus, each endpoint behaves as if the bandwidth available to the application as a whole is mB. One could justify such an approach by arguing that probe information exchanges between APs maintain a closed feedback loop. That is, an increase in aggregate sending rate beyond appropriate levels will result in increases in network delay and loss. In turn, this will cause calculated values of B to decrease, thus responding to current network conditions. Ideally B would settle on some new value which, when multiplied by m, results in the appropriate congestion-controlled level that would have otherwise been achieved by m independent flows.

Figure 7 shows that this is not the case. For each simulation, the number of CP-TFRC and competing TFRC flows is held constant at 24. The number of flowshares used by CP-TFRC traffic is then increased from k = 1 to m using the naive approach. The factor k is given by the x-axis. The normalized fair share ratio (with 1.0 representing perfect fairness) is given by the y-axis.

In Figure 7, increases in the number of flowshares cause the average bandwidth received by a competing TFRC flow to drop unacceptably low. By $k = 16$, TFRC flows receive virtually *no* bandwidth, and beyond $k = 16$, growing loss rates eventually trigger the onset of congestion collapse. Additional simulation work with RAP [14] (not presented in this paper) likewise shows unacceptable results, although with a somewhat different pattern of behavior suggesting that different congestion control schemes result in different types of failure.

B. The Packet Loss Problem

In the case of CP-TFRC, recall that RTT and loss event rates are the primary inputs to equation 1. We note that increasing the C-to-C aggregate sending rate should have no marked effect on RTT measurements since APs simply use any available CP packets for the purpose of probe information exchanges. In fact, increasing the number of available packets should make RTT measurements even more accurate since more packets are available for probing.

On the other hand, we note that a large increase in C-to-C aggregate traffic has a drastic effect on *loss event rate* calculations in CP-TFRC. TFRC marks the beginning of a *loss event* when a packet loss P_i is detected. The loss event ends when, after a period of one RTT, another packet loss P_j is detected. An *inter-loss event interval* I is calculated as the difference in sequence numbers between the two lost packets ($I = j - \mathrm{i}$) and, to simplify somewhat, a rate R is calculated by taking the inverse of this value ($R = 1/I$). Here we note that the effect of drastically increasing the number of packets in the aggregate traffic flow is to increase the inter-loss event interval I; while the likelihood of encountering a packet drop soon after the RTT damping period has expired increases, the number of packet arrivals during the damping period also increases. The result is a *larger* interval, or a smaller loss event rate, and hence an inflated available bandwidth estimation. This situation is depicted in Figure 8.

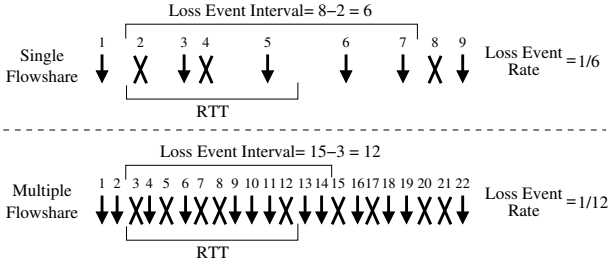

Fig. 8. Loss event rate calculation for TFRC.

In a sense, the algorithm suffers from the problem of inappropriate feedback. For CP-TFRC, too many packets received in the damping period used to calculate a loss event rate artificially inflates the inter-loss event interval. The algorithm has been tuned for the *appropriate* amount of feedback which would be generated by a packet source that is conformant to a single flowshare only.

C. BFLD

Our solution to the problem of loss detection in a multiple flowshare context is called *bandwidth filtered loss detection (BFLD)*. BFLD works by sub-sampling the space of CP packets in the network, effectively reducing the amount of loss feedback to an appropriate level. Essentially, the congestion control algorithm is driven by a "virtual" packet stream which is stochastically sampled from the actual aggregate packet stream.

BFLD makes use of two different bandwidth calculations. First is the *available bandwidth*, or B_{avail}, which is calculated by the congestion control algorithm employed at the AP. This represents the congestion responsive sending rate for a single flowshare. Second is the *arrival bandwidth*, or B_{arriv}. The value B_{arriv} is an estimate of the bandwidth currently being generated by the C-to-C application.

From these values, a *sampling fraction* F is calculated as $F = B_{avail}/B_{arriv}$. If $B_{avail} > B_{arriv}$, then F is set to 1.0. Conceptually, this value represents the fraction of arriving packets and detected losses to sample in order to create the virtual packet stream that will drive the congestion control algorithm. We refer to this virtual packet stream as the *filtered packet event stream*.

To determine whether a packet arrival or loss should be included in the filtered packet event stream, a simple stochastic technique is used. Whenever a packet event occurs (i.e., a packet arrives or a packet loss is detected), a random number r is generated in the interval $0 \leq r \leq 1.0$. If r is in the interval $0 \leq r \leq F$ then an event is generated for the virtual packet event stream, otherwise no virtual packet event is generated.

Packets chosen by this filtering mechanism are given a virtual packet sequence number that will be used by the congestion control algorithm for loss detection, computing loss rates, updating loss histories, etc. Figure 9 illustrates the effect of this process. In this figure, we see that

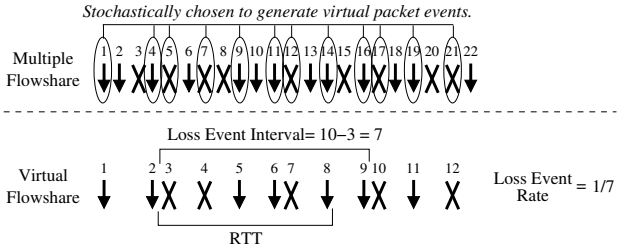

Fig. 9. Virtual packet event stream construction by BFLD.

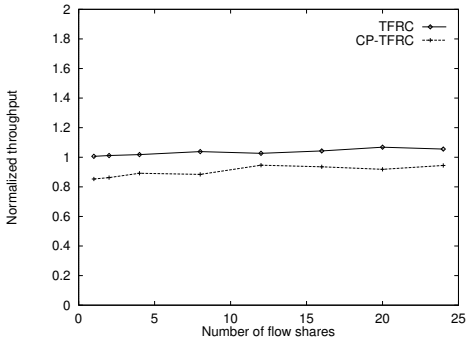

Fig. 10. Throughput for multiple flowshares using BFLD.

a subset of the multiple flowshare packet event stream is stochastically chosen to generate a virtual packet event stream. In this stream, we see virtual sequence numbers assigned to these packet events. As a result, the TFRC calculation for the loss event interval decreases from 12 to 7 remedying the problem illustrated in Figure 8. An interesting feature of this technique is that it can be applied *regardless of the number of flowshares* used by the C-to-C application. This is because the factor F adjusts with whatever the amount of bandwidth used.

D. Evaluation

Figure 10 shows the results of applying BFLD to the simulations of Figure 7 in Section V-A. As before, the number of CP-TFRC flows and competing TFRC flows are both held constant at 24, while the number of flowshares taken by CP-TFRC traffic as an aggregate is increased from $k = 1$ to m. The results show a dramatic improvement. Normalized throughput for CP-TFRC flowshares is consistently close to .9 while throughput levels achieved by competing TFRC flows are consistently close to 1.0.

VI. Implementation and Evaluation

In this section, we briefly describe our implementation of the Coordination Protocol using FreeBSD and Linux, including packet header placement, router modifications, application API, endpoint traffic generation, and experimental setup. We then go on to present results showing how BFLD performs in an experimental network with competing TCP connections and various levels of network delay, bottleneck bandwidth, random loss, and

background traffic loads. Overall, we find that CP does quite well in maintaining TCP-compatibility under a wide variety of network conditions.

A. Implementation

Our implementation of the CP architecture is a compromise between the approach described in Section III and an application-level approach. The implementation uses UDP packets with CP packet headers nested within the first 20 bytes of application data. Using UDP allowed us to avoid the requirement that application endpoints have modified network stacks.

While the endpoint implementation is handled at the application level, the AP implementation is handled at the kernel level using a dynamically loadable kernel module written for FreeBSD version 4.7. This module extends IP forwarding capabilities of first and last hop routers to provide full AP functionality. The module is configured to recognize UDP packets from particular source-destination host pairings as CP packets, triggering "deep processing" of the CP packet header nested within UDP application data. All state maintained at the AP is "soft" (i.e., created on demand and torn down by timeout).

An application-level library provides a thin layer of indirection within application send and receive calls at the endpoints. For send calls, the libary handles packetization and inserts a CP header at the beginning of each send buffer. For receive calls, the library first removes and processes the CP header, then passing data to the application level. API calls are provided that allows the application to query newtork and flow information.

To drive the system, we constructed a test application comprised of two endpoint clusters exchanging data as infinite data sources. Each endpoint acts essentially as a rate-based traffic generator, sending mock data to a remote endpoint at a rate equal to kB where B is the available bandwidth reported by CP and k is a multiplicative factor and input parameter. Our test application lacks the rich semantic relationships seen in real-world distributed multimedia applications, but provides us with the tools we need to verify system correctness and study overall AP performance. Endpoint hosts include both Linux hosts (version 2.4) and FreeBSD hosts (version 4.5).

B. Experimental Setup

Our experimental network is shown in Figure 11. 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

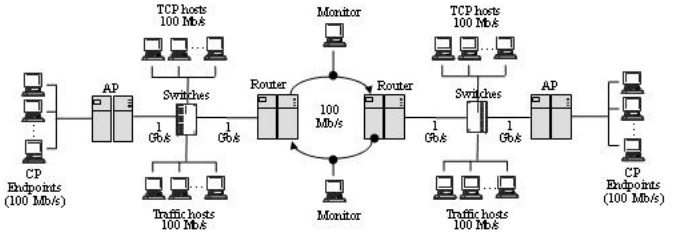

Fig. 11. Experimental network setup.

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.

Competing TCP flows are generated by TCP hosts on opposite sides of the network. These hosts use the well-known utility *iperf* [15] to generate long-lived flows with unlimited data. Each host is connected to its local switch using 100 Mb/s Fast Ethernet. TCP flows share the same bottleneck link with CP flows and thus compete with them for bandwidth.

Also sharing the bottleneck link are background flows between traffic hosts on each end of the network. More will be said about these flows in Section VI-G.

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* [16] 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.

C. Performance metrics

Overall, our goal is to compare aggregate CP flow performance to that of TCP under various network conditions to see whether the CP architecture can successfully maintain compatibility when the number of flowshares is scaled. Toward this end, we make use of two comparative metrics closely related to those described in [9].

First is *normalized throughput ratio* defined as the ratio of normalized average throughput for a single TCP flow to the normalized average throughput for a single CP flowshare.

$$R_{TCP,CP} = \frac{F_{TCP}}{F_{CP}} \tag{2}$$

Here F_{TCP} and F_{CP} are normalized flowshares as defined in Section IV-C and represent the average throughput for a single TCP flow or CP flowshare, normalized so

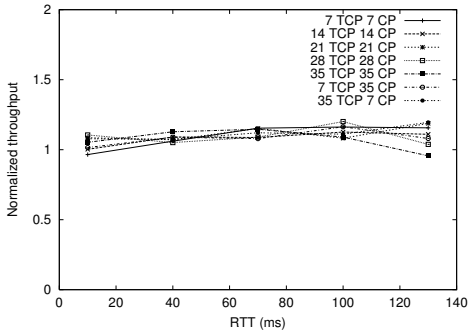

Fig. 12. Normalized throughput ratio as delay varies.

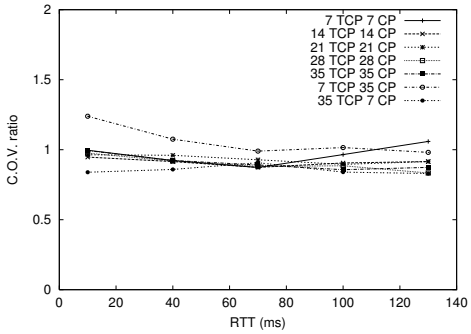

Fig. 13. C.O.V. ratio as delay varies.

that 1.0 is an ideal fair share. A value greater than 1.0 indicates that TCP flows on an average have received more bandwidth than CP flowshares, while for values less than 1.0 the reverse is true.

The second metric is the *coefficient of variance (C.O.V.) ratio* and is meant to compare the degree of throughput variation seen in aggregate TCP and CP traffic:

$$C.O.V._{TCP,CP} = \frac{C.O.V._{TCP}}{C.O.V._{CP}} \quad (3)$$

C.O.V. [17] is computed as the standard deviation of aggregate throughput samples for TCP or CP divided by the mean. A value greater than 1.0 indicates that more variance is seen in aggregate TCP throughput samples than in CP, while for values less than 1.0 the reverse is true.

D. Delay experiments

To test CP under various network delay conditions, we made use of the *dummynet* [18] traffic shaper 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.

For this set of experiments, we configured *dummynet* on the two bottleneck routers to simulate a range of combined propogation delays between 10 and 130 ms.

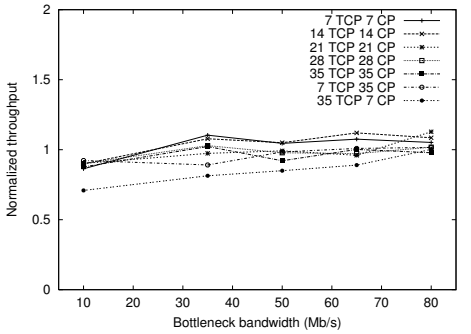

Fig. 14. Normalized throughput ratio as bottleneck bandwidth varies.

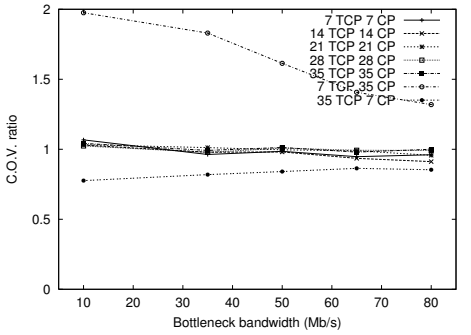

Fig. 15. C.O.V. ratio as bottleneck bandwidth varies.

Various combinations of CP and TCP flows are run to explore the effects of scaling (7-7, 14-14, 21-21, 28-28, and 35-35) and unequal flow distributions (7-35 and 35-7) on CP performance. For each combination, each of m CP flows sends at the reported bandwidth availability rate, for a total of m flowshares of aggregate C-to-C traffic.

Runs lasted for four minutes and begin after a 20 second ramp-up and stabilization period. Trials using a longer ramp-up and run interval did not show significantly different results. *Dummynet* loss rates were held constant at 1%.

Figure 12 shows normalized throughput results. In general, values remain very close to 1.0 for all trials, with TCP receiving slightly more bandwidth. C.O.V. ratios in Figure 13 likewise remain fairly close to 1.0 but show somewhat more variance in TCP within the 7-35 unequal flow distribution set.

E. Bottleneck bandwidth experiments

To test CP under conditions of various bottleneck bandwidths, we again used *dummynet* on the bottleneck FreeBSD routers. This time we varied the bottleneck bandwidth configuration from 10 to 80 Mb/s, meanwhile maintaining a constant 40 ms round trip time and 1% loss rate.

Normalized throughput results in Figure 14 are fairly close to 1.0 and consistent across all sets, although CP received somewhat more bandwidth in the 35-7 unequal flow distribution set–especially at the smallest bottleneck

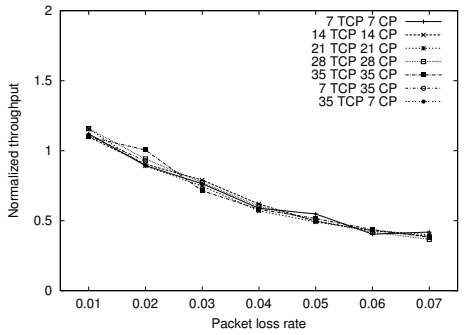

Fig. 16. Normalized throughput ratio as random loss varies.

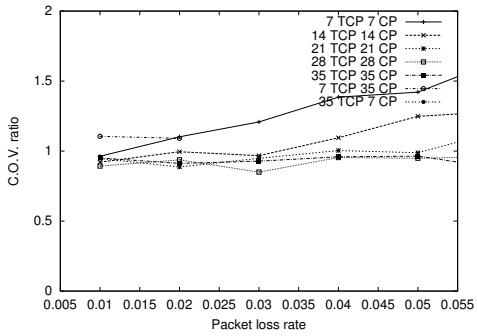

Fig. 17. C.O.V. ratio as random loss varies.

bandwidth levels. Figure 15 shows a very balanced throughput variance for all equal flow distributions, and a strikingly unbalanced throughput variance for unequal flows distributions. In particular, the 7-35 set shows TCP flow throughput variation to be nearly double that of CP. For the 35-7 set, CP shows significantly more variation.

F. Random loss experiments

To test CP under various loss levels we once again used the *dummynet* traffic shaper on bottleneck FreeBSD routers. We varied random loss levels from 1 to 5%, meanwhile maintaining a constant 40 ms round trip time.

Normalized throughput results in Figure 16 show a marked drop in ratio values as loss levels are increased, indicating that TCP is increasingly losing bandwidth to CP. This is a known problem with TFRC that has been described in [19]. Widmer theorizes that higher packet loss rates increasingly interfere with TCP's ability to maintain self-clocking since timeouts become more frequent. SACK TCP would likely perform better than FreeBSD's New Reno implementation but unfortunately is not supported by FreeBSD version 4.5.

G. Traffic 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 this dynamic, we tested CP performance against various background traffic workloads using a Web traffic generator known as $thttp$.

$Thttp$ uses empirical distributions from [20] 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 the behavior of 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 11. Emulated Web traffic was given a 20 minute ramp-up interval and competed with TCP and CP flows on the bottleneck link in both directions. We varied the number of browsers emulated from 1000 to 6000 and ran experiments focusing on 14-14 and 35-35 flow configurations. Resulting loss rates are shown in Figure 18 as measured at bottleneck router queues.

Figure 19 shows normalized throughput ratios for both experiment sets. Results look much improved over *dummynet* random loss trials shown in Figure 16. TCP flows average slightly more bandwidith than CP flowshares at low load levels for the 35-35 set, while the reverse is true for the 14-14 set. C.O.V. ratio results in Figure 20 show very similar levels of throughput variation in TCP and CP, with only a slight difference at the lowest load levels.

VII. Summary and Future Work

In this paper, we have discussed the need for aggregate congestion control for a class of distributed multimedia applications call cluster-to-cluster (C-to-C) applications. The Coordination Protocol (CP) was presented as a

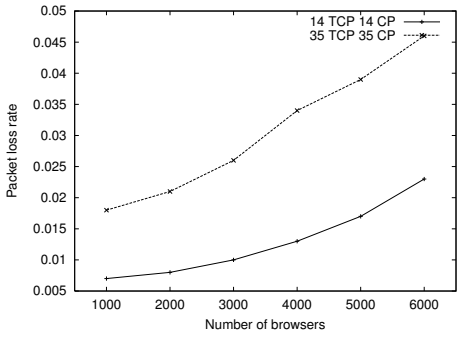

Fig. 18. Loss rates generated by background web traffic.

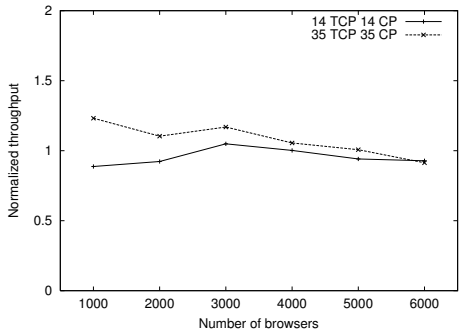

Fig. 19. Normalized throughput ratio as competing load varies.

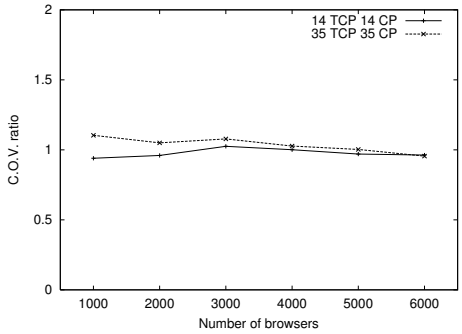

Fig. 20. C.O.V. ratio as competing load varies.

framework that makes possible the application of rate-based, single-flow congestion control schemes to this context. It does this by providing network probe mechanisms which measure RTT and packet loss for aggregate application traffic traversing the shared intermediary path. Using this information, CP estimates an available bandwidth for a single flowshare and informs application endpoints of this value.

We have shown how this framework can be extended to support *multiple flowshares*. In particular, we show that:

- Single flow congestion control algorithms do not scale naively to support multiple flowshares.
- *Bandwidth filtered loss detection (BFLD)* is a technique for stochastically sampling a packet arrival event stream to provide single flow congestion control algorithms with an appropriate amount of loss feedback.
- Using BFLD, aggregate C-to-C traffic can effectively realize multiple flowshares.

After demonstrating that CP performs reasonably well when compared to TFRC using *ns2* simulation, we go on to evaluate the performance of an actual CP implementation using FreeBSD and Linux under a wide variety of network conditions. Our results show the overall success of our approach.

Finally, an issue we have considered for future work is the use of wireless endpoints within a C-to-C application cluster. In this case, the assumption that endpoint-to-AP communication takes place with little loss or delay is not true. One idea is to design application endpoints and/or transport-level protocols that can use the CP framework to discriminate between *local* (i.e., wireless) and *AP-to-AP* sources of delay and loss. This can be done by comparing end-to-end measurements of network conditions with reported CP measurements and using discrepancies as an indication of conditions on the wireless portion of the path.

REFERENCES

[1] Ramesh Raskar, Greg Welch, Matt Cutts, Adam Lake, Lev Stesin, and Henry Fuchs, "The office of the future: A unified approach to image-based modeling and spatially immersive displays," *Proceedings of ACM SIGRAPH 98*, 1998.

[2] J. Grudin, "Computer-supported cooperative work: its history and participation.," *Computer*, vol. 27, no. 4, pp. 19–26, 1994.

[3] M. Weiser, "Some computer science problems in ubiquitous computing.," *Communications of the ACM*, vol. 36, no. 7, pp. 75–84, July 1993.

[4] T.-P. Yu, D. Wu, K. Mayer-Patel, and L.A. Rowe, "DC: A live webcast control system," *Proc. of SPIE Multimedia Computing and Networking*, 2001.

[5] Sally Floyd and Kevin R. Fall, "Promoting the use of end-to-end congestion control in the internet," *IEEE/ACM Transactions on Networking*, vol. 7, no. 4, pp. 458–472, 1999.

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

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

[8] H.T. Kung and S.Y. Wang, "TCP trunking: Design, implementation and performance.," *Proc. of ICNP '99*, November 1999.

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

[10] 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.

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

[12] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu, "Advances in network simulation," *IEEE Computer*, vol. 33, no. 5, pp. 59–67, May 2000.

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

[14] R. Rejaie, M. Handley, and D. Estrin, "RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet," *Proc. of IEEE INFOCOM*, March 1999.

[15] ," http://dast.nlanr.net/Projects/Iperf/.

[16] C. Kenjiro, "A framework for alternate queueing: Towards traffic management by pc-unix based routers," in *USENIX 1998*, June 1998, pp. 247–258.

[17] R. Jain, *The Art of Computer Systems Performance Analysis*, John Wiley and Sons, 1991.

[18] Luigi Rizzo, ," http://info.iet.unipi.it/ luigi/ip_dummynet/.

[19] Jorg Widmer, *Equation-Based Congestion Control*, Ph.D. thesis, University of Mannheim : Dept of Mathematics and Computer Science, February 2000.

[20] 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*, June 2001, pp. 245–256.